HAL
open science

Interactive Program Restructuring

Oleksandr Zinenko

To cite this version:

Oleksandr Zinenko. Interactive Program Restructuring. Human-Computer Interaction [cs.HC]. Université Paris Saclay (COmUE), 2016. English. NNT : 2016SACLS437 . tel-01414770

HAL Id: tel-01414770

https://theses.hal.science/tel-01414770v1

Submitted on 12 Dec 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

NNT : 2016SACLS437

THÈSE DE DOCTORAT
DE
L'UNIVERSITÉ PARIS-SACLAY
PRÉPARÉE À
L'UNIVERSITÉ PARIS-SUD

ECOLE DOCTORALE N° 580
Sciences et Technologies de l'Information et de la Communication

Spécialité de doctorat : Informatique

Par

M. Oleksandr ZINENKO

Restructuration Interactive des Programmes

Thèse présentée et soutenue à Gif-sur-Yvette, le 25 Novembre 2016.

Composition du Jury :

M. FEKETE Jean-Daniel	Directeur de recherche à Inria Saclay	Président
M. CONVERSY Stéphane	Professeur à l'ENAC et Université de Toulouse	Rapporteur
M. RAJOPADHYE Sanjay	Professeur à Colorado State University	Rapporteur
M. COHEN Albert	Directeur de recherche à Inria Paris et ENS Paris	Examinateur
M. HUOT Stéphane	Directeur de recherche à Inria Lille	Directeur de thèse
M. BASTOUL Cédric	Professeur à l'Université de Strasbourg	Co-directeur de thèse

Oleksandr Zinenko: *Interactive Program Restructuring*, © November 25, 2016

RÉSUMÉ DE THÈSE

La programmation est une activité complexe et hautement spécialisée qui nécessite beaucoup d'effort cognitif de la part de programmeur. Tout au long de leur histoire, les ordinateurs sont passés des systèmes disparates et très spécialisés aux systèmes homogènes à l'usage général, et sont revenus à la multitude des systèmes extrêmement différents grâce à l'avènement de l'informatique ubiquitaire. Les moyens de contrôler le fonctionnement des ordinateurs ont évolué au même temps que les architectures matérielles entraînant la création des abstraction empilées nécessaires pour gérer la complexité constamment croissante des systèmes informatiques. Cependant ces abstractions ont eu un coût de calcul non-négligeable diminuant l'efficacité des ordinateurs : les programmeurs n'ont plus de moyens, techniques ou temporelles, pour gérer les nombreux aspects du fonctionnement de l'ordinateur et doivent recourir aux approches génériques qui sont souvent suboptimales.

Des langages de programmation et des bibliothèques logicielles actuels aident à *créer* des nouveaux programmes efficaces, des outils de visualisation des logiciels permettent d'*analyser* l'efficacité des programmes, cependant aucun d'entre eux n'est conçu spécialement pour supporter la *restructuration* des programmes existants. La restructuration interactive des programmes est une interface interactive unifiant des visualisations des logiciels efficaces et des outils de haut niveau pour manipuler des programmes qui réduit à la fois le temps d'analyse et celui de modification des programmes. Les outils efficaces de restructuration des programmes doivent représenter les données sur le programme qui sont soit indisponibles soit difficilement accessibles dans des autres représentations, y compris dans le code source, ainsi que permettre la modification de ces données de telle sorte que le programme représenté soit aussi modifié. La conception de tels outils nécessite non seulement les techniques de visualisation et d'interaction appropriée, mais aussi des modèles des programmes puissants qui adaptés à la restructuration profonde.

Dans cette thèse, nous développons la *restructuration interactive des programmes*, proposons les principes et les directives pour la conception des outils de restructuration des programmes et utilisons ces principes pour évaluer un système interactif de parallélisation et d'optimisation des programmes basé sur le modèle polyédrique, l'état de l'art de représentation des programmes de calcul scientifique.

Chapitre 2 propose une revue de l'état de l'art des représentations et des visualisations des logiciels. Dans ce chapitre, nous établissons une *espace de conception* qui organise les outils et les techniques de vi-

sualisation des logiciels selon les informations qu'ils représentent et les interactions qu'ils supportent au niveau de modification des programmes. S'appuyant sur l'analyse morphologique des visualisations des logiciels, nous proposons les principes de base de la restructuration interactive des programmes.

Dans le chapitre 3, nous décrivons le *modèle polyédrique*, une représentation abstraite moderne des programmes qui fournit les outils d'analyse précis et les techniques de manipulation de haut niveau. Nous définissons la *structure de la représentation polyédrique* qui convient pour la manipulation interactive des programmes.

Chapitre 4 décrit la conception orientée utilisateur d'une technique de visualisation des logiciels basée sur le modèle polyédrique. Nous démontrons comment cette visualisation fournit, d'une manière compréhensible mais brève, l'information sur l'exécution d'un programme difficilement visible dans le code source, notamment les exécutions individuelles des boucles et les dépendances entre elles. Cette technique de visualisation est évaluée dans une expérience contrôlée avec un groupe des programmeurs ayant des niveaux différents. Le résultats décrits dans ce chapitre étaient présentés à la conférence VL/HCC en 2014.

Dans le chapitre 5, nous proposons une correspondance bidirectionnelle entre les transformations primitives dites «classiques» des boucles et la représentation polyédrique. Le nouveau ensemble des transformations permet de combiner les transformation primitives afin d'effectuer toute modification possible tant que le programme reste valable dans le modèle polyédrique. Nous proposons aussi un algorithme pour retrouver une séquence des transformations primitives qui correspond à une optimisation d'un programme calculée automatiquement sans que l'optimiseur automatique aient connaissance des nos primitives ni fonctionne au niveau syntaxique. Ensemble, l'ensemble des transformations et l'algorithme permettent à l'utilisateur d'interagir avec un compilateur polyédrique. Les résultats décrits dans ce chapitre étaient présentés à la conférence CGO en 2016.

Chapitre 6 s'appuie sur les deux chapitres précédents afin de permettre *manipulation directe* de la visualisation des programmes. Nous mettons en correspondance les entrées graphiques de l'utilisateur et les séquences de transformations primitives ainsi que les transformations primitives individuelles et les visualisations animées. Le système de manipulation des programmes ainsi conçu est évalué expérimentalement pour démontrer son utilité. Dans ce chapitre, nous étudions également l'utilisation les représentations textuelles et visuelles des programmes et les choix des programmeur grâce à la technique de suivi de l'œil. Une partie des résultats décrits dans ce chapitre étaient présentés au workshop IMPACT pendant la conférence HiPEAC en 2015.

Dans le chapitre 7, nous proposons d'intégrer les informations supplémentaires dans l'outil de visualisation des programmes en proposant une nouvelle technique d'interaction basée sur l'altération contrôlée du mouvement du curseur de la souris. Sur la base de cette extension, nous démontrons comment les modèles des programmes peuvent être adaptés aux techniques d'interaction et vice versa. Cette technique sert aussi d'exemple pour soutenir l'intérêt de la recherche en interaction homme-machine pour développer la restructuration interactives des programmes.

Finalement, nous discutons de la conception des outils de restructuration interactive des programmes dans le chapitre 8. Nous proposons des implications pour les modèles d'interaction et surlignons la nécessite de développer un partenariat homme-machine dans la restructuration et l'optimisation des programmes.

La généralisation de la restructuration interactive des programmes consiste à appliquer les approches de conception des logiciels centrées sur l'humain afin de créer des outils pour soutenir le cycle entier de la vie des programmes. Plus particulièrement, nous proposons de concevoir des outils d'aide à la programmation qui sont adaptables et modifiables par leurs utilisateurs. La conception de ces outils devrait se baser sur les études des utilisateurs plutôt que sur les détails de fonctionnement d'un système. Étant donné la puissance de calcul actuellement atteinte, les outils de restructuration permettront augmenter à la fois la performance du programmeur et celle du programme grâce au passage simplifié entre les différents niveaux d'abstraction. Ils donneront aussi plus de contrôle sur l'exécution de programme sans pour autant exiger de tout contrôler en détail grâce au partenariat homme-machine, où le développeur guidera l'ordinateur vers la solution correcte et l'ordinateur guidera le développeur vers la solution la plus efficace. Nous proposons de prendre en considération non seulement les langages des programmation, mais tout l'écosystème des outils de programmation interconnectés en combinant les techniques d'analyse et de manipulation pour soutenir les développeurs.

CONTENTS

1 INTRODUCTION 1
1.1 Bridging the Programming Abstraction Gap 1
1.2 Thesis Overview 5
2 RELATED WORK 9
2.1 Visual Languages and Software Visualization 9
2.1.1 Early Years of Visual Representations of Software 9
2.1.2 Defining Software Visualization 11
2.2 Taxonomies of Software Visualization 13
2.2.1 General Taxonomies 13
2.2.2 Taxonomies for specific types of software visualization 16
2.2.3 Classifications of information aspects 17
2.3 Information-centric View on Software Visualization . . 18
2.3.1 Code Lines 20
2.3.2 Slicing and Dicing 21
2.3.3 Iteration and Repetition 23
2.3.4 Object-Oriented Hierarchies 24
2.3.5 Memory Access and Management 26
2.3.6 Data Structures 28
2.3.7 Control Structures 29
2.3.8 Execution Traces 30
2.3.9 Multithreading Synchronization 31
2.3.10 Communication 32
2.3.11 Numeric Metrics 34
2.3.12 Software as Data Source 35
2.4 Visual Elements of Software Visualization 36
2.4.1 Visualization Primitives 36
2.4.2 Program Visualization Techniques 37
2.5 InfInt Design Space 39
2.5.1 Degree of Interactivity 39
2.5.2 Mapping Techniques into the Design Space . . 40
2.5.3 Implications for Design 40
2.6 Discussion 42
2.7 Conclusion 44
3 POLYHEDRAL PROGRAM REPRESENTATION 45
3.1 Program Representations 45
3.2 The Polyhedral Model 47
3.2.1 Development of the model 48
3.2.2 Limitations 49
3.2.3 Workflow in the Polyhedral Model 50
3.2.4 Forms of Representation 51
3.3 Representing Programs in the Polyhedral Model 52

3.3.1 Representing Statement Instances 53
3.3.2 Representing Execution Orders 55
3.3.3 Representing Memory Accesses 59
3.4 Dependence Analysis and Parallelism 59
3.5 Structure of the Polyhedral Scheduling Relation 62
3.5.1 Existing Scheduling Structures 62
3.5.2 Scheduling Structure Selection 64
3.5.3 Dimension Semantics 64
3.5.4 Scheduling Relation Equality and Equivalence . 66
3.5.5 Scheduling Normalization 67
3.5.6 Exposing Lexical Order in Scheduling Relations 68
3.5.7 Lexical Order Normalization 70
3.5.8 Scheduling Validity Conditions 71
3.6 Discussion 73
3.7 Conclusion 76
4 VISUALIZING POLYHEDRAL PROGRAMS 77
4.1 Visualization Techniques for Polyhedral Programs . . . 77
4.2 Participatory Design Workshop 78
4.2.1 Protocol 78
4.2.2 Results and Discussion 79
4.3 The Clint Visualization 82
4.3.1 Statement Instance Scatter Plot 82
4.3.2 Memory Accesses as Interactive Color Coding . 86
4.3.3 Memory Accesses as Nested Parallel Coordinates 88
4.3.4 Clint Interface 90
4.4 Architecture and Implementation 91
4.4.1 Statement Instances 91
4.4.2 Memory Accesses 93
4.5 Evaluation of Clint Visualization 95
4.5.1 Protocol 95
4.5.2 Results 96
4.6 Discussion 99
4.7 Conclusion 101
5 HIGH-LEVEL PROGRAM MANIPULATION 103
5.1 Semi-automatic Program Restructuring 103
5.2 From Transformation Directives to Polyhedra 104
5.2.1 Loop Transformations Expressed in Polyhedral Model 104
5.2.2 Revisiting Classical Transformations in Clay . . 107
5.2.3 Transforming between Arbitrary Scheduling Relations in Clay 112
5.2.4 Clay Directives in Practice 114
5.2.5 Discussion of the Transformation Set 116
5.3 From Polyhedra to Transformation Directives 118
5.3.1 Combining Manual and Automatic Program Transformation 118

5.3.2 Detecting Complementary Transformations . . 119
5.3.3 Aligning Relations and Matching Beta-vectors . 120
5.3.4 Generating Transformation Sequence 121
5.3.5 Discussion of the Algorithm 124
5.4 Directive Recovery in Practice 127
5.4.1 Recovering beta-vectors 127
5.4.2 Benchmarks 128
5.4.3 Example 129
5.5 Interacting with a Polyhedral Compiler 132
5.6 Conclusion 135
6 POLYHEDRAL PROGRAM RESTRUCTURING 137
6.1 Augmenting Directive-based Restructuring with Direct Manipulation 137
6.2 Direct Manipulation of Statements and Instances . . . 138
6.2.1 Mapping Transformation Directives to Graphical Actions 138
6.2.2 Discussion of the Mapping 141
6.3 Transformation Replay and Correction 143
6.3.1 Undoing and Replaying Transformations 143
6.3.2 Interacting with a Polyhedral Compiler Graphically 143
6.3.3 Correcting Automatically Computed Optimization 144
6.4 Evaluation of Direct Manipulation Benefits 149
6.4.1 Expected Benefits 149
6.4.2 Experimental Protocol 149
6.4.3 Results and Discussion 151
6.5 The Need for Code 154
6.5.1 Protocol 154
6.5.2 Duration and Correctness 158
6.5.3 Representation Choice 163
6.5.4 Visual Behavior 165
6.5.5 Discussion 169
6.6 Interactive Program Restructuring in the Polyhedral Model 172
6.7 Conclusion 173
7 CONSTRAINED MANIPULATION 175
7.1 Constrained Program Manipulation and Background Information 175
7.2 Pointing Transfer Functions and Pseudo-haptic Feedback 177
7.3 Morphology of the transfer function 178
7.3.1 Transfer function arguments 178
7.3.2 Transfer function shape structure 181
7.4 A design space for C-D gain change-based interfaces . 182
7.4.1 Design Space Dimensions 182
7.4.2 Using the Design Space: Combining Dimensions 183
7.4.3 Design Space Limitations 186

7.5 Communicating information through C-D gain 186
7.5.1 Transfer function shape for communicating information 187
7.5.2 Evaluating information communication 188
7.6 Using C-D gain changes alone 189
7.6.1 Experimental Protocol 189
7.6.2 Data Collection 192
7.6.3 Apparatus and Implementation 192
7.6.4 Metrics and data post-processing 193
7.6.5 Ordering effects 194
7.6.6 Effects of transfer function shape 195
7.6.7 Effects of transfer function discretization 196
7.6.8 Exploratory analysis: elements of strategy . . . 197
7.6.9 Qualitative elements of strategy 199
7.6.10 C-D gain change interpretation 200
7.6.11 Discussion 201
7.7 C-D gain change with background visualization 201
7.7.1 Experimental Protocol 202
7.7.2 Effects of visual feedback 203
7.7.3 Comparison with previous experiment 205
7.7.4 Discussion 205
7.8 Communicating Program-Related Information 206
7.8.1 Predicting Transformation 206
7.8.2 Representing Dependence Violation 207
7.8.3 Representing Access Locality 208
7.8.4 Selecting Feedback Source 211
7.9 Guiding Manipulation around Constraints 211
8 CONCLUSION AND PERSPECTIVES 215
8.1 Designing Tools For Interactive Program Restructuring 215
8.2 Contributions 218
8.3 Interacting with Inaccessible Object of Interest 220
8.4 Future Work and Perspectives 223
A EXTENDED INFINT DESIGN SPACE 227
B STATISTICAL METHODS 231
B.1 Key probability distributions 231
B.2 Confidence Intervals 232
B.3 Effect sizes and hypothesis testing 233

BIBLIOGRAPHY 235

CONVENTIONS

Hyperlinks to this document are highlighted in blue as in 1.

Citation hyperlinks are highlighted in green as in [275].

External URLs are highlighted in red as in https://ozinenko.com/.

Margin notes are used throughout this document polymorphically. Given the mix of disciplines and methodologies, margin notes often explain domain terms and abstractions. Notes with ↩ symbol serve as reference anchors for definitions of terms back-referenced in further text. Notes with ▷ symbol highlight important observations, results or conclusions. Nevertheless, footnotes [1] are still used for minor citations and URLs that fit poorly into margins.

this is a margin note

↩ key for margin symbols

▷ Margin notes are more contextual than footnotes.

1. The electronic version of this document will be available at https://ozinenko.com/phd.pdf

1 INTRODUCTION

1.1 BRIDGING THE PROGRAMMING ABSTRACTION GAP

Computer programming is a complex heavily-specialized activity that requires a lot of cognitive effort. During their history, computers went from specialized highly diverse systems to homogenized general purpose systems, and back to the multitude of widely disparate systems with the advent of ubiquitous computing [261]. The ways to control the computer operation evolved alongside computer architectures resulting in a stack of abstractions that is used to manage the constantly increasing computer systems complexity. Yet these abstractions came at a cost of reduced computer efficiency: programmers can no longer control all the numerous and diverse low-level aspects of the computer operation and rely on generic, often suboptimal abstraction lowering tools.

First computers were specialized, and reusable programs non-existent, letting engineers adapt every computer to the task up to the circuitry level reaching peak efficiency. Designs were so radically different that instruction and data circuitry could be fully separated (Harvard architecture) or combined (von Neumann architecture). ENIAC, often cited as the first general-purpose computer, was essentially a collection of physically implemented arithmetical functions [105] and was programmable by switches and rewiring (Figure 1).

In the early years, computer programming was separated from operating the computer, although tightly linked to its architecture. Programs were created on paper and then reproduced on the metallic or paper tape using special coding systems. They were specified using the machine codes directly or using the assembly language, which introduced textual mnemonics for machine commands. Beside being error-prone, machine codes had to be recreated for each computer with different internal organization and available instructions. Program representations reflected computer operation rather than the details of the task to solve.

High-level programming languages were designed to shift the focus from low-level computer operation to computational form of the task, making programming more abstracted from the target machine. *Plankalkül* for Konrad Zuse's Z3 computer is often considered the first theoretical proposal of the high-level language [143] [1] that used notation closed to relational formalism in mathematics. However, it

1. [143] gives an overview of all listed early programming languages.

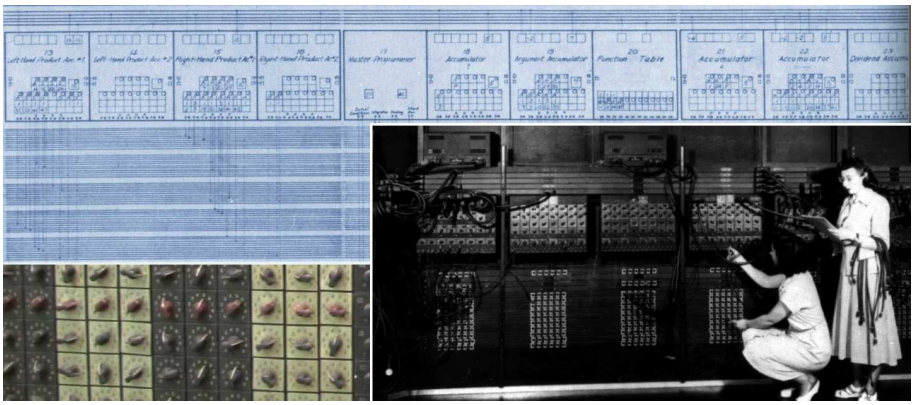


Figure 1 – First general-purpose computer, ENIAC, was programmed by switches and rewiring (image credit: US Army research laboratory http://ftp.arl.mil/ftp/historic-computers/).

was not implemented until 1975 [121]. *Short Code* is another early high-level language that allowed to translate mathematical expressions into a structured numerical form. Programs in *Short Code* were not translated into machine codes, but interpreted by a special program. *Mark I Autocode* is arguably the first *compilable* programming language: it allowed to represent mathematical expressions and used a separate program — an autocoder — to transform autocode into machine codes [143]. Autocode is a generic term that was used to designate first programming languages, and an autocoder is essentially a compiler. Their main innovation was the possibility to create the program for a computer using the same computer, hence the name.

Fortran is arguably the first high-level platform-independent programming language. Similarly to its predecessors, it was based around mathematical expressions. Lisp is another trailblazing high-level language that introduced a different, functional paradigm, closer to mathematical abstraction than to computer operation.

New programming languages kept appearing to address the increasing complexity of the tasks computers were required to solve. Major innovations in programming languages provided increasingly higher *levels of abstraction*, allowing to express more intricate domain areas with less effort. Structured programming in Algol and object-oriented programming in *Simula*, then *Smalltalk* were proposed to simplify programming and make it less error-prone [29]. With the development of graphical interfaces, visual programming languages started to appear reflecting the early graphical representations of the programs, such as flowcharts and graphs [62].

At the same time, computer architectures kept evolving to provide more computing power. Generations of computer architectures are divided by their element base: vacuum tubes were replaced by transistors, which were later packed into integrated circuits with ever-

increasing density. In 1965, Moore made an empirical observation that the number of transistors in the integrated circuit doubles every two years [189]. Nine years later, Dennard observed that, while the transistor size decreases, their power density remains constant, i. e. the power consumption is proportional to the area of the integrated circuit rather than to the number of components [79], resulting in the exponential growth of performance per watt of consumed energy. Dennard scaling essentially allowed to improve performance by replacing the computer hardware by a newer one with higher clock frequency without modifying the software.

While Moore's observation is still valid today [168, 172], exponential performance scaling resulting from component base frequency increase stopped due to huge thermal energy losses [141]. Instead, processor designers turned to parallel architectures with deep memory hierarchies to maintain performance scaling [239]. Although parallel architectures feature ever-increasing performance characteristics, they do not allow to make software run faster immediately. Instead, they shift the burden of performance improvements onto software developers who now have to create tools and techniques that efficiently exploit all features of parallel architectures [159].

Chip designers are under so much pressure to deliver ever-faster CPUs that they'll risk changing the meaning of your program, and possibly break it, in order to make it run faster.
H.Sutter

The independence in development of programming languages and computer architectures, granted by constant and almost effortless performance increase, resulted in radically different abstractions being used in languages and hardware. While programming languages let programmers express problem domains in more natural terms, they become increasingly hard to translate to machine languages and the result of this translation is barely understandable for programmers (Figure 2). Worse, compilers are often opaque and provide little to no control over the translation process. At the same time, programming languages cannot fully ignore higher-level program properties, in particular they must account for the possibility of parallel execution [35].

This abstraction gap creates an additional cost for program and programmer performance, but also an opportunity for improvement: when moving down in the abstraction stack, the additional details may be filled in automatically so as to exploit the available resources better. However, the abstraction stack is so deep and individual abstractions so diverse that even automated tools cannot yield best solutions and rely on imprecise *heuristics* [265]. Fully automatic parallelization has been a holy grail of program optimization for years, and is only partially possible [76].

Therefore, to be more efficient in exploiting the power of modern computers, developers need high-level program manipulation tools that give them control over lower-level processes without requiring to manage everything. This can be achieved by letting an automatic system, e. g. a compiler, to handle boilerplate program parts and

Figure 2 – Modern computers are programmed by bulk of text. Heavily optimized and highly abstracted programs are difficult to maintain and restructure. Left: a part of the code of a popular JavaScript library. Right: GCC error message from a single template error.

interact with the developer when a piece of information crucial for optimization is missing. In order to take informed decisions about program manipulation, developers need *more information than already available in the programming code*, or need to have the already available information *better accessible and easier to manipulate.*

The need for representing and manipulating information shifts the design of program restructuring into the area of Human-Computer Interaction. While first computer operators were necessarily programmers, major advances in HCI allowed people to simply *use* computers without *programming* them. For example, the *direct manipulation* interaction paradigm, ubiquitous in modern computer systems, was initially proposed as *a step beyond programming languages* [227].

While programming languages partially address the complexity of initial program development, they are rarely designed to simplify further steps of software life cycle. Even if new languages or language extensions are designed for efficient use of modern computer systems [222], they fall short at supporting *analysis* and optimizing *restructuring* of already existing programs. On the other hand, software visualization tools support program *analysis* [80], but rarely provide means to modify the program using the same interface. By proposing *interactive program restructuring*, we aim at designing efficient program manipulation tools that help programmers manage the complexity of the software on one hand, and provide insights for human-computer interaction research thanks to studying developers as *extreme users* on another hand. Design of interactive program restructuring tools requires addressing both the program optimization modeling and the interactive program visualizations.

1.2 THESIS OVERVIEW

This document supports the following thesis.

> Modern parallel programming languages and libraries help *create* new efficient programs, software visualization tools allow to *analyze* program efficiency, yet neither of them is consistently designed to support *restructuring* existing programs to improve efficiency. *Interactive program restructuring* combines effective software visualization techniques with high-level program manipulation tools in a single interactive interface, reducing both the program analysis and the program manipulation time. Efficient restructuring tools should present program data that is either not available or not immediately accessible in other representations, including source code, and allow for manipulating this data so as to change the program. Designing such tools requires not only the relevant visualization and interaction techniques, but also the high-level program models that support deep restructuring.

↩ *thesis statement*

This manuscript illustrates *interactive program restructuring* by proposing guidelines for program restructuring tools and using them to design and evaluate an interactive program parallelization and optimization system based on the state-of-the-art *polyhedral model* for program optimization. It is structured as follows.

Chapter 2 reviews existing work in software representation and visualization. It provides a *design space* structuring the software visualization tools and techniques with respect to program-related information they represent and interaction they support. Building on the morphological analysis of software visualization, it establishes basic principles of interactive program restructuring.

Chapter 3 presents the *polyhedral model*, a state-of-the-art abstract program representation providing precise analysis and high-level program manipulation capabilities. This chapter defines a *structure of the polyhedral representation* suitable for interactive program manipulation.

Chapter 4 describes the user-centered design of a software visualization technique based on the polyhedral model. It demonstrates how this visualization provides information that was hard to extract from code in a comprehensible yet concise manner. The technique is evaluated experimentally with a group of programmers having different expertises. The results presented in this chapter were published at the VL/HCC conference in 2014 [275].

Chapter 5 proposes a bidirectional mapping between high-level program transformation directives and the polyhedral representation. It contributes a set of directives that allow to perform an arbitrary program modification in the polyhedral model as well as an algorithm that allows to express an automatically computed program optimiza-

tion in terns of these directives. Combined, these contributions open a way for interacting with a polyhedral-enabled compiler. The results presented in this chapter were published at the CGO conference in 2016 [12].

Chapter 6 builds on the previous chapter to enable direct manipulation of the polyhedral program visualization. It maps user actions to transformation directives and transformation directives to animated transitions. The entire system is evaluated experimentally to demonstrate its utility. The chapter also addresses the use of textual and graphical program representations and programmers preference through an eye-tracking study. Partial results of this chapter were presented at the IMPACT workshop of the HiPEAC conference in 2015 [274].

Chapter 7 integrates additional information to the program visualization tool by proposing a novel interaction technique based on modifying mouse cursor motion. It demonstrates how program models can be matched to interaction techniques and exemplifies the interest of human-computer interaction research in interactive program restructuring.

Chapter 8 discusses the design of interactive program restructuring tools in general. It presents the implications for interaction paradigms and outlines the need for developer-compiler partnership in program restructuring and optimization.

Contributions

This thesis is an interdisciplinary work between human-centered computing and optimizing compilation. It resulted in following technical, empirical and theoretical contributions to the polyhedral program optimization and the design of interactive program visualization and manipulation tools.

INTERACTIVE PROGRAM RESTRUCTURING TOOLS

— *Clay* provides a set of high-level program transformation directives that allows to arbitrarily transform the programs that are amenable to the polyhedral model and satisfy *validity* criteria;
— *Chlore* allows to uncover the sequence of *Clay* transformations that corresponds to an automatically computed program optimization even though the polyhedral optimizer does not operate in terms of directives;
— *Clint* provides an interactive visual program manipulation interface that is likely to improve both program analysis and transformation by the developer.

EMPIRICAL OBSERVATIONS

— given interactive visual representation of the loop iteration-level representations, software developers are likely to extract and express loop-level parallelism faster and more successfully;
— designing visual interfaces for program restructuring so that the pertinent data discovery is faster and program manipulation easier, together with sufficient training, may improve the acceptance of visual program representations;
— modifying the speed of cursor motion can be perceived and quantified by the user and thus can be used to communicate supplementary, non-critical information in visually loaded program manipulation interfaces.

THEORETICAL PERSPECTIVES

— *InfInt* design space classifies software visualization techniques with respect to program-related information they present and the level of interaction they provide, giving program manipulation tool designers a framework for *interactive program restructuring*;
— Structure of the polyhedral program scheduling combined with *global validity* conditions ensures safe program transformation in the polyhedral model and allows to ensure *completeness* of the *Clay* transformation set.

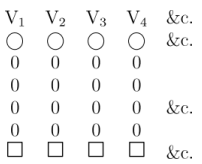
2 RELATED WORK

2.1 VISUAL LANGUAGES AND SOFTWARE VISUALIZATION

Software development and maintenance is an extremely complex area of activity that requires a multitude of skills and substantial mental effort. From the early days of computer programming, people used a variety of techniques to support creation, modification and use of software. Similarly to many engineering areas, a lot of these techniques leverage visual representations to provide insight into software, improve understanding and reduce the cognitive load of software developers. These representations allow, for example, to describe the target hardware more intuitively or to characterize the application domain in a more concise yet expressive manner.

Conventionally, a computer program is viewed as a sequence of instructions specifying its behavior and forming the programming code. Although modern digital computers operate with numbers encoded by electrical signals, instructions are most commonly represented as structured text better understandable for humans thanks to a higher level of abstraction that decreases the complexity of understanding software. Visualization and visual programming are often opposed to conventional "textual" programming [263]. However, evidence exists that experience in programming modern computers with code is closer to visual processing than to text due to the structure of the code and support tools such as syntax highlighting and code navigation [70, 202].

2.1.1 *Early Years of Visual Representations of Software*



Visualization of the Numbers on a Disc of Babbage's Analytical Machine.

From the very genesis of computer programming, people resorted to visual representations. In the Note B to the "Sketch of The Analytical Engine Invented by Charles Babbage" [1], Ada Lovelace illustrates the disc of the machine by a circle and a column of numbers. This structure is later reused in the famous image representing the first computer program (Note G of the same document). The representation of numbers and circle is not mandatory for the program itself, but serves as a means to explain how a program would operate on the machine: numbers are changed in the subsequent images to re-

1. Original text in French by Luigi Menabrea, Bibliothèque Universelle de Genève, No. 82, Geneva, 1842.; English translation and commentary by Ada Lovelace, in Scientific Memoirs, Vol. 3, pages 666-731, editor Richard Taylor, London, 1843. Available online at https://www.fourmilab.ch/babbage/sketch.html.

flect the data input and the expected output. One can thus argue that the first computer program included a visualization of the (partial) program execution.

Programming first electronic computers required thorough preparation. In order to represent conditions in computer instructions, pioneers of computing reused flowcharts, a well-known engineering tool for representing processes. First flowcharts with computer programs appear, among others, in works of von Neumann [2]. Once computers become powerful enough to be used as a support tool for programming, first software design and visualization techniques based on flowcharts started to appear [118].

With the advent of higher-level programming languages like FORTRAN and LISP featuring better expressivity and conciseness of programs, visual representations were adapted to representing the programming *code*. While flowcharts were still used to describe algorithms, it became possible to use a more concise textual representation and thus create larger programs. Algorithm description is not necessarily made in an actual programming language, but may be written in *pseudocode*. New programming paradigms that emerged around high-level languages required new forms of visualization. For example, structured programming allowed for a neat visualization of the nesting structure for code blocks, combinable with flowcharts [194].

▷*Early program representations corresponded to the internal operation of computers.*

On the other hand, the use of visual form to *create* a program rather than to explain it emerged as a separate paradigm giving birth to the visual programming. SketchPad [238] with its innovative graphic interface is sometimes considered as a precursor for visual programming thanks to its support for user-specifiable constraints and graphic element reuse operations [40]. Early visual programming languages include, for example, graphical variants of the LISP tailored to visual specification of graph and list algorithms called AMBIT/G [62] and AMBIT/L [63] respectively. In another early visual programming system, PYGMALION [229], the user would create and alter the program by connecting graphical icons representing actions and properties. PYGMALION also defined the notion of *programming by example* that is still actively used in end-user programming, often in a form of automation or macro engines. Computer-mediated education remains the large field of application for visual programming, starting from the seminal "Mindstorms" book by Pappert [198] that presented the Logo language and its famous Turtle Graphics. Modern visual programming languages, such as Scratch [216] or Looking Glass [110, 111], are also targeted at computing education for non-programmers.

Visual tools for programming support emerged fast and were often multipurpose, resulting in a confusion between all pertinent aspects.

2. H.H. Goldstine and J. von Neumann. «Planning and Coding of Problems for an Electronic Computing Instrument», Part 2, Vol. II. Institute for Advanced Study, Princeton, NJ, 1948. 66 pages.

The terms *visual programming* and *program visualization* were often used interchangeably [106] (as cited by [190]). Both terms were also used to describe programmatic creation of graphical content (charts, animations) independently of the program itself being specified in code or using graphics.

▷Terms visual programming and program visualization were often used interchangeably.

Early systems for visual programming tightly coupled the *program specification* and its *output*, i. e. visual programming was primarily intended for creating visual content such as images or graphical interfaces. On the upside, it allowed to developers to directly operate the visual elements in the program. On the downside, it contributed to the confusion between visual programming and programming visual interfaces. Even today, graphical user interface design tools, such as Interface Builder [85], are used for graphically defining graphical interfaces in the otherwise textually-specified program.

The proliferation of terms related to graphics and programming urged the software visualization community to propose precise definitions to distinguish between program visualization and visual programming as well as to separate adjacent technologies.

2.1.2 *Defining Software Visualization*

The first work to provide a definition of *program visualization* was exactly motivated by the terminology confusion [190]. It separates programming-by-example as an orthogonal concept to visualization and draws a line between visual programming and program visualization. It gives the following definition for the former.

> "Visual Programming *refers to any system that allows the user to specify a program in a two (or more) dimensional fashion.*"

And another definition for the latter.

> "*In* Program Visualization, *the program is specified in the conventional, textual manner, and the graphics is used to illustrate some aspect of the program or its run-time execution.*"

These definitions imply that textual specification of programs is inherently one-dimensional, according to the way the code is written and read. Note that *textual* specification by this time is considered *conventional*. Although first program specifications were rather numerical or based on mathematical symbols, defining programs using mathematical formulae is one-dimensional in the same way as text and therefore is not visual programming.

The next attempt at defining program visualization precisely includes elements of the classification [208].

> "Program visualization *is the visualization of actual program code or data structures in either static or dynamic form.*"

This definition by Price is later reused in the heavily cited book by Stasko *et. al* along with the book-title standing as another definition "Programming as *multimedia experience*" [236].

In a later work, Price defines the term *software visualization* to underline the difference from programming or program specification, the software is already existing. This new concept also includes means and motivation as well as interaction [209].

> "Software visualization *is the use of crafts of typography, graphic design, animation and cinematography with modern human-computer interaction and computer graphics technology to facilitate both the human understanding and effective use of computer software.*"

This definition is now widely accepted in the field. However, as Price notices himself in the paper, this definition may also cover a representation of the program written in the visual language. He further argues that visual languages are meant for simplifying *specification* of the software rather than its *understanding* and thus should not be considered as a form of software visualization. This argument may also apply to textual programming aid techniques such as syntax highlighting or code folding.

Further remarks on software visualization concentrate on clarifying its motivation and underlining the need for understanding complex systems. In their overview of 3D software visualization techniques, Teyseyre and Campo rephrase the earlier statement by Diehl [80] to justify the use of advanced visual effects [245].

> "*The aim of software visualization is not to create impressive images but images that evoke user mental images for better software comprehension.*"

Similar notions of the mental model and improved software comprehension is found in what the target audience expects from the software visualization. For example, this definition given by Mili from the software engineer's point of view [186].

> "Software visualization *is a representation of computer programs, associated documentation and data, that enhances, simplifies and clarifies the mental representation the software engineer has of the operation of a computer system.*"

Most of these definitions refer to visualization as representation of *software* and do not necessarily mention the code. Code is considered as another *representation* of the software and as such can be seamlessly integrated in most classifications.

Once the term *software visualization* was defined, researchers were able to separate tools and techniques specific to this field from others, or identify visualization aspects of larger systems for assisting software developers. These techniques required further morphological analysis and classification according to their visual form, presented software aspects and other characteristics.

2.2 TAXONOMIES OF SOFTWARE VISUALIZATION

Several taxonomies were proposed for classifying software visualization techniques and tools. These taxonomies either attempt to classify all tools and techniques [175, 209, 219] or focus on a particular aspect of visualization and data represented, such as three-dimensionality of representation [245] or representation of software evolution [147].

2.2.1 *General Taxonomies*

First taxonomies emerged shortly after the appearance of the first software visualization techniques in an effort to provide guidance in creating new visual representations. They were essentially proposed as design spaces for gathering insight about what was done and what else could possibly be done to represent increasingly complex programs graphically.

An early work by Myers [190, 191] is a result of growing interest in the non-textual representation of programs. He is the first to propose a distinction between *visual programming* and *program visualization* as cited above. In order to better illustrate the distinction, Myers proposes different taxonomies for the systems that support visual program specification and for the systems that visualize already existing programs. For program specification, he highlights visual programming as a category along with programming by example and interactivity of programming.[3] For program visualization, he suggests classifying the systems according to what they illustrate: *code* or *data*, and with respect to display: *static* or *dynamic*. At the time, dynamic visualizations were most often connected to the execution of the program and are meant to explain it, making a tight semantic link between the information being represented and the way it is represented. Static visualizations mostly covered algorithm and data structure visualizations. This classification starts to structure the large and disparate field and does not go into details about what aspects of code and data are represented and using which techniques.

Price *et. al* coin the term *software visualization* to avoid confusion with visual programming and better reflect the growing complexity of the field covering algorithm visualizations, program animations, program component relation diagrams, data structure displays, etc [208]. They propose six general categories subdivided into multiple characteristics specific to software visualization as follows.

▷*First taxonomies define the term "software visualization".*

3. In modern terms, interactive programming roughly corresponds to read-evaluate-print loop (REPL) in interpretable languages or to live coding, and batch programming corresponds to compilation and execution.

Scope regroups general characteristics of the software visualization system and its limitations including the class of programs, scalability and support for concurrency.

Content category comprises characteristics describing whether the system can represent code, data structure, algorithms along with it completeness with respect to the underlying execution model.

Form category allows to provide more detail on the visualization technique itself: use of visual attributes, animation, etc.

Method category covers the process of specifying the desired visualization and customizing it by the end user.

Interaction category shows whether the end user can interact with the visualization itself, focusing on navigation and filtering.

Effectiveness category tries to evaluate the system in general.

This arguably first taxonomy for software visualization reflects in its complexity the size of the emerging field. It makes an important point in distinguishing *what* is being visualized from *how* it is done.

In a later work [209], same authors characterize multiple software visualization systems available at the time and provide more detailed description of the categories, each of which is subdivided into multiple subcategories. For example, the *Content* category related to the program aspects being represented is subdivided into subcategories *Program*, *Algorithm*, *Fidelity and Completeness* and *Data Gathering Time*. The first two subcategories are further described in terms of control flow and data flow. This revisited taxonomy reflects the previous classification by Myers [191] in the code-data separation. Finally, it highlights the relation between program execution and dynamicity of visualization specifying, e. g., whether the animation in the visualization technique represents the time of execution or has a different purpose.

Stasko and Patterson reinforce the distinction between visualized information and visualization technique in their taxonomy designed for understanding software visualization systems [234]. They propose a set of the following characteristics.

Aspect characteristic highlights a specific feature of the program being visualized.

Animation characteristic captures the fact that an animation may not only serve to represent dynamic properties of the program, but to explain static aspects.

Abstractness of the visualization demonstrates how close is it to the programming code and the execution model compared to the semantics of the program, e. g. by representing time data on a clock rather than as plain numbers.

Automation degree represents the effort the programmer should make, in particular in modifying the program, in order to enable visualization of their software.

This classification build an important basis for designing visual program manipulation tools. It stresses out that *different tasks under various programming paradigms require different aspects to be visualized* in its *Aspect* dimension. For example, flowcharts naturally map to imperative programming languages but are much less useful for functional and declarative programming languages. In another characteristic, *Animation*, it captures the fact that, during the animation process, the program may *temporarily become invalid* as the animation is performed for the sake of explanation and understandability.

Roman and Cox take a different, more graphics-oriented approach to classifying software visualization systems [219, 220]. They propose the following four classification criteria.

Scope describes what aspect of the program is being visualized, focusing on distinction between code, data and execution traces.

Abstraction defines what kind of information is conveyed and what is abstracted away.

Technique specifies the particular visualization approach.

Specification Method characterizes how the visualization is constructed from the software and how it is parameterized.

From the interactivity point of view, the *Specification Method* criterion describes whether the user can interact with the visualization directly to update it rather than writing configuration files, re-executing the visualization software or modifying the program. This criterion essentially says whether the system is *interactive* and, more specifically, whether it supports direct manipulation [227, 228]. Roman and Cox also discuss the distinction between program visualization and algorithm animation arguing that program is characterized by its code and control flow rather than by an algorithm.

More recently, Maletic *et. al* revisit taxonomies of software visualization with task-oriented approach [175]. They propose to structure software visualization techniques according to:

Task the visualization intends to solve;

Audience it addresses;

Target that describes aspects of the program that have to be visualized for solving the given task;

Representation that specifies a particular visualization technique used to represent the *Target* information; and

Medium reflecting the growth of multimodal representations.

This work not only accounts for new developments in the field including systems leveraging large multi-display or immersive environments, but reorients the classification to a human-centric approach by describing the user and their intent while using the system. They also relate the *Representation* to general user needs for information visualization applications proposed earlier by Shneiderman [226] and adapt the Card *et. al* model for Visualization [46] to the software visualization field. By doing so, they focus attention on the software-related

data and the process of transforming it before visualization. It allows to separately classify interaction with the visual structures and with the data transformation.

▷*Later taxonomies included aspects of interaction with software visualization tools.*

At the early years, software visualization taxonomies and design spaces aimed at clearly defining what software visualization is and what it is not, drawing the borders between program visualization, visual programming, algorithm animation and illustrated design documentation. Later works put the user into the center of classification in order to better describe *interaction* with software visualizations.

2.2.2 *Taxonomies for specific types of software visualization*

Several taxonomies, design spaces or classification frameworks were proposed for specific problems addressed by software visualization, such as explaining algorithms through animation, modeling and analyzing the architecture of large software systems through real-world metaphors or understanding the changes of software over time.

ALGORITHM ANIMATION Important for exploratory and educational purposes, algorithm animation is a separate research domain with its own classifications and frameworks [44]. Most techniques are specific to animating classes of algorithms solving a particular problem, e. g. Sorting Out Sorting, a video and a later case study on using video for teaching algorithms [9, 230]. Shaffer *et. al* provide an overview of algorithm animation techniques structured by the class of algorithm represented [223, 224]. Nevertheless, several systems allow to construct ad-hoc algorithm animations, for example Tango [232].

SOFTWARE ARCHITECTURE As the scale of software development grew larger and faced new challenges, more and more new notions were visualized to assist developers in the emerging tasks. It led to the creation of *software architecture* visualization tools, e. g. software cities [262] and software landscapes [14], or even visual languages for creating and describing software architecture, e. g. UML [127]. Several authors proposed classification frameworks [102] or structured surveys [47] of these techniques paying attention to the effectiveness, target audience and visual metaphors.

SOFTWARE EVOLUTION On the other hand, developing and supporting software for the long term required a visual representation of *software evolution*. This domain covers a broad range of techniques from difference visualization [61, 86] to version control system graph representations [268], and from source code lines modification [59, 242] to evolution of software architecture [182]. One system may feature multiple levels of granularity for visualizing software evolution, for example the system by Eick *et. al* does so for both the source

code line-by-line changes and for relations between source files in project [87]. A framework was proposed to assess the effectiveness of software evolution visualization tools [258] as well as a systematic review crystallizing the research questions and the user requirements in the software evolution visualization [196].

SOFTWARE METRICS Finally, major advances were made in defining and visualizing *software metrics* [100]. To some extent, many software visualization techniques are in fact representations of previously extracted numeric metrics of the software. This view is consistent with Card *et. al* model for visualization that separates the data transformation into a separate step of the visualization design process [46]: the raw data (programming code, execution traces, version history) is transformed to a certain quantitative data by following the metric definition. Number of source code lines modified by each revision and their location in the file may be seen as a specific metric that may or may not be represented visually using as in, e.g., Code Flows [242]. Lanza and Ducasse provide a clear separation between software visualization techniques and software metrics in their work on understanding software evolution [156].

2.2.3 *Classifications of information aspects*

STATIC ASPECTS Following on the earlier distinction between static and dynamic visual representation and static or execution-related information [191, 219], Caserta and Zendra provide a detailed survey of the techniques for visualizing static aspects of software [49]. In addition to describing numerous software visualization techniques, they structure these techniques by the aspects of the software that are represented. At a large scale, they identify *code-line-centered* techniques, for example SeeSoft [86] or Sv3D [176]; *class-centered* techniques, for example Class Blueprint [84]; visualizations of *relations in the software* which include different types of call graphs and class diagrams, such as JavaVis [197] or Vampir [144], in addition to software architecture, evolution and metrics described above. This work sheds some light on the *kinds* of information present in software and is suitable for visual representation and analysis.

PARALLELISM Visualizing parallel programs was a primary concern for the software visualization field since its genesis due to the conflict between inherent sequentiality of the code and parallelism of execution. Kraemer and Stasko are arguably the first ones to provide a structured overview of tools and techniques specific to parallel software visualization [148]. They propose several general categories based on what information is presented and how. *Program Graphs* include various node-link diagrams mapped to the code or program

execution data, such as call graphs featuring order of execution (including parallel execution in, e.g., ParaGraph [119]), execution traces and event causality graphs. *Communication Graphs* display communication between processors and, eventually, the hardware topology of the underlying system. *Statistical Displays* provide aggregate descriptive statistics using common statistics visualization techniques such as bar charts, scatter plots or spider diagrams. *Memory Access Displays* are mostly focused around 2D spatial representation of memory as a table of cells in order to observe access patterns through animation. They also qualify certain visualization techniques as *application-specific* when they correspond to a particular execution model or *abstract* when they are configurable. This classification develops specific categories for the *source* and *kind* of information being represented visually: execution traces, system configuration, data access traces, etc.

Zhang *et. al* revisited the classification in their work on the role of graphics in parallel programming [272]. They paid particular attention to graph-based visualizations and the underlying models including dependence graphs, Petri nets and space-time diagrams. Besides the graph model *Formalism*, they propose to classify software visualization systems by *Granularity* and *Scalability* of the model itself rather than of the visualization approach.

2.3 INFORMATION-CENTRIC VIEW ON SOFTWARE VISUALIZATION

Many existing classification frameworks provide means for separating software (program) visualization from visual programming. The main separation criterion is whether the program is created and modified through the visual interface or is created in code and than represented visually [190]. Those software visualization classifications that involve interactivity imply *interactivity of the visual interface*, i. e. the possibility to interact with the visualization itself for exploration and analysis, but assume the software is not affected by this interaction. Visual languages, on the other hand, support *interaction with software* in a sense that interacting with the visual representation will change the software itself. Rather than binary separation, we propose to define a *continuum of interaction*: one can specify how the user interacts with the visual representation and how these interactions affect (or not) the software itself.

▷Existing classifications do not account for interaction with software through visual representations.

We revisit state-of-the-art related work on software visualization techniques considering the *continuum of interaction* with particular aspects of the program, describing for each technique:

↩ Classification criteria.

— what specific aspects of the software are represented;
— how are they mapped to the visual representation;
— what level of interaction is supported: with the visual representation or with the software itself for each of these aspects.

While some elements of the software visualization techniques are related to the properties of the software, others may be context or interaction-related or even have a purely aesthetic function. Identifying the set of visualized aspects of the program will allow to select a representation appropriate for a particular software development task. Specifying the mapping between program aspects and visual elements as well as evaluating the interactivity will help tool designers to consider program manipulation through the visual representation.

Analyzing a bulk of existing software visualization techniques described in the literature, we found that most of the early techniques do not focus on a particular software development task, but rather propose a tool for representing certain *information* about the software, letting the user select a set of the tools according to their goal. Although newer software visualization systems include multiple representations connected by a common need, we decided to structure our exploration by the *program aspects* that are visualized. We also observed that certain program aspects, such as class-based inheritance or function call order in a code block, are easier manipulable and may thus support more advanced interaction. We systematically extracted the aspects of the program that are visualized by different techniques and noted repetitive general characteristics, using a variation of Grounded Theory methodology [180]. These aspects were later arranged into the following categories and subcategories.

↩ Program-related information categories.

- program instructions and specification:
  - code line-based statistics;
  - program slicing and dicing;
  - iteration and repetition;
- program structure:
  - object-oriented hierarchies;
  - memory management;
  - data structures;
  - control structures;
- program execution:
  - execution traces;
  - multithreading or multiprocessing synchronization;
  - communication in distributed programming models or environments;
- software metrics.

Some of the software visualization tools and techniques represent multiple aspects of software and therefore fall into more than one of these categories. For example, mapping synchronization primitives' call times extracted from execution trace to per-thread temporal axes is both execution trace and synchronization visualization, but these aspects may be rendered independent: execution trace may be visualized for one thread while synchronization is depicted separately. They will be analyzed in detail in the most appropriate category ac-

cording to the motivation of the particular software visualization technique, for the same example of tracing synchronization primitives for debugging deadlocks, the target category is synchronization.

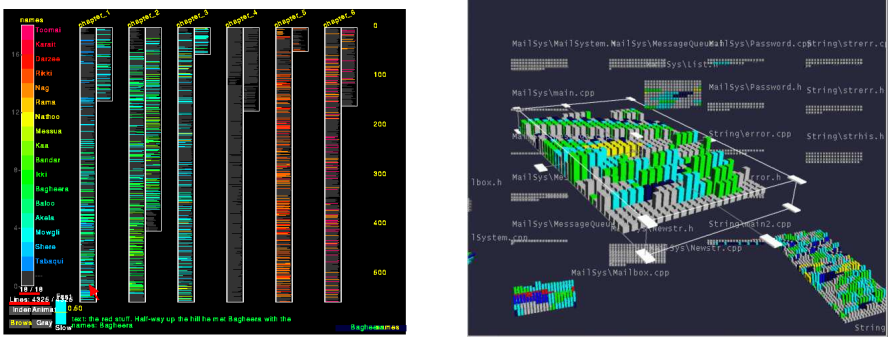
2.3.1 *Code Lines*

For the programs created in high-level languages, first visualization techniques were built around their code. Moreover, such features of the code editor as syntax highlighting or visual blocking may be considered as visualization elements since they are used to improve program comprehension rather than being part of the program itself. Further development of this idea allowed to propose visual abstractions for representing large quantities of code and their properties without demonstrating the code itself.

TOOLS AND TECHNIQUES

SeeSoft is one of the first tools proposed for visual analysis of large software projects with substantial code bases [86]. It maps categorical or numerical data to a color for each line of the source code thus keeping a tight connection between the visual and textual representation (Figure 3a). *SeeSoft* combines software evolution data, namely line modification times, author and VCS comment, with analysis of program *execution trace*. *SeeSoft* features basic interaction with the visualization. The user may zoom from overview representation to code and use brushing to select multiple parts of the program and highlight them in the code editor. These changes does not affect the software itself.

VCS — Version Control System, e. g. svn or git.



(a) SeeSoft, reproduced from [86] (b) Sv3D, reproduced from [176]

Figure 3 – Software visualization tools based on minimized code line representation in two or three dimensions.

Sv3D augments the *SeeSoft* technique by adding the third dimension and thus allowing for different numeric values to be mapped to color and height of the visual element [176]. *Sv3D* is demonstrated with mappings from surrounding *control structure* type and its nest-

ing level mapped to the shape and color of 3D graphical elements as shown on Figure 3b. Most interactions in *Sv3D* provide support for using 3D representation on a 2D display featuring pan and zoom, 3D rotation and translation. Contrary to its predecessor, *Sv3D* features element filtering by parameter through a separate interface. As in *SeeSoft*, none of the visually performed changes affect the software being visualized.

Tarantula uses a line-based technique to visualize testing results [132]. It maps test coverage information, i. e. the number of test which invoke a particular line, along with the information on the test run success to the line color and opacity. Tarantula provides basic menu-based interaction for filtering away passing or failing tests and selecting time intervals. It also provides precise numerical information on failures and coverage when hovering specific code lines. It does not support interaction with the program, even at the level of disabling certain tests.

OPPORTUNITIES FOR INTERACTION

Code-line based software visualization techniques provide overviews of large code bases and *characterize* the code rather than represent it in a manipulable form. When available, manipulation is often restricted to text editing without special support for program semantics. Unless the language imposes additional constraints on the code formatting, e. g. mandatory block indentation in Python, non-content changes in the code lines will not affect the behavior of the program, however they may improve readability or visual perception of the code by the developer. Stronger semantics related to code statements and blocks may be provided by coupling line-based abstraction with the model of the code such as a syntax tree, in which case the entities in this model may become manipulable through the code-line representation. Most code editors and IDEs perform some code modeling to support features like syntax highlighting or block folding.

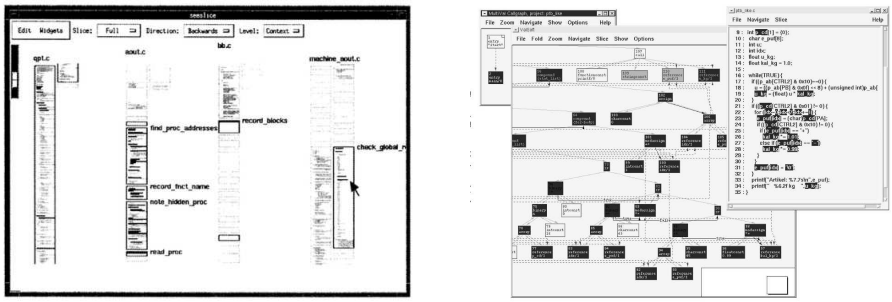
2.3.2 *Slicing and Dicing*

In program analysis and debugging, dividing the program or its components, e. g. functions or control flow blocks, into small and manageable parts is a recurrent task. Program slicing automates this division by including a slicing criterion for creating these small parts, or *slices*, for further analysis. Whenever the analysis requires user attention, visualization allows to simplify the exploration of program slices and relations between them.

TOOLS AND TECHNIQUES

SeeSlice is an evolution of *SeeSoft* for visualizing program slices [13]. *SeeSlice* uses a minimized *code line* representation with nodes sur-

rounding each slice, which can be a module, a function, a block or a statement as depicted on Figure 4a. Once a slice is selected by clicking or brushing, other slices that access the same data are highlighted. It also allows for slice filtering through menu-based interface and switching between overview and full scale code representation. However, it does not affect the slicing mechanism, nor does it allow editing the code.



(a) SeeSlice, reproduced from [13] (b) Krinke *et. al* visualization [149]

Figure 4 – Tools for visualizing program slices rely on code-line representations or node-link diagrams coordinated with code selection.

Program Slice Browser is a tool for visualizing program slices featuring node-link diagrams [78]. Similarly to *SeeSlice*, it starts with functions or code blocks as slices and allows for dividing them into smaller parts. Mainly intended for debugging, *Program Slice Browser* features grouping slices into supernodes. This simplifies the debugging by pruning correct slices and focusing on the reduced number of problematic slices and connections between them. *Program Slice Browser* contains an integrated code editor featuring coordinated selection with the graph representation. Despite this coordination, changes to the program code can only be made textually.

Krinke proposed a graphical tool for visually representing control and data dependences in C programs integrated with program slicing [149]. Its functionality is similar to both its predecessors: coordinated selection of program slices in visual and textual representation, highlighting of dependent slices (see Figure 4b). In addition to slices, it computes chops (sometimes also called dices) — program parts between two statements that convey the influence of one statement on another, e. g. assignments to the variables used in both statements. Considering the distance in the source code lines between two statements belonging to a chop, the tool visualizes regions of tightly coupled computations where multiple statements are influencing multiple other statements in a scatterplot-like style. Similarly, it identifies code regions accessing disjoint memory regions. This tool relies on general purpose graph alignment algorithms, such as GraphViz [90], and therefore offers no interaction with the visual representation.

OPPORTUNITIES FOR INTERACTION Program slicing visualizations are designed to simplify debugging by isolating problematic parts [260]. However, they lack tighter integration with the debugger and the slicing algorithm to enable program manipulation through the visual representation. Slicing algorithm can also be dynamically parameterized to control the slice granularity [146]. Dynamic program slicing can benefit from a visual interface to make criterion selection easier as well as to provide history navigation and undo mechanisms. The slicing technique can be augmented by fully manual slice creation and management, or by an example-based criterion specification.

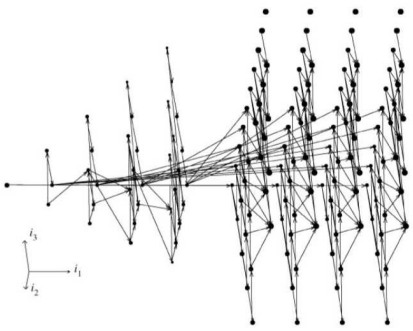
2.3.3 *Iteration and Repetition*

Contrary to the majority of software visualization techniques that tend to be more abstract than the code, several tools focus on individual loop iterations starving to provide *more* detail about the execution. These techniques are mainly used for detecting fine-grain parallelism.

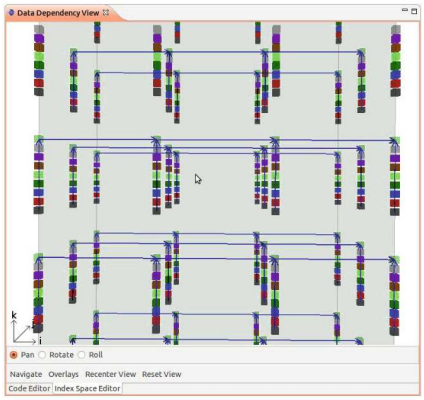
Fine-grain parallelism appears between individual statements or executions thereof, coarse-grain parallelism appears between larger software components.

TOOLS AND TECHNIQUES

3D Iteration Space Visualizer represents each iteration of a loop nest [270]. Instead of extracting an access pattern from the code, it analyzes memory accesses of every single iteration and builds a data dependence graph with links between iterations that access the same data. This graph is presented as a combination of a 3D scatterplot and a node-link diagram. By analyzing the dependence visualization, the user can discover loop-level parallelism present in the program, but has to modify the source code externally to express and exploit this parallelism. *3D Iteration Space Visualizer* only provides typical interaction techniques for manipulating 3D objects through pan, rotation and zoom, without affecting the code.



(a) 3D Iteration Space Visualizer, reproduced from [270]



(b) Tulipse, reproduced from [266]

Figure 5 – Loop iterations and loop-carried dependences are often visualized since they are not immediately visible in the code.

DECO is a tool that combines a memory access pattern visualization with the source code editor. The system is targeted at cache hit optimization [241]. It allows, for each statement within a loop nest, to visually examine the set of addresses accessed by a statement and its presence in multiple caches. Thanks to the selection coordination between the visual representation and the code, DECO enables the user to quickly switch between code editing and analysis. However, it does not allow data restructuring, e. g. by changing alignment, through the visualization itself.

Tulipse is an Eclipse IDE plugin that visually combines performance measurements with loop-level dependence analysis [266]. *Tulipse* includes two main visual representations: the call graph as a node-link diagram, enriched with a list of loop nests executed inside each procedure, and the loop iteration space with data dependences as a 3D scatter plot similar to *Iteration Space Visualizer*. Being an IDE plugin, Tulipse has strong support for navigating between code and visual representation in addition to typical pan and zoom iteraction for node-link diagrams. Nevertheless, the iteration space or the call graph are not modifiable in the visualization.

OPPORTUNITIES FOR INTERACTION

Iteration-level visualizations provide *more detail* than the code, namely allow to expose loop-level dependences and parallelism. To support interaction, iteration-level representation require a program model that abstracts per-iteration modifications to the program, significantly restructuring the code. Such models are often used inside automatic tools for program optimization [265] and are notoriously complex requiring the developers to rely on imprecise characteristics. At the same time, an expert user could analyze the visual representation and provide complementary input to the model and rely on the automatic tool to perform the required program modifications.

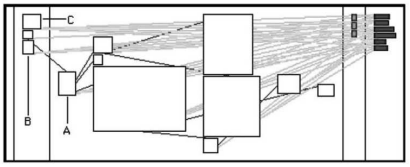
2.3.4 *Object-Oriented Hierarchies*

Object-oriented programming provides a higher-level abstraction for program representation, including objects and, typically [4], classes of objects. Objects may include other objects while object classes may inherit other object classes, objects also belong to a class. These aggregation and inheritance relations form graph structures of objects and classes that can be represented visually using general purpose graph visualization techniques or specific approaches for software visualization. UML is a common base representation for these techniques [152].

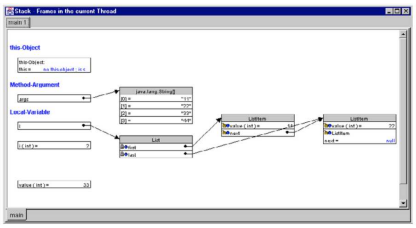
4. Most modern object-oriented languages use a class-based model, but alternatives exist, e. g. JavaScript or Self with prototype model

TOOLS AND TECHNIQUES

SHriMP Views [237, 267] is one of the first tools to automatically extract and visually represent class information from Java programs. It represents packages and classes as nodes and inclusion/inheritance as edges of a node-link diagram. *SHriMP Views* features expandable nodes, each of which may be transformed into an entire node-link diagram with edges mapped to a different property than the enclosing diagram. In order to build its visual representation, *SHriMP Views* collects Java reflection information about packages, classes, inheritance, interface implementation, and a list of typed class fields and methods. Although advanced reflection mechanisms provide means for modifying the class content, *SHriMP Views* allows interaction only with the visualization without modifying the source program.



(a) Class Blueprint, reproduced from [84]



(b) JAVAVIS, reproduced from [197]

Figure 6 – Visual tools for managing object-oriented programming languages focus on object and class hierarchies and class members.

JavaVis uses Java Debug Interface to create dynamically updating representations of objects during the execution of a Java program [197]. It simultaneously builds a node-link diagram representing the structure and the values of the objects, shown on Figure 6b, and a UML-style sequence diagram of method calls. Similarly to debuggers, it captures and stores function call frames allowing the user to analyze variables' values and lifetimes, even after the function returned. *JavaVis* provides animated transitions for object diagram updates. It also enables the user to navigate both diagrams visually and use search and filter dialogs. Nevertheless, it does not allow modifying object values through the interface as a traditional debugger does.

The Class Blueprint provides a detailed view of a class, its methods and attributes [84]. It separates the class into five layers based on their interaction with class data: initialization, interface, implementation, accessor and attribute as depicted on Figure 6a. Edges between nodes correspond to method calls or attribute accesses. Finally, node shapes may be mapped to numeric *metrics*. The Class Blueprint allows to identify which methods may affect an attribute and, after examining the code, how. The user can directly manipulate the visual representation reorganizing node layout and following edges, but cannot modify the program without moving to the code.

Beck *et. al* highlight that the graph representation of program entities includes multiple different types of edges due to various relationships between these entities [27]. They propose a visualization that combines node-link diagrams and parallel coordinate plots where these relations are represented in different parts of the visualization [1]. Their tool uses the following individual relationships: inheritance, aggregation, use, co-change (entities change in the same revision in the VCS) and source code overlapping. The tool supports a focus+context representation and a brushing selection, but does not allow to modify relationships between program entities.

OPPORTUNITIES FOR INTERACTION

Generally, these architecture-level visualizations may support high-level program restructuring. For example, changing a link in a visualized inheritance graph [243] would also change the inheritance in the associated program. Some UML-based systems allow to either generate classes from the graph representation [8] or to reconstruct this representation from the source code [68]. However, they offer very limited support for modifying the existing code seamlessly [152]. While most modern object-oriented langauges provide reflection mechanisms to examine the structure of the classes dynamically, frequently used as data source for the visualization, they rarely allow to arbitrarily modify it on-the-fly. Object and class-level restructuring would require integrating the visualization tool with a compiler or, at least, a program model in order to properly resolve, e. g. , inheritance modification issues such as access to no longer available parent fields and methods or unwanted shadowing.

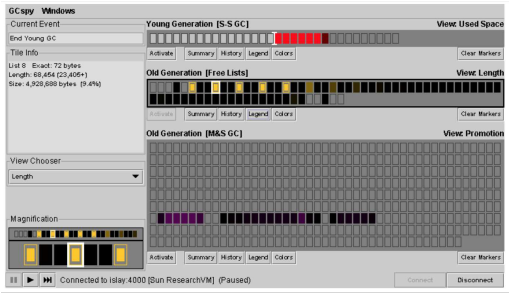
2.3.5 *Memory Access and Management*

Memory-related visualizations typically focus on two aspects: memory access patterns, used to build and visualize dependence graphs, and memory management strategies, including dynamic allocation and garbage collection.

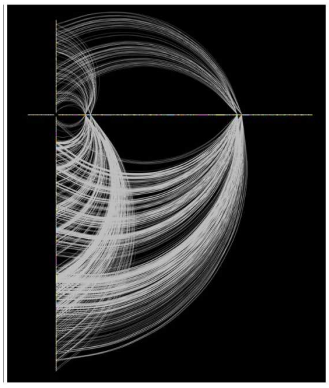
TOOLS AND TECHNIQUES

MAPA is a tool for memory access pattern visualization in FORTRAN [43]. It is tailored to FORTRAN memory representation for multidimensional arrays and allows to visually examine access to different array elements from loop-based program parts. Although *MAPA* rewrites array access code to enable dynamic updates of the visualization, it represents only the accessed elements without the associated program part thus preventing the user from modifying it directly.

GCSpy is a visual tool for monitoring the use of heap and the operation of garbage collection mechanisms in JVM-based languages [210]. Its visualization is built around memory regions that are accessed



(a) GCSpy, reproduced from [210]



(b) Memory allocation and death plot, reproduced from [257]

Figure 7 – Visual aid for memory management often focuses on memory allocation and automatic or manual deallocation.

over time using "hot-cold" metaphor (Figure 7a). Although it uses a customized JVM implementation to track memory, it does not provide control over the garbage collection or object relocation from the interface.

Memory allocation and death plot was designed to identify when a memory region is no longer used, before it is collected by the system [257]. It uses a version of the hive plot connecting allocation times with object "death" times allowing the user to track simultaneous creations and deletions as well as the lifetime of an object, as seen on the Figure 7b. The tool relies on execution tracing to collect information about objects' lifetimes. *Memory allocation and death plot* is presented as a static picture and does not allow any interaction.

OPPORTUNITIES FOR INTERACTION

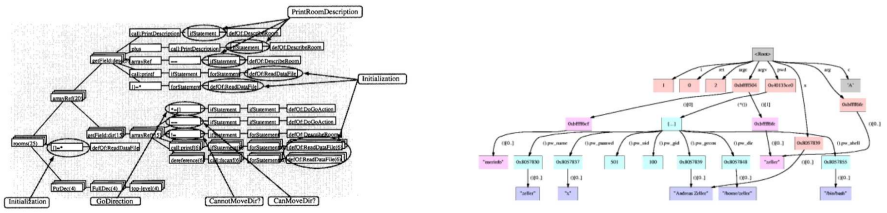
Programming systems with manual memory management may benefit from the tools that analyze memory deallocation and report potential problems, e. g. leaks or double deallocation. Systems with automatic memory management rely on garbage collection for memory deallocation [134]. Unless the garbage collection mechanism provides specific interface for controlling its behavior, the only pertinent interaction is forcing the collection process. However, most memory-related visualizations represented the memory itself (addresses, values) rather than the program statements that manipulate it. Without these statements, it is challenging to deduce the program modification from the changes to the visual representation.

2.3.6 *Data Structures*

In addition to a more abstract representation of the data structures and related algorithms, such as self-balancing binary trees or heaps, the mapping of the data structures to the programming model is important for program understanding. For example, C-like imperative programming languages are based on the concept of pointer that is internally used to represent almost any kind of connection while functional programming languages of the Lisp family are built around lists represented as nested pairs of heterogeneous objects.

TOOLS AND TECHNIQUES

Zimmermann *et. al* propose to capture the program state in a memory graph, in which data entities are connected by operations like dereferencing (pointer access), indexing or member access, and to use graph visualization techniques to display it [273]. As such memory graphs are large for real-world programs, their tool allows to explore them by node folding and filtering along with conventional pan and zoom interaction. Even though it piggybacks on the GNU debugger that allows to change the data on the fly, the tool does not translate data modification functionality to the interface.



(a) Star diagram, reproduced from [42]

(b) Memory graphs, reproduced from [273]

Figure 8 – Visual representations of data structures may allow modifying the data, but rarely the structure.

The *star diagram* interactive visualization technique is a rare case of a technique designed for *restructuring* the original program through the visual interface [42]. An example of the *star diagram* is presented on the Figure 8a. It represents data structures and successive accesses to them for a Lisp dialect Scheme. In addition to exploring the star diagram, which is visualized as a node-link diagram, through conventional node folding, filtering, pan and zoom, this interactive visualization supports selection of repetitive data accesses and their extraction in a function as well as the inverse, inlining, operation. It also allows to manipulate the number of function parameters, the function names and introduce function calls to the existing code.

OPPORTUNITIES FOR INTERACTION

Data structures are usually represented as separate entities in the pro-

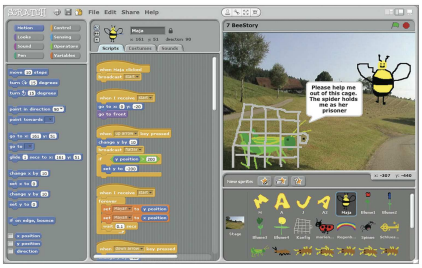
grams, making them easier to manipulate independelty from the rest of the program. However, it often requires changing not only the structure itself, but all the parts of the program that access it, which boils down to numerous modifications at the syntactic level. The contents of the data structure, on the other hand, can be modified relatively easily thanks to debugging and memory access mechanisms. Automatic tools were proposed to optimize memory layout [64]. As they are based on modeling or heuristics, they may benefit from interaction with the expert user.

2.3.7 *Control Structures*

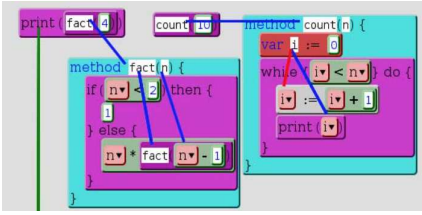
As the code representation is mostly centered around control structures, software visualization tools often avoid representing control structures to prevent information duplication. However, code line-based tools such as *SeeSoft* [86] and Sv3D [176] do represent the surrounding control structure since their visualizations are very close to the code. On the other hand, visual programming systems often let the user manipulate control constructs.

TOOLS AND TECHNIQUES

Many visual programming systems, such as *Scratch* [216] and *Looking Glass* [110], are built around visual representations for manipulating the control flow, for example see Figure 9a.



(a) Scratch visual programming language, reproduced from [216]



(b) Tiled Grace system, reproduced from [122]

Figure 9 – Control structures are mostly visualized as a part of visual programming language where they are turned into visual building blocks of the program.

Tiled Grace is a system that combines textual and visual programming [122]. As it can take a textual representation of a program an turn it into a visual one, it may be considered, to some extent, a software visualization technique. On the other hand, it provides full-fledged visual editing capabilities for arbitrary modification of the program. In addition to control flow and expression blocks, *Tiled Grace* allows for visually creating functions and calling them by creating a node-link diagram as shown on the Figure 9b. Although it

gives full control over the program code, *Tiled Grace* is not meant for *restructuring* programs via instruments like function extraction, but rather for program development and understanding.

OPPORTUNITIES FOR INTERACTION

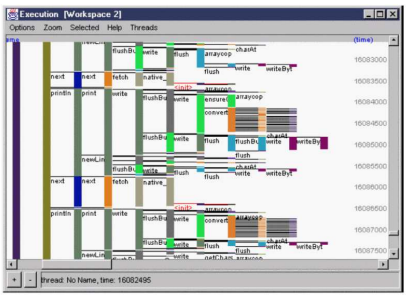
Numerous visual programming languages are essentially graphical representations of the conventional control flow structures. They allow to fully specify and modify the program, but do not provide supplementary information that would help analyze the program. Visual tools manipulating control flow constructions could benefit from larger-scale refactoring manipulations, such as function extraction, triggered by simple user actions.

2.3.8 *Execution Traces*

Solving software performance issues often requires analyzing the program execution on the real hardware with the real scale of the input data. This is often done by recording pertinent information during the actual program execution. This data can be then analyzed algorithmically or represented visually for the expert user. Multiple tools rely on the execution tracing to represent program behavior, therefore this subsection only exemplifies those where the execution trace is a part of primary visual representation.

TOOLS AND TECHNIQUES

TraceView is one of the first tool for visual representation of program execution traces [177]. It combines Gantt charts for displaying events, such as entering or leaving a function, performing an I/O operation, with graphs of numerical values over time. Given the system level of the information represented, *TraceView* is not intended for any manipulation of the original program. However, the visual representation can be navigated by pan and zoom.



(a) Jinsight, reproduced form [201]

(b) Google Chrome developer tools allow to trace webpage loading

Figure 10 – Execution traces are visualized as actions with duration along the time axis.

Jinsight is a visual tool combining *object-oriented hierarchy* and execution tracing [201] for Java. In addition to object state, it visualizes the call stack of the program as a horizontal icicle diagram with the vertical axis representing the execution time and the horizontal axis representing method calls demonstrated on the Figure 10a. By combining object creation and access times, object states and method calls, *Jinsight* lets the user identify repetitive and inefficient access patterns and prompt them to modify the corresponding regions of the code. However, being based on a trace of an already completed program, it is not able to influence the code or subsequent executions.

OPPORTUNITIES FOR INTERACTION

Execution traces are rich data sources for program analysis that convey information about actual execution compared to static analyses or heuristics. They are visualized *after* the program terminates and describe a particular execution. Therefore, trace-based visualizations cannot serve as basis for modifying the actual program unless tightly connected to a static program model. Some tracers operate as the program runs and provide information about, e. g., function calls. In this case, the user may interact with the tracer connected to the debugging mechanism or the language run-time in order to control the function call stack, parameters or force immediate returns.

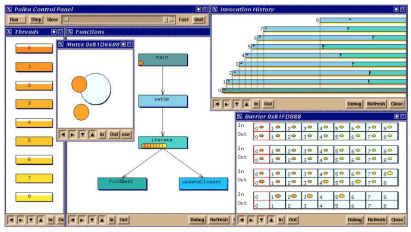
2.3.9 *Multithreading Synchronization*

Although mapping parallel execution to the sequential nature of the code is complicated in general, synchronizing execution of the parallel threads is particularly challenging as it leads to a whole range of undesired effects. Multiple visualization techniques address specifically synchronization and related language constructs.

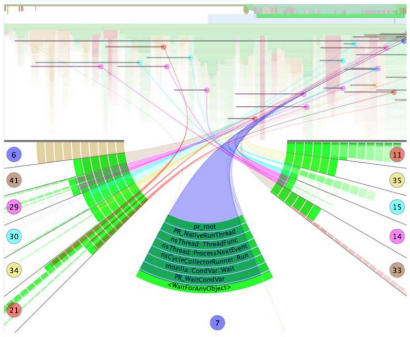
TOOLS AND TECHNIQUES

Gthreads is one of the first approaches to visualizing thread-level synchronization primitives [235]. *Gthreads* is an implementation of POSIX thread specification (pthread) that traces thread-related events and stores them in a file for a subsequent visualization in Polka toolkit [233]. In addition to Gantt charts capturing which function was executed by which thread at a specific time, Gthread represents synchronization primitives, namely mutexes and barriers, and thread states with respect to them as shown on the Figure 11a. The visualization is incremental and is dynamically updated as the program executes, however the user cannot influence the execution in any way.

JaVis leverages Java Debug Interface to display `synchronized` method calls within Java classes as a UML sequence diagram [183]. It allows to follow the call stack that led to calling a synchronized method for each thread and thus detect a deadlock or an excessively long waiting.



(a) Gthread/POLKA, reproduced from [235]



(b) SyncTrace, reproduced from [136]

Figure 11 – Tools for visualizing synchronization in parallel programs offer a combination of ad-hoc views of synchronization primitives and Gantt-like charts.

Being based on tracing, JaVis provides interaction with the visualization through pan and zoom, but not with the underlying program.

SyncTrace is a recent visualization technique for parallel software execution traces targeted at discovering synchronization and I/O problems [136]. It combines a Gantt-like chart with elements of an icicle diagram to display function call stack over time. It also features a bended Gantt-like chart, similar to SunBurst visualization technique, in order to provide focus+context display of the execution trace as depicted on Figure 11b. *SyncTrace* provides rich interface for selection, aggregation and filtering of traced events through direct manipulation and inspector views. At the same time, it does not allow to manipulate the program itself or immediately access the source code from the visualization.

OPPORTUNITIES FOR INTERACTION

Synchronization primitives require specific support from the hardware and are managed by the operating system as well as by the language runtime [35]. Therefore, any interaction with these mechanisms would also require support from the OS or the runtime. Modifying the program statements accessing this functionality may help resolve synchronization problems, but requires the program model to identify synchronization primitives and to analyze data flow between these primitives in order to ensure program correctness after such restructuring. Existing tools only provide coordinated selection to help the user identify the relevant code regions.

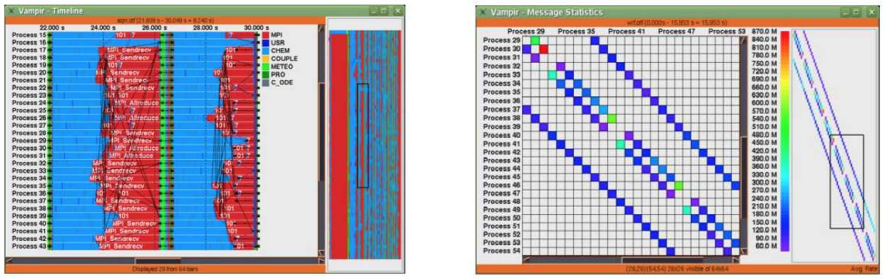
2.3.10 *Communication*

Message passing is a frequently used paradigm to communicate data between different objects in object-oriented systems. It allows

to better model a distributed system where objects may belong to different parts of the system. Message passing may influence program performance and, when used for synchronization, correctness. Therefore, some tools for visualizing communication in a distributed environment also visualize multithreading synchronization.

TOOLS AND TECHNIQUES

Jerding *et. al* used the Polka toolkit [233] to visually represent message passing in an object-oriented system [130]. Their prototypes feature multiple views representing object creation and deletion as well as message-based communication between different objects over time. Each communication line is depicted as a line connecting files, classes or objects. Analyzing the communication data, the tool identifies recurrent interaction patterns that can be represented in a concise node-link form. These patterns are based on the message sending traced during the program execution thus removing the possibility to restructure the program. Jerding's tool only supports structured navigation and filtering mechanisms available in the Polka toolkit.



(a) Communication view in Vampir (b) Heatmap view in Vampir

Figure 12 – Communication visualizations may focus on individual communications or on the cumulative statistics of exchange. Images reproduced from [144]

Vampir features an interactive visualization of MPI-based program traces for performance analysis [144]. It combines Gantt chart of program call stack with communication diagram similar to the Jedring *et. al* tool demonstrated on the Figure 12a. In a complementary view, Vampir displays the communication intensity between multiple processes as a colored table as shown on the Figure 12b. It offers interactive functionality for navigating visual representations: semantic zooming from overview to detailed views, to numeric values, traditional pan and zoom techniques, filtering capabilities, coordinated selection of communicating functions in multiple views. Nevertheless, it does not allow to interact with the program itself or directly move from visual element to the relevant place in the source code.

OPPROTUNITIES FOR INTERACTION

Message passing visualizations are typically based on tracing, but

feature visual elements specific to parallel or distributed execution. Whenever they rely on tracing, they do not allow to modify the program directly from the interface for the same reasons as tracing tools. When communication is used as a means of synchronization in the distributed systems, interaction opportunities remain the same as for multithreading synchronization except for additional support from the language runtime if the user interacts with the running program. Interactively modifying the communication itself may consist in changing the communicated information, its structure or its propagation method and participants, but would require a system-level communication broker that is connected to the interactive visualization tool.

2.3.11 *Numeric Metrics*

Large scale software systems require a rapid overview of the system and high-level comparison capabilities. Software metrics allow to compute numeric values quantifying various properties of the software, from its complexity to performance. As any other numeric data, software metrics may be represented visually using diverse techniques. However, specialized tools allow to relate this information to the particular aspects of the software such as source file structure.

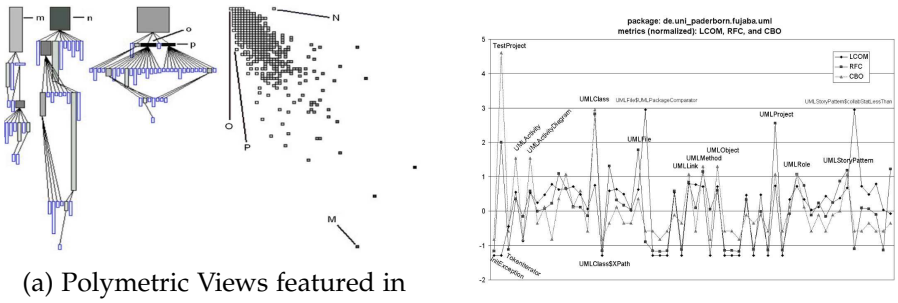
TOOLS AND TECHNIQUES

Shimba is a tool set that combines software metrics and their visualizations with object-oriented hierarchy visualization [240]. It visually associates numerical measurements of the class properties, such as depth of inheritance tree or weighted methods per class [157], to the graphic aspects of the node-link diagram. It also provides a class-indexed chart for comparing measurement values across multiple classes of the system. *Shimba* is based on *Rigi* program visualization environment [246] that enables the user to construct their own visualization using a Tcl-like scripting language.

Polymetric Views [154] featured in *CodeCrawler* [155] allow to represent up to five metrics in a single view with various layout methods. CodeCrawler's layouts are essentially structured node-link diagrams or scatter plots where several metrics are rendered as positions of visual objects. *CodeCrawler* enables the user to directly manipulate the visualization by panning, zooming or filtering. It also features an integrated source code editor allowing the user to quickly switch between the code and the visualization.

OPPROTUNITIES FOR INTERACTION

Software metrics provide brief overview of software properties [100]. In general, visualizations of software metrics do not allow to manipulate the program itself otherwise than by changing its source code



(a) Polymetric Views featured in CodeCrawler, reproduced from [155]

(b) Index charts featured in Shimba, reproduced from [240]

Figure 13 – Code metrics can be represented using common information visualization techniques or specialized program-related approaches.

in the coordinated view. The metrics are aggregated abstractions that cannot be unfolded back to the program unambiguously.

2.3.12 *Software as Data Source*

Computer software is a complex, constantly evolving artifact rich in information. It has a long life cycle with numerous tasks for specification, modification and maintenance. While software specification is still predominantly [5] made using textual representation , i. e. code, program analysis is often supported by supplementary visual representations to improve software understanding.

Program understanding is a prerequisite for the program modification. Yet most software visualization tools do not allow to modify the program directly from the visual representation, but require the developer to modify the code. They provide coordinated selection and dynamic update mechanisms to support this analysis, but they introduce a gap between the representation used for analysis and the representation used for modification.

▷ *Most software visualization tools do not allow to modify the program.*

Beside long-lasting argument between visual and textual programming [33, 123, 263], this situation is due to two factors:

— the program model underlying the visual representation is not precise enough, i. e. does not fully specify the program as the code does, but rather abstracts it;

— the visualization does not provide a meaningful backwards mapping from visual manipulations to the program changes.

For example, *execution traces* provide information about an individual execution of the program, not about the program itself. Therefore, trace-based tools require a separate program model connected to the trace in order to allow program manipulation from within the visual

5. *Scratch* is the first visual language in TIOBE programming language popularity index (`http://www.tiobe.com/tiobe-index/`) and occupies 22nd place with 1.039% score as of August 2016.

representation. On the other hand, *class diagrams* such as UML rarely map the on-screen position of the visual element to the program-related values making the user-initiated element movement useless for program transformation.

In order to enable program restructuring from the visual interface, it is necessary to analyze typical visual representations of software and individual visualization primitives considering how they could support interaction between the user and the program model.

2.4 VISUAL ELEMENTS OF SOFTWARE VISUALIZATION

Before selecting the appropriate model for visual program manipulation, we review the properties of visual representations with respect to user input. First, we analyze visual properties [45] and give some ideas on how they can be manipulated (directly or through instruments) in the program restructuring interfaces. Then, we evaluate the general-purpose visualization techniques that we found in the software visualization tools reviewed in the previous section.

2.4.1 *Visualization Primitives*

POSITION 2D spatial coordinates are often a primary choice in visualization. They afford direct manipulation using typical spatial input devices (mice, trackpads). Systems for visualizing large quantities of data often support spatial navigation and distinguish between navigation and manipulation through (quasi-)modes or instruments. Spatial coordinates can be selected with high precision thanks to advanced interaction techniques such as semantic pointing [34] or snapping [32]. A third dimension is sometimes added to the visualization. It trades off information density for interaction complexity: conventional input and output devices are two dimensional and 3D visualizations require specific interaction techniques to support manipulation and navigation [67]. For animated visual representation, position in time may be considered as another mapping target, especially convenient for the temporal information [52]. However, it may only be used to render information: the users cannot navigate in time time dimension unless it is mapped to a spatial dimension.

▷ *Position is directly manipulable.*

MARKS, SIZE AND SHAPE Information visualization techniques use various mark types classified by Bertin [31] and revisited by Mackinlay [174] and Card [45]. Mark primitives basically correspond to the one-, two- and three-dimensional geometrical primitives: points (1D), lines (2D), areas (2D), surfaces (3D) and volumes (3D). Marks are typically used to represent categorical information where different shapes of the mark are mapped to different categories. The user can interact with the mark and thus change the category by, e. g.,

clicking. Two- and three-dimensional marks also allow for changing their shape. However, the interaction to do so would often require either changing the *position* of shape control points (corners, rotation centers) or selecting a shape from a list, also represented in space or time. Nevertheless, as the information is mapped to the mark type or its shape, the interaction to change it may be decoupled from the spatial and temporal position of its individual parts. For example, the number of loop iterations may be mapped to the number of corners of a regular polygon without connecting any information to their spatial position. The size of the mark along any of its dimension may encode additional information: for example, multiple 2D areas may have the same area while having different linear sizes as long as their product remains the same. Interaction with linear sizes may use essentially the same techniques as interaction with the spatial position by virtually fixing a point of the object and stretching it by, e. g., dragging another point.

▷ *Interaction with marks and shapes is typically mediated by other objects.*

RETINAL PROPERTIES Card and Mackinlay identified the following properties relevant to the visualization: Color, Connection, Enclosure, Texture [45] as well as Size and Shape discussed above. In modern interfaces, it makes sense to also include Opacity. Color is ubiquitous in visualizations and is often used for representing categorical (color coding) or ordinal ("heat maps") data. However, interaction with the color itself is complex and requires a set of specific tools [128]. Basic interactions with color involve selecting one from the list or using a mixing palettes, both of which rely on spatial representation and repositioning rather than on interaction with the color itself. Similar argument holds for Texture. On the other hand, Connection and Enclosure allow to encode the connection between objects without invoking any category. They lend themselves to the interaction that involves creating, deleting or modifying the relation between objects.

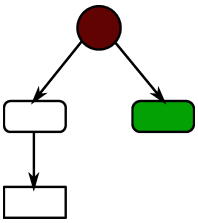
2.4.2 *Program Visualization Techniques*

CODE LINES Representing software as its code allows, in the first place, to resort to the multitude of text input and editing techniques from simple typing and copy/paste, to search and replace through regular expressions, to advanced modern techniques for interacting with text [244, 264]. In many cases, programmers use secondary visual cues in the text to convey additional semantic meaning, for example horizontal alignment represents blocking and line spacing separates logically independent parts within one block. Minimizing the code representation, e. g. like SeeSoft [86], allows to keep these secondary cues while abstracting away the textual content. This minimized representation allows for moving the lines and blocks of code

```
for (int i = 0; i < N; i++) {
  D[i] = 0;
    for (int j = 0; j < M; j++) {
      A[i+j] += B[i] * C[j];
      B[i] += D[j];
    }
}
for (int i = 0; i < N; i++) {
  <...>
}
```

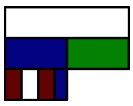
Minimized code lines

around or between files, modules or other higher-level entities. Code folding techniques achieve the same result still keeping parts of the code readable [188]. To some extent, the full text representation supports the same restructuring, but is limited by the relatively small amount of visible code.It also supports source-code level refactorings such as function extraction or block fusion [184].

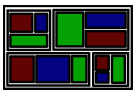


Node-link diagram

NODE-LINK DIAGRAMS Node-link diagrams have a strong incentive for repositioning the nodes while the links are being updated simultaneously. However, the node position is rarely mapped to the software-related data, it is rather used for visual arrangements or to untangle links. Widespread node-link diagram editors routinely provide separate tools or modes for creating, deleting or retargeting links between nodes. The links may also be associated with additional information by using color, thickness or texture, in which case supplementary manipulation tools are introduced for these properties.

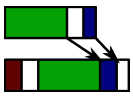


Icicle diagram

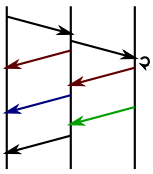


TreeMap

ICICLE DIAGRAMS AND TREE MAPS Both icicle diagrams and tree maps were created to represent hierarchical data. Contrary to node-link diagrams that rely in visual connection of elements, tree maps are based on enclosure and icicle diagrams on enclosure and adjacency of elements. This allows to refocus interaction on the relation since directly moving the element on the diagram changes its position with respect to other elements and thus their relation. For example, putting an element inside the area of another element in the tree map corresponds to creating the hierarchical relation between these two elements. Icicle diagrams, on the other hand, allow for easily moving the element up and down in its hierarchy.

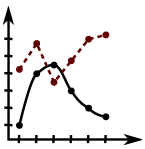


Gantt chart

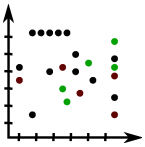


Sequence diagram

GANTT CHARTS AND SEQUENCE DIAGRAMS Gantt charts represent temporal information for multiple objects, namely for multiple parallel processes, as well as communication and synchronization. Elements in the Gantt diagram are often sparsely distributed and may be directly moved along the time axis, reordered or moved between rows. Communication and synchronization points, represented by lines similarly to node-link diagrams, may be used as landmark points or modified using the link editing techniques for node-link diagrams. They may also be reordered following the position of the associated elements. Sequence diagrams abstract away the duration of the process to focus on the order and communication. Same link manipulation techniques apply given that spatial dimensions offer a strong semantics of time and parallelism as opposed to node-link diagrams with arbitrary or nonexistent spatial mapping.



Index chart



Scatter plot

INDEX CHARTS AND SCATTER PLOTS Index charts include key axes and value axes. The points in the chart, usually connected to

each other, may be moved along the axes, as well as grouped, aligned or distributed, e. g. to create a continuously growing line, with appropriate changes made to the value. One can also interact with the key axis by changing the order of elements, for example if it is related to the order of statements in the program. Some keys may be deleted, or new keys may be added given their initial value is selected by an unambiguous rule, e. g. interpolated between adjacent values. Scatter plots do not feature lines connecting points, but the points remain manipulable in mostly the same way: displacement, alignment, grouping. Elements on the scatter plot axes can be individually modified similarly to the key axis of the indexed chart if they represent discrete values.

Considered separately, visualization techniques support various ways of interaction. For example, visual elements can be directly manipulated by dragging, their color can be changed indirectly through inspector views or filling instruments. However, only few software visualization techniques map this interaction to the changes to the program they represent.

2.5 INFINT DESIGN SPACE

We propose the *InfInt* design space that captures *Inf*ormation and *Int*eraction aspects of the software visualization techniques. It makes explicit the connection between the program-related data and the associated visual element. It also includes the *degree of interactivity* the technique provides for this particular element, as described below. In general, *InfInt* consists of three dimensions:

↩ Dimensions of the InfInt design space.

— category of the program-related *information*;
— associated *visual element*;
— degree of *interactivity*.

Information categories correspond to those described in Section 2.3 and visual elements to ones analyzed in Section 2.4.

2.5.1 *Degree of Interactivity*

For each of different information aspects visualized by various techniques, we evaluate the level of interaction the given technique provides for this particular aspect. We define four interactivity levels:

No Interaction — no interaction is possible with this information, the visual representation is a static, non-navigable picture;

General-purpose interactive visualization — the user is able to interact with the visual representation itself, but not with the program being visualized, using conventional techniques for this kind of visualization;

Program-specific interactive visualization — the user is able to interact with the visual representation itself, but not with the program, using interaction techniques that take into account the specificity of the underlying software-related data;

Program Restructuring — the user is able to change the underlying program by interacting with its visual representation.

For example, a 2D node-link diagram representing class hierarchy for an object-oriented software that supports pan-and-zoom interaction features *General-purpose* interaction as these techniques are used for representing any tree-structured data [120]. However, if the same representation allows to rearrange the layout according to class inclusion into packages, it is considered to have *Program-specific* interaction. Finally, if the user is able to change the actual class hierarchy by moving the class node in the node-link diagram, the technique allows for software *Restructuring*.

Coordinated code-visual selection is not considered as a tool for program restructuring.

2.5.2 *Mapping Techniques into the Design Space*

A software visualization technique is mapped into the *InfInt* design space by first identifying all visually represented aspects of the software and putting them on the *Information* dimension, then associating these aspects with the *Visual Elements* and, finally, specifying the degree of *Interactivity* for each of them.

Appendix A illustrates the *InfInt* design space with all dimensions. Figure 14 provides a compact view of *Information* and *Interactivity*. Software visualization techniques appear as many times as they cover *Information* categories. It demonstrates the lack of program visualization tools that support restructuring without code modification.

A user may refer to this design space in order to chose an appropriate tool, or a set of tools, for the task at hand. For example, the user willing to understand the slowdown caused by long communication on a remote cluster computer is interested in a tool for visualizing communication from program execution traces. Vampir [144] happens to be the perfect match. However, if this user wants to modify the program from the visualization, no technique currently offers enough interaction to perform this task. This information may be used by designers of the software visualization tools.

2.5.3 *Implications for Design*

Program restructuring is an iterative process that consists of two repeatedly alternating steps. First, *understanding* the program, its properties and behavior, identifying problems, e. g. incorrect results or low performance, if necessary. Second, *modifying* the program to change its properties and behavior and understanding the effects of change.

		No interaction	Interactive Visualization General-purpose	Interactive Visualization Program-specific	Program Restructuring
code-related	Code	Tarantula	Sv3D SeeSoft SeeSlice		
	Slicing	*Krinke2004*	SliceBrowser SeeSlice		
	Iteration		3D-ISV Tulipse Deco		
structure	Hierarchies		Shimba Jinsight JaVis CodeCrawler	SHriMP *Beck2011* JavaVis	
	Memory	*Veroy2013*	Jinsight Deco MAPA SHriMP *Jerding1997*	GCSpy MemGraphs	StarDiagram
	Data			MemGraphs JavaVis StarDiagram	
	Control	*Krinke2004* SeeSoft Tarantula Sv3D	Tulipse JaVis	Jinsight SHriMP	TiledGrace StarDiagram
execution	Traces		Gthreads Vampir JavaVis JaVis	SyncTrace Jinsight	
	Sync.		Gthreads JaVis JavaVis	SyncTrace	
	Comm.		Vampir *Jerding1997*		
	Metrics	SeeSoft Sv3D	Shimba Deco	*Beck2011* CodeCrawler	

Figure 14 – InfInt Software Visualization Design Space combines the notions of software-related Information and the degree of Interactivity.

Software visualizations are mostly designed to improve the understanding of the program and to help finding solutions to existing problems [80]. It reduces the time required for understanding and, occasionally, the number of restructuring iterations thanks to better understanding and thus more efficient modifications. However, it does not affect the program modification, which is still to be done in the original, predominantly textual, representation. Worse, it introduces the need to establish the bidirectional mental mapping between the textual representation and the visualization on every iteration step.

Interactive program restructuring implies using the same visual representation for *understanding and modifying* the program. The mapping between the textual and the visual representation is then performed only once as shown on Figure 15. We expect interactive program restructuring to significantly reduce the program restructuring time.

In order to enable interactive program restructuring through the visual interface, the system should provide:

↩ Elements of Interactive Program Restructuring.

— a model of the program that contains information of a certain type (quantitative, categorical, etc.) and unambiguously maps modification of this information back to the program;

— a way to ensure program correctness after modification or, in its absence, a way for understanding the changes made to the program;

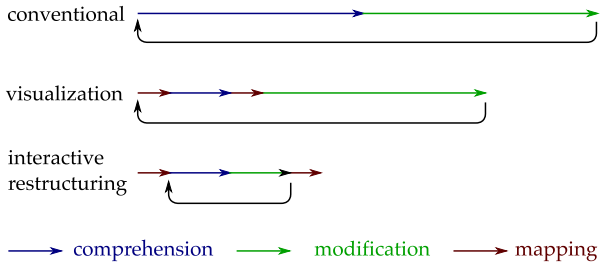


Figure 15 – Software visualization reduces program comprehension time at the expense of the mapping between different representations, but does not affect the modification time.

— a mapping of this information to visual elements that support the desired interaction and can be queried for the new values in a consistent way.

To be efficient, interactive software visualization should not only provide the user with pertinent program-related information, but allow to manipulate it easier than the source code. Representation transition techniques that facilitate the perceptual mapping between textual and visual representations, such as Fluid Documents [53] or Gliimpse [83], may allow to minimize the time overhead for establishing the mapping between multiple representations of the program.

2.6 DISCUSSION

VISUAL AND TEXTUAL PROGRAM REPRESENTATIONS Visual representations of programs are usually opposed to textual representations, be it for creating programs through visual or textual languages or for understanding and restructuring programs [195, 263]. Developers may be reluctant to visual representations of programs despite empirical evidence in favor of visualization [80, 263].

PRIMARY AND SECONDARY VISUAL NOTATION Comparing readership skills and graphical programming, Petre argues that significant expertise is required for using both textual and visual programming [202]. Moher *et. al* study the comprehensibility of textual and visual programs on the case of Petri nets [187]. The find that, despite their inherently graphical nature, visual specification of Petri nets does not outperform textual forms. It rather depends on the secondary visual cues, such as alignment, not included in the specification. CPN2000 improves interaction with the Petri nets by relying on secondary cues [25, 26]. Generalizing, both textual and graphical program representations include primary notation, related to program description, and secondary notation, not affecting the program and used by experts to improve comprehension. Secondary notation can be modified manually, e. g. indentation and line spacing in textual representation, or provided automatically, e. g. syntax highlighting

in textual representation or automatic node-link diagram alignment in visual representation. This argument may be extended to textual and visual representation of programs that are not necessarily meant for programming, but for analysis and modification. A user interacting with the visual representation of the program may, in fact, interact with the primary or the secondary notation. In the former case, this interaction may potentially affect the program itself. In the latter case, however, the interaction is performed for the sake of analysis, understanding or pure aesthetics.

REFERENCE MODEL FOR INFORMATION VISUALIZATION Enabling interactive program restructuring may modify the reference model for information visualization of Card *et. al* [46] by making the mapping between program and its model bidirectional (support unambiguous mapping of the modifications) and by adding a back-link from the view to the model to reflect interaction, see Figure 16.

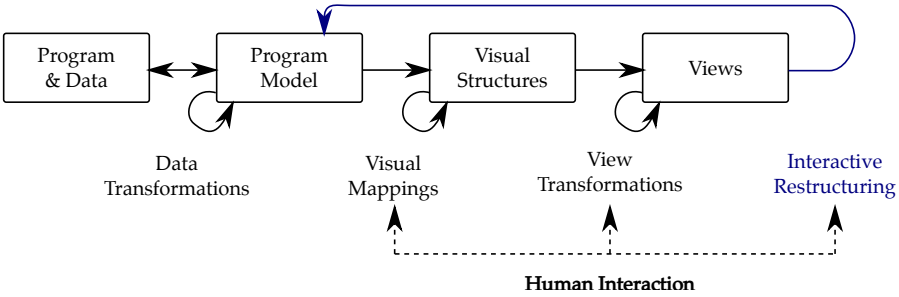


Figure 16 – Adding interaction back-link to the reference model for visualization [46] to support interactive restructuring.

VISUAL ANALYSIS OF TEXTUAL PROGRAMMING Modern code editing tools use a variety of graphical properties — colors, typography, animation, miniaturized graphical elements — blurring the line between textual and visual representations. Conversy applied Bertin's Semiology of Graphics [31] and ScanVis descriptive model [71] to analyze the perception of programming code [69, 70]. He demonstrated that code perception fits into the model for perceiving graphics and, furthermore, that secondary visual cues in the code are used to improve this visual perception. Conversy argues for unifying the concepts of visual and textual programming.

INSTRUMENTAL INTERACTION FOR PROGRAM MANIPULATION We develop this argument by stating that the program is a domain object, in a sense defined in Instrumental Interaction paradigm [23, 24, 26], that is not directly accessible to the user. It is rather visible through a set of *representations* that highlight some aspects of the program while potentially hiding others. Program representations should be designed not only to provide relevant information, but to enable interaction with it. In this sense, visual views and code are

Domain object is the object of interest for the user in Instrumental Interaction model.

complementary representations of programs. An appropriate representation, or a set of representations and a composition technique, should be chosen based on the relevancy of the notation they provide for the task at hand, accessibility of this information and possibility to interact with it. A representation may be useful for the user if it provides *new* information compared to other representations or if this information is *better available* for the user to understand or interact. This interaction affordance is specific to the user, but should be considered during the representation design process [103].

▷ Program representation should provide the user with either new data, or make the existing data easily manipulable.

MULTIPLE REPRESENTATIONS OF A PROGRAM Representations can be grouped in a single interface for program manipulation, with advanced techniques for coordinating manipulation. Indeed, most modern IDEs include multiple representations of the software: code, file structure tree, node-link diagram of the class hierarchy, call stack for debugging, etc. Such composite systems could simplify the transition between different representations through, e. g., coordinated selections or animated transitions. Representations can also be stacked on top of each other as long as there exists a consistent bidirectional mapping between them. For example, a textual representation of the program may be represented grouped in blocks which, in turn, can be visually represented as a node-link diagram.

2.7 CONCLUSION

In this section, we reviewed the related work in software visualization techniques and tools. We proposed a design space to classify these techniques according to the program-related *information* they represent and the degree of *interaction* they provide to manipulate it. Analyzing the existing work, we observed that few techniques allow to perform interactive program restructuring directly from the visual representation while most require code modification using separate tools. We argue that the lack of interaction with the program is due to "read-only" program models used in software visualization tools and to imperfect mappings between program-related quantities and manipulable visual elements.

In order to design an efficient system for interactive program restructuring, one should first select the program model that supports modification and interaction, then design a visualization technique helping to solve a particular problem with program restructuring thanks to newly visualized information and finally propose techniques to interact with this visualization. In the following chapters, we focus on the program parallelization and performance optimization on the level of individual statements and control structures by extending an advanced program model to support interaction.

3 POLYHEDRAL PROGRAM REPRESENTATION

3.1 PROGRAM REPRESENTATIONS

Automatic parallelization as a part of program optimization process has been a cornerstone of compiler research since its genesis [265]. In the last decade, it has become increasingly important due to end of Dennard scaling, that allowed to increase processing performance by scaling down the element base size and scaling up the clock frequency, that resulted in parallelism becoming the predominant path for increasing performance [239]. However, fully automatic parallelization remains far out of reach in the general case due to the complexity of both the program analysis process and the targeted parallel architectures, although it is feasible for domain-specific language and particular cases. With the anticipated end of multicore scaling and the advent of energy-saving approaches to deal with power limitations [91], computer architectures and programming models become increasingly sophisticated and difficult to handle even for the expert programmers resulting in an underuse of processing power in the best case and in intricate intermittent parallelism errors during the execution in the worst case [199].

Chipmakers are busy designing microprocessors that most programmers can't handle.
— D. Patterson

Manual parallelization being too tedious and error-prone, most modern approaches to parallelization focus on semi-automatic methods where a programmer may provide extra information about program behavior and properties. An automatic tool may then exploit this information for parallelization or provide the programmer with feedback as to problems preventing parallel execution. These semi-automatic approaches require a program representation that is easy for a human programmer to understand on one hand, and easy for an automatic tool to analyze and extract parallelism on another hand.

Given the complexity of architectures and programming models, making a representation better understandable for the human programmer requires powerful high-level abstractions, such as functions and call graphs or communication diagrams. Most software visualization techniques discussed in Chapter 2 rely on various high-level program representations as a base for visualization. High-level representations often bring extra information about the program or its particular execution, such as performance counters, software metrics, execution time statistics or bottleneck regions, that is not necessarily visible in the original code representation. This extra information is important for taking decisions as to what part of the program to optimize, how and to which extent.

However, these representations remain limited for program transformation as they often provide *aggregate* data where particular details of the program were abstracted away. For example, if a function with no side effect is called multiple times in the same block and is reported as "5 repeated calls to pure function f() in a block", the developer is unlikely to be able to remove some of the calls and cache the result unless they go back to the code.

On one hand, this is typically caused by the fact that the high-level representation does not fully represent the program anymore in a sense that a lower-level representation cannot be fully reproduced, or that it is not modifiable. Even when it is, the question of program behavior equivalence after a modification arises and is challenging to answer.

On the other hand, automatic tools operate on lower-level representations, bringing the program closer to the hardware level in order to better exploit available resources. Essentially, a compiler performs multiple transformation between representations. For example, LLVM C frontend (Clang) takes the program code as text strings, which it transforms into tokens, then builds an abstract syntax tree and performs series of transformations on it; it then builds a control flow graph, from which an intermediate representation (that may have a textual, a binary or a graph form) featuring static single assignment is generated. The process with abstract syntax tree transformations is repeated for the intermediate representation to result in a sequence of machine codes for the target architecture [160].

While all of these representations capture full information about the program, they are tailored for efficient automatic processing rather than human understanding due to the abundance of low-level details related to the functioning of both the compiler and the target architecture. They also remove multiple high-level structuring parts that are not important for the program execution, but crucial for understanding, such as classes and other programming model abstractions, comments, variable names or indentation. The transformations inside compilers are so drastic that a reconstructed code, whenever such reconstruction is still possible, will not be any close to the original code and often as hard to analyze as the internal representation. Finally, the internal representations offer little information about the program that is not already available in the code or the platform specification.

While some approaches build representations stacks, including visual representations [4, 251], on top of low-level representations like syntax trees, they are mostly used for teaching or performing compiler design. Even if these representations can be manipulated by their users, the manipulation instruments and paradigms are connected to the low-level details decreasing their interest for an aggressive program optimization.

In order to build a representation for interactive program restructuring we must find a program model that:

— is at a level low enough to fully model the program and allow answering questions like program equivalence;
— provides extra information about the program relevant for the optimization that is not easily available in other representations;
— allows for high-level manipulation tools understandable for the user thanks to clear effects on the representation and the underlying program.

↩ Requirements for the model that supports interactive program restructuring.

The ideal high-level representation for interaction between the human developer and the automatic tool should provide an unambiguous explicit mapping between the high-level user input and the fully modifiable program model.

Most of low-level representations are centered around the notion of *statement* or *instruction*, close to the source code or to the processor operation. However, most execution time is spent in iteration where the statements are executed repeatedly, and this iteration offers the most opportunity for optimization. The *polyhedral model* is a particular case of abstract representation that captures individual executions of iterations providing extra information for the loop-level parallelization and vectorization, that are otherwise impossible to achieve due to the lack of analysis granularity in statement-level models [98]. It fully represents execution of a subset of imperative programs allowing to demonstrate independence of certain program parts and ensure program equivalence in case of transformation. It has been the basis for multiple major advances in compiler technology over the last decades [37, 96, 206] and is present in major production compilers such as GCC [204], LLVM [112] and IBM XL [39]. Thanks to the strict mathematical formalism it allows to concisely express complex program restructurings and provide and unambiguous back-mapping to the other program representations, necessary for building efficient program manipulation instruments. In the rest of this chapter, we introduce the polyhedral model and constrain it so that it can serve as a basis for a stack of visual representations and high-level program manipulations leveraging user intuition while guaranteeing program correctness.

A statement is a unit of a programming language that expresses an action to execute, e.g. `a = 42;`

An instruction is the smallest piece of work executed by a processor, e.g. put 42 in a register.

3.2 THE POLYHEDRAL MODEL

The polyhedral model is an algebraic representation of a subset of imperative programs that encodes individual executions of statements, e.g. multiple iterations of a loop [98]. These executions are represented as sets of points in a multidimensional space constrained by affine hyperplanes forming a polyhedron, hence the name of the model [165]. Thanks to its geometrical interpretation, it allows for an intuitive visual representation that may leverage the previous knowl-

Polyhedral model is also referrled to as polyhedron model or polytsope model. In geometry polyhedron is a 3D polytope. In algebra, polytope is a bounded polyhedron.

edge of the user. Figure 17 illustrates a part of the polyhedral representation and its possible visualization for the given code snippet. It supports aggressive program restructuring through concise high-level transformations [104, 109, 207, 221] and may be easily transformed back to the code [16]. These considerations make the polyhedral model a perfect backend for interactive program restructuring.

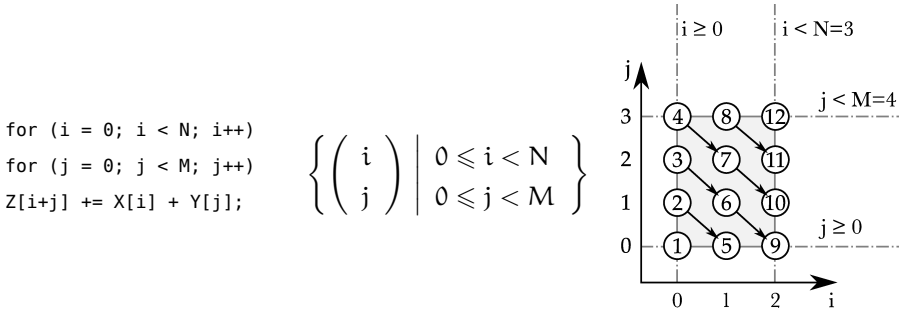


Figure 17 – Example of the source code, algebraic abstraction and visualization in the polyhedral model.

3.2.1 *Development of the model*

The first idea of modeling individual executions of repetitive computations for extracting parallelism to build a parallel processor is found in the seminal paper by Karp, Miller and Winograd on uniform recurrence equations [135]. In his paper on Fortran DO loop parallelization, Lamport uses hyperplanes to separate independent parts of the loop nest [151]. A combination of hyperplanes defines a (possibly unbounded) polyhedron. Quinton proposed to use convex multidimensional sets with a timing-function (scheduling function) for dependence analysis during hardware generation for parallel processing [215].

Since this linear-algebraic representation may benefit from a large and formal mathematical apparatus developed for the optimization problem, such as linear programming [75], it could be leveraged for improving program performance. However, two problems prevented its immediate application: (1) programs could depend on and be controlled by their input, i. e. be parametric, otherwise they could be replaced with the computation result once known and (2) defining a partially parallel traversal order in terms of control flow constructs was a difficult task. The first problem was addressed by the introduction of parametric integer programming [93] that allows to find optimal solutions to a linear programming problem depending on the values of integer parameters. The second was first solved by Ancourt and Irigoin in their algorithm for scanning polyhedra with DO loops [6]. The name "polytope model" was first proposed by Lengauer [165] in his work on automatic loop parallelization.

In the model, a parameter is a value that is not known during the optimization, but is guaranteed to be constant during execution. Parameters are processed symbolically.

Other works on loop parallelization followed including those of Feautier on polyhedral scheduilng [95, 96], optimal parallelism extraction algorithm by Darte [77] and the approach to minimize synchronization by Lim and Lam [166]. Boulet gives a detailed survey of loop-level parallelization in the polyhedral model [41].

3.2.2 *Limitations*

STATIC CONTROL One of the most stringent limitations of the polyhedral model that prevents its widespread application is the requirement for *staticity of control*, i. e. control flow of the program should not depend on the input data, except for parameters guaranteed to be constant during execution, as only static control constructs can be expressed and analyzed statically, without running the program. This limitation can be alleviated either by parametric overapproximation of the data-dependent control flow [28], generating inspector and executor loops in order to gather all non-static control in the former [252], dynamic change of schedules from sequential to parallel [11] or runtime speculation following the static analysis [131]. Manually parallelized loops can now be modeled as well [55].

Polyhedral model is applicable to static control parts, or SCoPs.

AFFINE CONDITIONS AND SUBSCRIPTS Early versions of the polyhedral optimization frameworks required that loop and branching conditions as well as array subscripts were affine expressions of outer loop iterators. This requirement comes from the need to model the program in linear-algebraic terms where, for example, an access `A[i*j]` cannot be expressed. Furthermore, using non-affine conditions for exact program analyses involves solving intractable, if not undecidable, problems [98]. Even with affine conditions, many algorithms in the polyhedral model are subject to combinatorial explosion that is alleviated by relatively small sizes of human-written programs. Parametric overapproximation is also suitable for circumventing the condition affinity limitation [28] as well as the inspector/executor model [252]. Linearized multidimensional array or pointer accesses may be reconstructed in the polyhedral representation [114].

FOCUS ON LOOP TRANSFORMATIONS Although the polyhedral model allows to compute scheduling for the entire program part that it represents, loop-level transformations remain its primary strength. It is also limited by the affinity of the scheduling that does not allow to express, e. g., loop unrolling or parametric tiling. Recent work by Shirako et.al. demonstrates how polyhedral and syntactic-level program transformations can be efficiently combined [225].

3.2.3 *Workflow in the Polyhedral Model*

RAISING PROGRAMS TO THE MODEL Polyhedral frameworks operate on a mathematical representation of loop-based programs that differs substantially from the conventional syntax tree. The process of transforming code, or *raising* it into the polyhedral representation is mostly automated through parsing. In general, it identifies parts of the program with static control automatically or relying on user-specified annotations and then extracts from either the code or its syntactical representation, e. g. AST, the affine expressions that define loop bounds and access addresses. Integrated polyhedral analysis tools, starting at the code level, e. g. LooPo [109] or PIPS [126], rely on custom parsing techniques to extract polyhedra from code. Compilers supporting polyhedral optimization comprise a raising algorithm, such as GRAPHITE for GCC [204] and Polly for LLVM [112]. emph-Clan is an extendable polyhedral raising tool for C-like language syntax [20, 28] suitable for syntax-level experiments [92]. *Pet* leverages *clang* parser to support full C99 syntax in polyhedral extraction [254].

TRANSFORMING PROGRAMS IN THE MODEL
Automated Transformation — when raised to the model, the program can be transformed automatically or manually using the rich mathematical apparatus available. LooPo is arguably the first practical loop parallelizer implementation [109]. Automatic scheduling for loop-level-parallelism extraction and locality optimization is implemented in Pluto [37] and its successor Pluto+ [36], however some transformations, e. g. tiling, should be requested manually as a command-line option. `isl` library includes its own optimizing scheduler [253]. LeTSeE allows to explore the legal transformation space, i. e. iterate through all the possible schedules that preserve the original program semantics [205, 206].

Semi-automatic Transformation — semi-automatic program transformation interfaces in the polyhedral model allow the user to request specific loop-level transformations that are automatically performed by the framework. UTF defines classical syntactic-level transformations in the polyhedral model [138]. URUK enables the composition of transformation sequence decoupled from any syntactic form of the program [104]. CHiLL ensures that the transformation results in an equivalent program, i. e. verifies the transformation legality, via intermediate dependence checks [58]. AlphaZ complements control flow transformations by memory layout modifications for optimizing cache usage [271].

TRANSFORMATION SAFETY ANALYSIS Program transformation approaches, especially those with input from the user, rely on dependency violation checkers to ensure program semantics preservation.

A program transformation engine must ensure that the transformed program yields the same result as the original program. Even though proving program equivalence is undecidable in general case [133], one can demonstrate that a program *transformation* results in an equivalent program if it does not *violate data dependences* [97]. The concept of *dependence*, introduced by Bernstein [30], allows to bind together parts of the program that access the same data. A program transformation *violates* a dependence, if two dependent parts are executed in a different order after the transformation, and the program may yield different result. For the loop-level dependence analysis, the Omega test provided a relatively fast solution [211], but relied on imprecise heuristics for parametric cases. Parametric integer programming (PIP) enabled the analysis of the parametric control flow [94]. Eisenbeis and Sogno propose a preprocessing reduction and algorithm selection approach to simplify the dependence analysis [88]. Vasilache *et. al* demonstrated that with convenient representations and algorithms, the exact instance-wise dependence tests do scale in practice [250], which may be further used to automatically correct the user-specified transformation if it violates the original program semantics [18, 249].

By transformation, we understand permuting execution order of the program without otherwise modifying it.

GENERATING CODE FROM THE MODEL Finally, once the program is modified in the polyhedral model, it should be transformed back to the previous representation, code or abstract syntax tree. This task reduces to building the loop nests that traverse all integer points inside the transformed polyhedra. Ancourt and Irigoin proposed the first algorithm for code generation based on Fourier-Motzkin elimination algorithm [6]. Quilleré *et. al* proposed an algorithm allowing for trade-off between control overhead and code size [214], which was later improved by Bastoul in CLooG [16]. CodeGen+ explores the control-size trade-off even more [57]. PPCG provides code generation for graphics accelerators with CUDA architecture [255].

3.2.4 *Forms of Representation*

Programs in the polyhedral model can be represented using multiple different, often complementary, formalisms.

A parametric extension to the Chernikova algorithm that transforms a system of inequations into a set of multidimensional vertices is a cornerstone of the plurality of representations in the polyhedral model [161]. `PolyLib`[1], one of the first libraries for polyhedral computation, features the dual representation based on lines and rays on one hand, and on inequations on another hand [167]. `isl` library uses sets and maps of integer points bounded by linear constraints as its main abstraction [253]. Its scheduling mechanism internally uses a

1. https://icps.u-strasbg.fr/polylib/

tree-based representation annotated by sets of affine constraints [256]. The Parma Polyhedral Library uses ray-baysed representation of polyhedra defined in rational numbers and intersects polyhedra with integer grids to represent loop-related information [10].

3.3 REPRESENTING PROGRAMS IN THE POLYHEDRAL MODEL

The polyhedral model allows for multiple different mathematical descriptions and representations. Throughout this work, we use and extend the state-of-the-art *union of relations* representation. This representation was implicitly introduced in the Omega library [2] for operations on relations defined by Presburger formulas [139]. A relation defined by a Presburger formula containing disjunction [231] can be transformed into a *union* of relations, each of which defined by one disjunct. Thus Omega can be viewed as operating on unions of relations. The modern version of the representation that actually features unions of relations was implemented in `isl` library [3] and referred to as union of maps [253]. We use the `OpenScop` library [4] that features matrix-based representation of relations and unions thereof [17] rather than inequation forms of `isl`.

In `isl`, union of basic maps refers to unions of relations with equal dimensionality and union of maps to those with different dimensionalities.

A relation in our model is defined between multidimensional points in two spaces, the *input* space and the *output* space. Mathematically, it is an n-ary relation on $\mathbb{Z}^n$ where n_i dimensions are considered as input and the remaining $n_o = n - n_i$ dimensions as output. A *convex integer polyhedral relation* is a relation that is defined by a system of linear inequations that define a convex polyhedron in $\mathbb{Z}^n$. A concave polyhedron can be represented as the union of convex polyhedra, although the minimal decomposition into convex polyhedra is proven to be NP-hard [56]. Therefore, a non-convex polyhedral relation can be represented as a *union* of convex polyhedral relations.

An equation can be rewritten as two complementary inequations.

Relations in the model are *parametric* as they are described involving a finite number of symbolic integer constants, or parameters. The numerical values of these parameters are not known when operating in the model. These parameters can be used to model compile-time constants or values invariant throughout static control part execution.

All aspects of the static control part, namely the loop nest and its order of traversal as well as the memory accesses, are represented as unions of relations:

$$\bigcup_i \mathcal{R}_i(\vec{P}) = \bigcup \left\{ \vec{\imath} \to \vec{\sigma} \,\middle|\, \exists \vec{l}_i \in \mathbb{Z}^{\dim \vec{l}_i} : \bigwedge_j \left(f_j(\vec{\imath}, \vec{\sigma}, \vec{l}_i, \vec{P}) \geqslant 0 \right) \right\}, \quad (1)$$

2. http://www.cs.umd.edu/projects/omega/
3. http://isl.gforge.inria.fr/
4. http://icps.u-strasbg.fr/~bastoul/development/openscop/index.html

where f_j is an integer affine function with integer coefficients of input $\vec{\iota}$ and output $\vec{\sigma}$ dimensions, parameters $\vec{P}$ and local dimensions $\vec{l}_i$.

↩ Input and Output dimensions.

$$\begin{aligned} &f(\vec{\iota}, \vec{\sigma}, \vec{l}_i, \vec{P}) = \vec{k}_\iota^T \cdot \vec{\iota} + \vec{k}_\sigma^T \cdot \vec{\sigma} + \vec{k}_{l_i}^T \cdot \vec{l} + \vec{k}_P^T \cdot \vec{P} + q; \\ &\vec{k}_\iota \in \mathbb{Z}^{\dim \vec{\iota}}, \vec{k}_\sigma \in \mathbb{Z}^{\dim \vec{\sigma}}, \vec{k}_{l_i} \in \mathbb{Z}^{\dim \vec{l}_i}, \vec{k}_P \in \mathbb{Z}^{\dim \vec{P}}, q \in \mathbb{Z}. \end{aligned} \tag{2}$$

Although the terms affine and linear function are often used interchangeably, in geometry, affine functions include a constant term while linear functions do not.

Local dimensions are parameters of the function f are *existentially quantified*, i.e. they should exist and be integer. Local dimensions serve to express, e.g. , non-dense sets of points. For example, $\exists l \in \mathbb{Z} : i = 2 \cdot l$ requires i to be even. Local dimensions are individual for each relation in the union, hence their name.

For the sake of consistency, we consider a set to be a degenerate relation with *no* input dimensions, $\dim \vec{\iota} = 0$, that can be processed in the same way as all the other relations in the union of relations representation.

Note that an equation can be expressed as two non-strict inequations.

3.3.1 *Representing Statement Instances*

Consider the loop nest for polynomial multiplication in Listing 1. It comprises two tightly nested loops with one statement inside, and this is roughly how it would be represented in a syntactic-level model. The same element in the array `C` is accessed in multiple iterations of the loop. In the polyhedral representation, an iteration is identified by a vector containing values of all iteration variables ordered according to the loop nesting. For our example, an iteration is identified by (i, j). Two different iterations of the loop, (i_0, j_0) and (i_1, j_1), access the same element in `C` if $i_0 + j_0 = i_1 + j_1$. Accessing the same value makes these iterations *dependent* and requires to preserve their order of execution after the program transformation in order to preserve its semantics. In the absence of other dependences, groups of ordered dependent iterations may be executed in any order with respect to each other, including in parallel. The syntactic-level model, however, does not capture individual iterations of the loop nest since they are not visible in the syntax, and therefore does not allow parallelizing this loop.

```
for (int i = 0; i < N; i++) {
  for (int j = 0; j < M; j++) {
S:  C[i+j] += A[i] * B[j];
  }
}
```

Listing 1 – C code of the polynomial multiplication kernel.

The polyhedral model represents individual executions of a statement in the loop nest, referred to as *statement instances*, as points in multidimensional space. Each dimension of this space corresponds to a loop in the nest. The entire space is thus referred to as *iteration*

↩ Statement instance.

space. Coordinates of the point in the iteration space correspond to the values of iteration variables for this particular execution of the statement. A statement is encoded in the polyhedral model as the set of its instances inside the loop nests forming the *iteration domain* of the statement.

↩ *Iteration domain.*

The iteration domain for the statement is constructed by constraining the iteration space on each dimension by lower and upper boundaries of the corresponding loop. For the polynomial multiplication example, the outer loop is bounded by inequations $i \geqslant 0$ and $i < N$ and the inner loop is bounded by $j \geqslant 0$ and $j < M$. The iteration domain for the statement S from the Listing 1 is defined in (3).

$$\mathcal{D}_S(N, M) = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \middle| \begin{array}{l} 0 \leqslant i < N \\ 0 \leqslant j < M \end{array} \right\} \tag{3}$$

Polyhedral representation is inherently geometric and allows consistent visual representation of iteration domains: each loop corresponds to an axis, boundaries are presented as lines (or hyperplanes) and instances as points in a multidimensional space. For the polynomial multiplication example, the geometric interpretation is depicted on the Figure 18, given the parameter values $N = 3$ and $M = 4$.

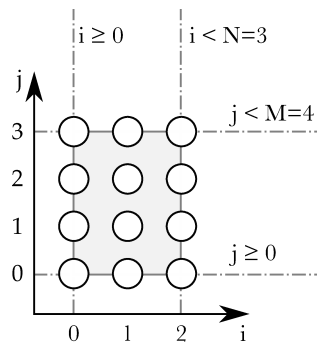


Figure 18 – Iteration domain of the polynomial multiplication kernel. Axes correspond to loops. Dashed lines represent loop boundaries. Circles correspond to statement executions.

The polyhedral domain also captures branch constructs surrounding the statement as long as their conditions are expressed as linear forms of outer loop iterators. Conjunction terms are represented as individual inequations, all of which should be respected inside the domain. Disjunction terms, on the other hand, transform the constraint system defining the domain into a *union*, elements of which differ only in the disjunction term. For example. the code in Listing 2 features a conjunctive condition (operator &&) in the upper bound of the inner loop and a disjunctive branch condition (operator ||). The iteration domain of the statement S in the polyhedral model is rep-

resented by a union (4). Its geometric interpretation is presented in Figure 19.

```
for (int i = 0; i < 3*N; i++)
  for (int j = 0; j < M && j < 8; j++)
    if (i < N || i >= 2*N)
S:    call_function(i, j);
```

Listing 2 – Example of the complex conditions in C code

All parts of the iteration domain union must have identical dimensionality as the same statement is only present in the one loop nest with a known depth. Even if two statements are textually identical, they are considered different statements in the model.

$$\mathcal{D}_S(N, M) = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \middle| \begin{array}{l} 0 \leqslant i < N \\ j \geqslant 0 \\ j < M \\ j < 8 \end{array} \right\} \cup \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \middle| \begin{array}{l} 2N \leqslant i < 3N \\ j \geqslant 0 \\ j < M \\ j < 8 \end{array} \right\} \quad (4)$$

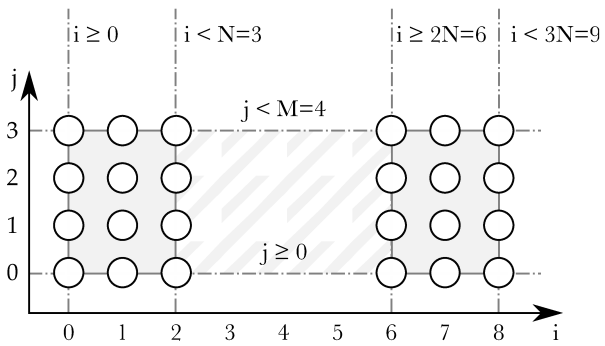


Figure 19 – Iteration domain union with embedded conditional.

3.3.2 *Representing Execution Orders*

Each point in the iteration domain, representing a statement instance, should be executed during the program run. The order in which the statement instances are executed is defined by the *scheduling relation* that maps the coordinates of the statement instance in the iteration space to multidimensional *schedule space*, which defines the execution date and place. Execution place refers to the physical unit of the parallel system, e. g. processor, that will execute the statement instance. The dimensionality of the schedule space should not necessarily correspond to the dimensionality of the iteration space. Dates may be multidimensional, in which case the statement instances are executed following the *lexicographical order* of their execution dates.

↩ Scheduling relation. Scheduling relation is also referred to as mapping relation.

Multidimensional points are ordered according to the first dimension and, in case of ties, according to the following dimensions, e. g. hours and minutes are ordered lexicographically.

The *identity scheduling* relation simply maps each iteration domain dimension to a date dimension making the statement instances execute in the same order as the original loops. For the polynomial multiplication example from Listing 1, (5) is the identity scheduling relation. As see from the Figure 20, all the instances for $i = 0$ are executed before any of the instances for $i = 1$ following the lexicographical order.

$$\theta_S(N, M) = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \rightarrow \begin{pmatrix} t_1 \\ t_2 \end{pmatrix} \middle| \begin{array}{l} t_1 = i \\ t_2 = j \end{array} \right\} \tag{5}$$

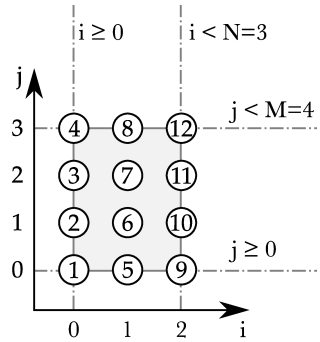


Figure 20 – Trivial scheduling relation for the polynomial multiplication kernel. Numbers inside circles denote order of execution of the iterations.

In many cases, one may want to schedule different parts of the domain differently. To address this issue, one can use a *union of scheduling relations*. Each relation in the union features constraints of its applicability to a part of the iteration domain. For example, the identity scheduling of the polynomial multiplication kernel may be decomposed into a union (6), each part of which can be modified independently.

$$\theta_S(N, M) = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \rightarrow \begin{pmatrix} t_1 \\ t_2 \end{pmatrix} \middle| \begin{array}{l} t_1 = i \\ t_2 = j \\ i \leqslant N \end{array} \right\} \cup \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \rightarrow \begin{pmatrix} t_1 \\ t_2 \end{pmatrix} \middle| \begin{array}{l} t_1 = i \\ t_2 = j \\ i > N \end{array} \right\} \tag{6}$$

A *program transformation* in the polyhedral model corresponds to the modification of the scheduling relation for a statement or a group of statement. This change will modify the order in which the statement instances are executed.

▷ *Program transformation corresponds to the change in scheduling relation.*

For example, one may change the trivial scheduling relation for polynomial multiply kernel (5) by adding the input dimension j to the definition of the output dimension t_1 (7). This *transformed schedul-*

ing relation exploits the fact that the statement `S` features an array subscript `i+j` by making this subscript a fixed value for a given iteration of the new outer loop t_1.

$$\theta_S(N,M) = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to \begin{pmatrix} t_1 \\ t_2 \end{pmatrix} \middle| \begin{array}{l} t_1 = i \\ t_2 = j \end{array} \right\} \xrightarrow{\text{"add } +j \text{ to } t_1\text{"}}$$
$$\theta'_S(N,M) = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to \begin{pmatrix} t_1 \\ t_2 \end{pmatrix} \middle| \begin{array}{l} t_1 = i+j \\ t_2 = j \end{array} \right\} \quad (7)$$

In order to obtain the loop traversal order, we combine the iteration domain and the scheduling relations (8). This combination relies on the Generalized Change of Basis, a method based on an unpublished work by Le Verge [162] rediscovered independently by Bastoul [16]. Generally, it consists in introducing *output* dimensions (t_1 and t_2 in our example) that correspond to the transformed loops in the beginning and keeping *input* (domain) dimensions in the end. If the new dimensions are sufficient to establish the total lexicographical order, the domain dimensions are not necessary for scheduling and become *degenerate*, i. e. they take a fixed value that can be expressed as a function of the previous dimensions. In practice, *degenerate* dimensions correspond to nested loops that have a single iteration and can be removed [16].

Total order means that for any two points in the domain, the order is defined.

$$\mathcal{T}_S(N,M) = \left\{ \begin{pmatrix} t_1 \\ t_2 \\ i \\ j \end{pmatrix} \middle| \begin{array}{l} t_1 = i+j \\ t_2 = j \\ 0 \leqslant i < N \\ 0 \leqslant j < M \end{array} \right\} \quad (8)$$

Being able to express *output* dimensions as functions of *input* dimensions is not guaranteed for all scheduling relations. In fact, it requires the scheduling to be an *invertible function*. Scheduling invertibility requirement was a long-standing restriction of the polyhedral model [6] until the introduction of the Generalized Change of Basis. The latter relaxes invertibility requirement by keeping the input dimensions. For the non-invertible scheduling relations, Generalized Change of Basis will result in less *degenerate* dimensions, allowing to express, e. g., loop tiling in the scheduling relation.

Tiling is a common loop transformation that introduces a nested loop with a fixed number if iterations.

Figure 21 illustrates the Generalized Change of Basis for the polynomial multiply kernel. Given that j is a *degenerate* dimension with $j = t_2$, we depict them on the same axis. Because the transformed scheduling relation (7) is defined by an invertible function, the projection on (t_1, t_2) of the polyhedron defined by the combined domain-scheduling relation corresponds to the *transformed iteration domain* (9). Projection on the (i, j) gives the original iteration domain.

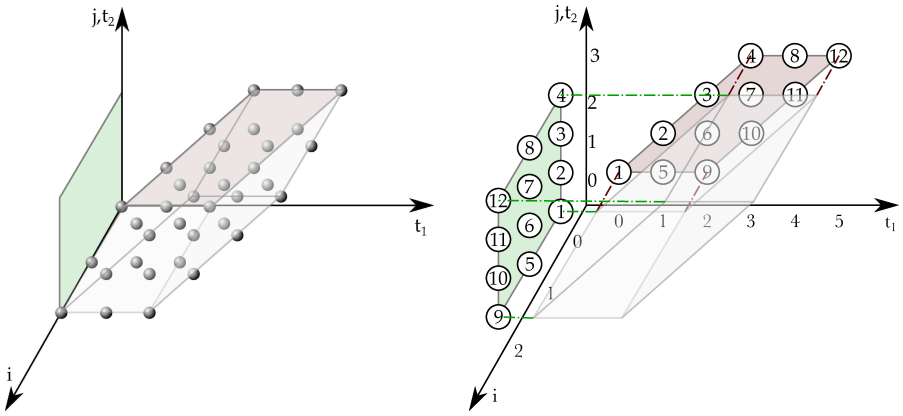


Figure 21 – Illustration of the Generalized Change of Basis: both the original (input) and the scheduling (output) dimensions are considered simultaneously. Dimensions j and t_2 are combined to remain in 3D. Projection on (i, j) (light green) corresponds to the iteration domain. Projection on (t_1, t_2) (dark red) corresponds to the transformed iteration domain given scheduling relation invertibility.

$$\mathcal{D}'_S(N, M) = \left\{ \begin{pmatrix} t_1 \\ t_2 \end{pmatrix} \middle| \begin{array}{l} 0 \leqslant t_1 - t_2 < N \\ 0 \leqslant t_2 < M \end{array} \right\} \tag{9}$$

The transformed iteration domain can be visually represented as a parallelogram, a skewed version of the original rectangle, as shown in the Figure 22. Left part of the Figure shows how bounds are expressed for new loop iterators and includes the numbers featuring original execution order within the points. Right part of the image shows the new execution order by means of numbers inside points.

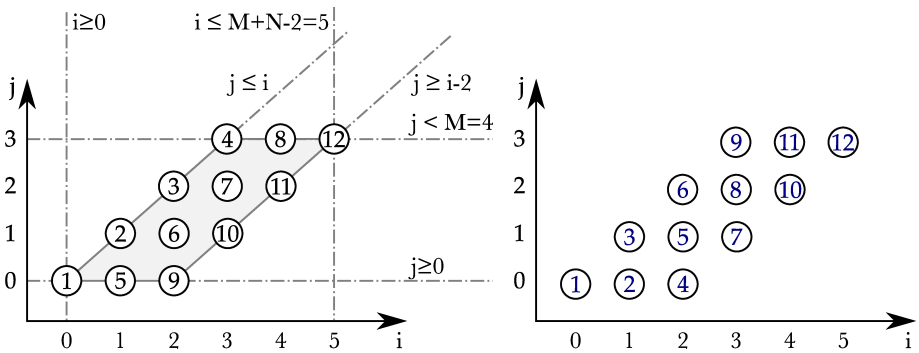


Figure 22 – Transformed domain of the polynomial multiplication kernel. Numbers inside points correspond to the original execution order on the left and to the new execution order on the right.

This transformed iteration domain corresponds to the code in Listing 3. Thanks to the change in the loop traversal order, iterations of the outer loop became independent of each other and can be exe-

cuted in parallel. Loops that correspond to the *degenerate* dimensions are commented out as they iterate only once and can be optimized away.

```
#pragma omp parallel for private(t2)
for (t1 = 0; t1 <= N+M-2; t1++)
  for (t2 = max(0, t1-N+1); t2 <= min(t1, M-1); t2++)
    // for (i = t1; i <= t1; i++)
      // for (j = t2; j <= t2; j++)
S:      C[t1] += A[t2] * B[t1-t2];
```

Listing 3 – C code of the transformed polynomial multiplication kernel featuring a parallel loop. Single-iteration loops corresponding to degenerate dimensions are commented out.

Throughout this document, we refer to the relation θ_S as *scheduling relation*, or *scheduling* for short. We call the set of scheduling relation unions for the entire polyhedral program part a *SCoP schedule*. Finally, we refer to the set of execution dates obtained by applying a scheduling to a particular domain (or a point in the domain) as *schedule*.

3.3.3 *Representing Memory Accesses*

In order to allow precise analysis, the polyhedral model represents memory accesses form multidimensional arrays with subscripts expressed as linear forms of loop iterators and parameters. Each memory access in a statement is represented by an *access relation* that maps points of the iteration domain to the array identifier and the multidimensional index that is accessed in this array. Access relations are defined separately for *reading* from memory and *writing* into it. The relation is annotated by the type of access.

For example, the polynomial multiplication kernel in Listing 1 comprises one statement featuring four memory accesses in one execution: read and write access to `C`, and read accesses to `A` and `B`. The read from `C` access relation is shown in Equation (10). Other access relations may be expressed similarly.

$$\mathcal{A}_{S,C,\text{read}}(N, M) = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to \begin{pmatrix} a_1 \\ \text{id} \end{pmatrix} \middle| \begin{array}{l} a_1 = i + j \\ \text{id} = C \end{array} \right\} \tag{10}$$

3.4 DEPENDENCE ANALYSIS AND PARALLELISM

The exact instance-wise nature of the polyhedral model allows to analyze the dependences between individual statement instances and identify, e. g., loop-carried dependences that prevent parallelization.

COMPUTING DEPENDENCES Exact instance-wise dependences may be computed in the union of relations representation by combining iteration domains and access relations of multiple statements in a single *dependence relation* [18]. This relation encodes three conditions:

existence — instances belong to the iteration domains of the respective statements;

conflict — instances access the same element of the same array and at least one of the accesses writes into that element;

causality — one instance, called dependence *source*, is executed before another instance, called dependence *target*.

simultaneous reads from the same memory create no concurrency conflict

If any of the polyhedral description components are unions of relations, the dependence may also be described as a union of relations. Similarly, the dependence union is created in case of multiple accesses to the same array in one statement. The dependence relation maps a point in the iteration space of one statement to all the points in the iteration space of another (or the same) statement that depend on it. Parametric integer programming solvers allow to find the values of parameters so that no dependence exist between statements and they can be safely executed in parallel.

For example, the polynomial multiplication kernel has three dependences caused by `+=` operation on elements of `C` array. These dependences only differ by their type: read after write (*flow* dependence), write after read (*anti* dependence) and write after write (*output* dependence). In the relational form of one of this dependences (11), first two inequations present the existence condition, then three equations encode the conflict condition and the final inequation is the causality condition.

$$\Delta^{flow}_{S,C\to C}(N,M) = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to \begin{pmatrix} i' \\ j' \end{pmatrix} \middle| \begin{array}{l} 0 \leqslant i, i' < N \\ 0 \leqslant j, j' < M \\ a_1 = i + j \\ a'_1 = i' + j' \\ a_1 = a'_1 \\ i' - i \geqslant 1 \end{array} \right\} \tag{11}$$

COMPUTING VIOLATED DEPENDENCES As the transformation in the polyhedral model corresponds to a change in the scheduling relation union, it may change the order of execution so that a pair of dependent instances is executed in the opposite order, provoking a *dependence violation*. Given two schedulings, it is possible to compute the exact set of instance-wise dependences that are violated by the new scheduling [18]. Again, the approach consists in creating a *violated dependence relation* by combining iterations domains, reference

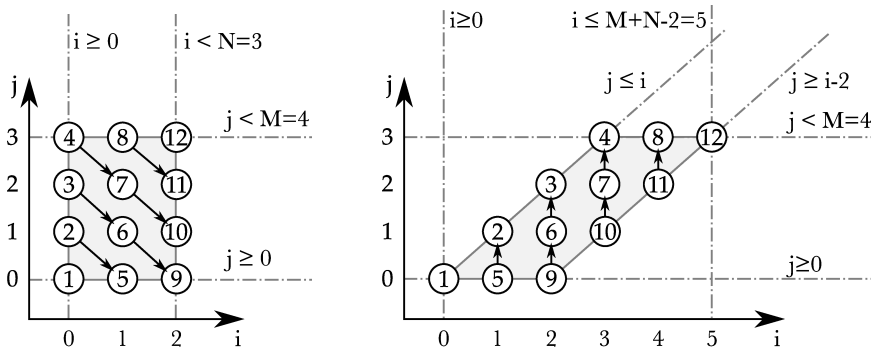


Figure 23 – Arrows show which instances in the iteration domain depend on which previously executed instances.

and new schedulings relations and access relations using the three conditions:

dependency existence — instances belong to the iteration domains of respective statements and access the same element of the same array and at least one access is a write;

scheduling — with the reference scheduling, the source instance is executed before the target instance;

causality violation — with the new scheduling, the source instance is executed at the same time as or after the target instance.

For the polynomial multiplication kernel (3), when performing the transformation (7), i. e. changing $t_1 = i$ to $t_1 = i + j$, the violated dependence is expressed by Equation (12). We denote t_1, t_2 the output dimensions of the reference scheduling relation and t'_1, t'_2 the output dimensions of the new scheduling relation, domain and access dimensions remain common.

$$\Gamma_{S,C\to C}(N,M) = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to \begin{pmatrix} i' \\ j' \end{pmatrix} \middle| \begin{array}{l} 0 \leqslant i, i' < N \\ 0 \leqslant j, j' < M \\ a_1 = i + j \\ a'_1 = i' + j' \\ a_1 = a'_1 \\ t_1 = i \\ t_2 = t'_2 = j \\ t'_1 = i' + j' \\ t'_1 - t_1 \geqslant 1 \end{array} \right\} \tag{12}$$

Using parametric integer programming [93], one can demonstrate that this relation is *not* empty for positive parameter values. Which means that this transformation is not legal and the transformed program may yield incorrect result. In our example, the transformation violates the dependence and therefore is *illegal*. Figure 23 allows to

observe this as the direction of the dependence arrows change after the transformation.

Violated dependence relation can also become a union if any of these conditions are expressed by a union of relations. This relation allows to ensure *legality* of the transformation in the polyhedral model, i. e. to guarantee that the transformed program yields the same result as the original one.

EXPRESSING PARALLELISM As the polyhedral model was initially designed for loop parallelization or parallel hardware synthesis, it allows to expose all instance-level and statement-level parallelism available in the static control parts of the program. However, this parallelism is not always trivially exploitable by existing parallel constructs in programming languages and may ultimately lead to a slowdown due to the explosion of the control and synchronization [12, 247].

Assigning equal execution dates to multiple statement instances is the seemingly simplest way of expressing parallelism in the polyhedral scheduling relation. However, in the relational form, it can only express parallelism in the most nested dimensions as first dimensions serve to provide a partial lexicographical order. Furthermore, identical dates allow for ambiguous interpretation: they may *require* instances with identical dates to be executed at the same time, or may *allow* them to be executed in an arbitrary order including in parallel. For example, CLooG uses the second, permissive interpretation, but does not allow to enforce simultaneous execution.

Therefore, we propose to express parallelism by adding independence or parallelism semantics to the scheduling relation dimensions as described below in Section 3.5.3.

3.5 STRUCTURE OF THE POLYHEDRAL SCHEDULING RELATION

3.5.1 *Existing Scheduling Structures*

In the general case, a scheduling relation union may take an arbitrary form as long as it has a finite non-parametric number of elements defined by affine (in)equations. However, most approaches to program transformation in the polyhedral model impose a certain structure on the scheduling relation to simplify its modification and create schedule semantics better visible for the user.

UNSTRUCTURED SCHEDULINGS Although they give the broadest possibilities for modifying the execution order of the statement instances, unstructured schedulings are hard to manipulate from a high-level perspective.

D+1 STRUCTURE This structure maps each loop in the nest to a particular dimension in the polyhedral scheduling thus capturing the shape of the source code. A loop lexically following another loop at the same depth in the nest is scheduled to start after the last iteration of the previous loop, e. g. if a loop has N iterations, the following will be scheduled starting at $N + 1$. The last dimension is required to represent the order of statements within the loop nest. Given maximum d loops in the nest and an extra dimension for the lexical order, the structure uses $d + 1$ dimensions, hence its name. It was widely used thanks to its simplicity and closeness to the code representation, making it easier for human user to operate. However, it requires encoding part of the iteration domain information, namely the number of loop iterations, in the scheduling, which prevents from manipulating it independently of the domain.

2D+1 STRUCTURE To decouple the scheduling information from the iteration domain, the $2d + 1$ structure relies on adding auxiliary dimensions for each loop depth in the loop nest: given d loops, it uses one auxiliary and one loop iteration dimensions as well as one dimension for the order of statements in the most nested loop. Iteration and auxiliary dimensions, which we call α and β dimensions, respectively, are alternating in the scheduling relation. The role of β-dimensions is similar to the role of the last dimension in the $d + 1$ structure — establish the order in which loops or statements are executed at a certain depth. Separating the order information into specific dimensions allows for creating domain-agnostic schedulings since it is no more necessary to know the number of the iterations available in the domain. β-dimensions are typically expressed as non-parametric constants. This choice allows to identify a particular loop or statement by a sequence of constant values assigned to each of the auxiliary dimensions making this structure closer to the code representation and easier to manipulate on loop-wise level.

PLUTO PLOOP TREES Polyhedral program optimizer Pluto uses a tree-like structure for the schedulings in addition to a variant of $d + 1$ structure [37] [5]. In this tree, leafs correspond to individual statements and nodes correspond to loops containing statements. On one hand, this allows to simplify loop-level manipulations, such as tiling for vectorization or loop parallelization. On another hand, this lowers the computational complexity of the scheduling algorithm as multiple loops and statements can share a partial scheduling encoded in the tree nodes.

5. as presented in the Spring School Maths-Info-HPC, May 9-13, 2016, St Germain au Mont d'Or, France

SCHEDULE TREES The latest structure for the polyhedral schedulings comes from the observation that optimizers, including Pluto and `isl`, operate on trees while other scheduling structures are built to encode the tree structure of the loop nest in the scheduling implicitly [256]. Indeed, values of β-dimensions in the $2d+1$ structure can be structured as a tree reflecting the nesting and the order of the loops. Schedule trees representation makes this shape explicit, allowing nodes that represent sequential or parallel execution that are not necessarily mapped to the loops. These nodes may contain partial schedules for their leafs corresponding to statements. Tree structure makes it simpler to schedule a particular loop, statement or even part of the statement iteration domain. At the same time, it reduces the uniformity of representation requiring a tree traversal to reconstruct a system of inequations fully defining the scheduling of the statement.

3.5.2 *Scheduling Structure Selection*

▷ *Combination of $2d+1$ structure and union of relations form.*

We combine the union of relations form with the $2d+1$ structure: each relation in the union has $2d+1$ structure and defines a convex multidimensional polyhedron. Different parts of the union are allowed to have different dimensionality as long as the scheduling is *compatible* with the iteration domain, i. e. has the same number of the *input* dimensions, related to the domain, whereas the number of the *output* dimensions may vary. In section 3.5.3, we propose an extendable dimension typology that includes not only the type of dimension (iteration α or auxiliary β), but also a set of flags related to the scheduling of the statement instances along this dimension, e. g. independence, parallelism or unrolling. In section 3.5.6, we demonstrate how a tree structure may be recovered from this structure. $2d+1$ structure alone does not guarantee that the scheduling may be completely decoupled from the iteration domain, in the section 3.5.8, we provide conditions of *global validity* of the scheduling that ensure a scheduling is applicable to any domain of the compatible dimensionality without previously discussed negative effects. The combination of these properties allows to use our $2d+1$-union representation for building a precise and expressive program transformation engine in the polyhedral model. This engine is based exclusively on changes to the scheduling relations as opposed to previous approaches, such as CHILL [58] or URUK [104], that may modify iteration domains.

3.5.3 *Dimension Semantics*

We assign to each dimension of the scheduling relation a flag or a combination of flags that define its higher-level semantics. Although this semantics may improve the modeling, it remains optional and the relation itself fully represents the scheduling. Our dimension

semantics is based on the $2d + 1$ structure, but it may be applicable to other structures. We propose the following semantics.

LOOPS Each even output dimension, called α-dimension, corresponds to a loop iteration variable, as in the basic polyhedral representation.

$\hookleftarrow$ α and β dimensions.

LEXICAL ORDER Each odd output dimension, called β-dimension, in the scheduling relation is a constant representing the order of statements in the loop of corresponding depth. Zero loop depth corresponds to the statements and loops at the root of the SCoP, e.g. at a function level, that are not enclosed by any loop.

In previous work, these β-dimensions are also called auxiliary.

INDEPENDENCE Both α and β-dimensions can be declared independent meaning that they do not carry a dependence, i. e. that the any permutation in this dimension for the current statement is legal. Essentially, dimension independence means that the actual values of this dimension can be ignored when selecting the order of execution. For the sake of consistency with non-semantic relations, these values are set to identical values. Independent α-dimensions can be expressed as parallel loops while independent β-dimensions may correspond to parallel threads or tasks. If multiple statements share a loop, the corresponding dimension should be marked as parallel for *all* of these statements in order to produce a parallel loop.

STRIP-MINING Partitioning the loop into smaller blocks is a widely used technique enabling, e. g., vectorization [265]. The particularity of this transformation is that it creates a new dimension in the scheduling relation, and this dimension depends on another dimension requiring division and modulo operations for its definition. For integer scheduling relations, it can be modeled by a pair of inequations [16]. Marking dimension as strip-mined allows to treat it separately when analyzing the validity of the scheduling (see section 3.5.8) on one hand and to perform vectorization on another hand.

UNROLL Loop unrolling that replaces individual iterations by new statements cannot be represented in a polyhedral scheduling as this scheduling is defined for each individual statement. Nevertheless, this syntactic transformation is often used alongside polyhedral engine and is included into high-level transformation engines such as CHiLL [58]. Marking an α-dimension as unrolled allows to treat this dimension separately in the analysis and to generate the corresponding code with multiple statements when requested.

3.5.4 *Scheduling Relation Equality and Equivalence*

Since the scheduling relation establishes the *order* of execution of the iteration domain points, the actual numerical values of execution dates may vary as long as the order is preserved.

EQUALITY Two scheduling relations are considered *equal* if they yield numerically equal logical execution dates. Equal scheduling relations arise, in relational form, due to description of the scheduling as linear (in)equation-bounded relations: operations such as multiplication by a constant factor or per-element addition do not affect systems of linear (in)equations. For example, two scheduling relations may be equal even though the equations defining the relation are different (13). The linear system of the second scheduling relation can be obtained by adding the second equation to the first one, which does not affect the solution of the system.

$$\left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to \begin{pmatrix} t_1 \\ t_2 \end{pmatrix} \middle| \begin{array}{l} t_1 = i \\ t_2 = j \end{array} \right\} = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to \begin{pmatrix} t_1 \\ t_2 \end{pmatrix} \middle| \begin{array}{l} t_1 + t_2 = i + j \\ t_2 = j \end{array} \right\} \tag{13}$$

A change in any scheduling relation union that results in an equal scheduling relation is called an *invariant transformation*. Any algorithm operating within the polyhedral representation is free to perform *invariant* transformations for its needs since they do not affect the resulting schedule in any way. Two scheduling relations can be compared for equality by first defining a normal form of the constraints that define it and then directly comparing the constraints.

EQUIVALENCE Two scheduling relations are considered *equivalent* if they yield identical execution orders irrespective of the numerical values of the execution dates. All equal scheduling relation pairs are also equivalent. Only the entire SCoP schedulings should be compared for equivalence: two scheduling relations may be equivalent if taken separately, but lose the equivalence once combined with another scheduling relation. For example, taken alone, two scheduling relations θ_1 and θ'_1 (14) are equivalent as the order of execution of any domain remains the same due to their monotonicity. However, if the SCoP scheduling includes two statements with scheduling relations θ_1 and θ_2 (15), essentially alternating the instances of two statements, the transformed SCoP scheduling with relations θ'_1 and θ_2 does not respect the initial order of instance execution with respect to each other. Furthermore, scheduling relations may be equivalent for some domains, but not for others, e. g. if they are defined as unions with

different conditions for different parts of the domain. These cases are analyzed in Section 3.5.8.

$$\theta_1 = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to \begin{pmatrix} t_1 \\ t_2 \end{pmatrix} \middle| \begin{array}{l} t_1 = i \\ t_2 = 2j \end{array} \right\}; \theta'_1 = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to \begin{pmatrix} t_1 \\ t_2 \end{pmatrix} \middle| \begin{array}{l} t_1 = i \\ t_2 = 3j \end{array} \right\} \tag{14}$$

$$\theta_2 = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to \begin{pmatrix} t_1 \\ t_2 \end{pmatrix} \middle| \begin{array}{l} t_1 = i \\ t_2 = 2j + 1 \end{array} \right\} \tag{15}$$

A change to the scheduling relation union that results in an equivalent scheduling relation is called *equivalent transformation*. A demonstration of SCoP scheduling equivalence for any set of domains that may be scheduled requires considering all the scheduling relation unions simultaneously.

3.5.5 *Scheduling Normalization*

In order to compare scheduling relations for equality, we need to provide a normalized form of the scheduling relation. We start by replacing all strict inequations by their non-strict alternatives Then, we replace pairs of complementary non-strict inequations $z \geqslant f()$ and $z \leqslant f()$ by equations $z = f()$. We separate the set of constraints defining the relation into a set of equations and a set of inequations. In the following, we consider a set of equations as a system of affine equations with symbolic dimensions as unknowns. We define two normalized forms, *input* form and *output* form depending on the shape of the linear system.

In integers, $z > 0 \Leftrightarrow z \geqslant 1$.

↩ Normalized forms of scheduling relations.

INPUT FORM The input form is obtained by symbolically solving the linear system for *input* dimensions in the integer numbers treating output dimensions as symbolic parameters. For example, the input form of the transformed scheduling relation (7) for polynomial multiply kernel is

$$\theta_S^{in} = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to \begin{pmatrix} t_1 \\ t_2 \end{pmatrix} \middle| \begin{array}{l} i = t_1 - t_2 \\ j = t_2 \end{array} \right\} \tag{16}$$

Note that the solution in integers is only possible if the square matrix formed from the coefficients for *output* dimensions is *unimodular*, i. e. has a determinant 1. In the other cases, we allow non-unit coefficients for input dimensions, as in $2i = t_1$. However, we require these coefficients to be minimal positive values. Input form can be obtained by further representing the linear system as a matrix and running a integer variant of LU-decomposition.

We also substitute *input* dimensions in inequations by their expressions in terms of output dimensions obtained by solving the linear system. If the equation has a non-unit coefficient for the input dimension, we multiply the inequation by this (positive) coefficient before substitution. We then divide each inequation by the largest common divisor of its coefficients. Finally, we remove redundant inequations, i. e. $x \geqslant 0$ is redundant in presence of $x \geqslant 3$.

OUTPUT FORM Similarly, the output form is obtained by symbolically solving the linear system for *output* dimensions in the integer numbers treating input dimensions as symbolic parameters. Most scheduling relations presented before are in the output form, including the transformed scheduling relation for polynomial multiplication kernel

$$\theta_S^{\text{out}} = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to \begin{pmatrix} t_1 \\ t_2 \end{pmatrix} \middle| \begin{array}{l} t_1 = i + j \\ t_2 = j \end{array} \right\}$$

Unimodularity requirements now apply to the matrix formed of the coefficients for *input* dimensions. We also perform output dimension substitution in the inequations in the similar way to the input form.

EXPLICITLY, IMPLICITLY-DEFINED AND DEGENERATE DIMENSIONS Given the separation between equations and inequations, we differentiate the *output* dimensions in the *output* form as follows.

An output dimension is *explicitly-defined* if it can be expressed by an equation that does not involve other output dimensions. An output dimension is *degenerate* if it can *only* be expressed by an equation involving other output dimension. Finally, an output dimension is *implicitly-defined* if it is expressed by a system of inequations, e. g. $t_1 \geqslant 3t_2 \wedge t_1 \leqslant 3t_2 + 2$.

3.5.6 *Exposing Lexical Order in Scheduling Relations*

Using the $2d+1$ representation allows to expose the lexical order of statements and their nesting in loops in the scheduling relations. This allows to target a particular statement or a loop nest with a scheduling change on one hand, and simplifies the reasoning on scheduling properties, including equality and equivalence, on another hand.

The $2d+1$ representation alternates β dimensions that represent the lexical order and α dimensions that correspond to loop iteration variables. For example, the transformed polynomial multiplication

scheduling relation, originally with two output dimensions (7), now features five output dimensions, including three β dimensions (17).

$$\theta''_S(N,M) = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to \begin{pmatrix} \beta_1 \\ \alpha_1 \\ \beta_2 \\ \alpha_2 \\ \beta_3 \end{pmatrix} \middle| \begin{array}{l} \beta_1 = 0 \\ \alpha_1 = i + j \\ \beta_2 = 0 \\ \alpha_2 = j \\ \beta_3 = 0 \end{array} \right\} \tag{17}$$

As the statement instances are executed in lexicographical order, i. e. the latter dimensions are considered in the order when the former dimensions are equal, including an extra dimension between any pair of loop-related α dimensions allows to fully separate statements from each other in the schedule. These β dimensions are non-parametric constants. β_1 encodes the lexical order of the outer loops in the SCoP and individual statements outside any loops. Similarly, β_i encodes the lexical position of the objects iterating over α_i, or, alternatively, the order of objects in $(i-1)$-th loop. The number of β-dimensions corresponds to the number of the loops enclosing the given statement plus one.

It also allows to start numbering of both α and β dimensions from 1.

As the β dimensions are constant, they can be written in a more concise form for any scheduling relation. A β-*vector* or beta-vector is an ordered list of constant values of β-dimensions. For the running example of polynomial multiplication kernel (17), the β-vector is $(0,0,0)$. Beta-vectors completely represent the lexical order of statements and loops. A given β-dimension corresponds to a given loop enclosing a statement and holds the position of this loop in its enclosing block. Therefore, statement and loops that belong to the same outer loop will share a β-*prefix*, a prefix of the β-vector. For a more elaborate example using the conventional beamforming kernel in Listing 4, β-prefixes and β-vectors are provided as comments. Note that the "root" of the SCoP has an empty β-prefix, meaning that it encloses anything else in the code.

```
                                                  // ()
t = 0;                                              // (0)
t_val = DBL_MIN;                                    // (1)
for (i = 0; i < N; i++) {                           // (2)
  a_i[i] = 0;                                         // (2,0)
  a_r[i] = 0;                                         // (2,1)
  for (j = 0; j < M; j++) {                           // (2,2)
    a_r[i] += s_r[j]*m_r[i][j] - s_i[j]*m_i[i][j];   // (2,2,0)
    a_i[i] += s_i[j]*m_r[i][j] + s_r[j]*m_i[i][j];   // (2,2,1)
  }
  val = a_r[i]*a_r[i] + a_i[i]*a_i[i];                // (2,3)
  t = (val >= t_val)? (t_val = val, i) : t;           // (2,4)
}
```

Listing 4 – Conventional Beamforming kernel

In a sequential program, scheduling relations have unique β-vectors as they represent not only the lexical position, but also the statement-wise order of execution and two statements cannot be lexically at the same position, nor executed at the same time. A β-vector cannot be a prefix of another β-vector for the same reason. Beta-vectors form a tree where each leaf corresponds to a statement, and each node represents a loop. This tree can be constructed from the set of β-vectors using the algorithm on Figure 25. The tree of β-vectors for the Listing 4 is shown in the Figure 24. In a parallel program with statement-level parallelism, identical β-vectors may be used to describe parallelism. However, it would break the semantics of the lexical order. Therefore, we represent statement-level parallelism by adding *parallel* semantics to the β dimension.

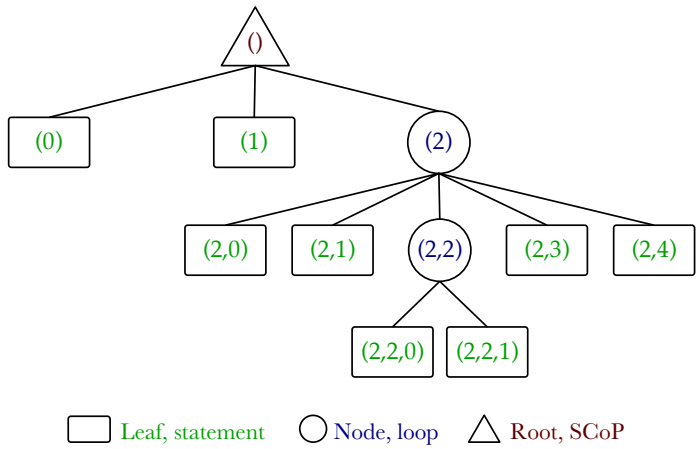


Figure 24 – Beta-tree of the conventional beamforming kernel.

One statement has as much β-vectors as it has relations in its scheduling union. Although it is a single statement, parts of its domain will be scheduled differently, possibly in different enclosing loops or branches leading to the multiple *occurrences* of the statement. Each occurrence may have its own lexical position. Therefore, on a general rule, each relation in the union has a different β-vector encoding the lexical position of the corresponding occurrence.

3.5.7 *Lexical Order Normalization*

Like all *output* dimensions, β-dimensions define the order of execution rather than exact statement positions. For the sake of simplicity, β-vectors can be brought to the *normal form* with minimal non-negative values on each dimension that preserve the order. For example, the normal form of the β-vector set $\{(0,1),(2,1)\}$ is $\{(0,0),(1,0)\}$. Once the beta-tree is built, the algorithm from Figure 26 allows to update β-vectors and prefixes so that they are normalized. The β-*normalization* is an *equivalent* transformation that does not affect other dimensions.

```
Data : β-tree defined is a directed graph G = (𝒱, ℰ, root) defined
       by the set of nodes 𝒱 and, set of edges ℰ and the root node
𝒱 ← root ;
foreach statement beta-vector β⃗ do
    for i = 1 to dim β⃗ do
        let ρ⃗ be the prefix ρ⃗ = β⃗_{1..i} ;
        𝒱 ← 𝒱 ∪ ρ⃗ ;
        if dim ρ⃗ = 1 then
            ℰ ← ℰ ∪ (root, ρ⃗);
        else
            let ρ⃗′ = ρ⃗_{1..(i−1)};
            ℰ ← ℰ ∪ (ρ⃗′, ρ⃗) ;
        end
    end
end
```

Figure 25 – Algorithm for beta-tree construction

```
Data : β-tree G = (𝒱, ℰ, root), node v
/* algorithm starts with v = root                                  */
select children of a node v, 𝒱′ ← {v′ ∈ 𝒱 : (v, v′) ∈ ℰ} ;
sort 𝒱′ by β-prefix ;
foreach v′ ∈ 𝒱′ do
    change β-prefix of v′ to (ρ_1, ..., ρ_{dim ρ⃗}, i) where ρ⃗ is the
      β-prefix of the parent node v and i is the index of the
      current node v′ in sorted 𝒱′ ;
    recurse with v = v′ ;
end
```

Figure 26 – Algorithm for beta-tree normalization

When the polyhedral analysis is applied to the syntactical representation, such as the source code or a syntax tree, we use a polyhedral raising tool, such as *Clan* [6], that generates normalized β-vectors for each scheduling relation.

3.5.8 *Scheduling Validity Conditions*

One of the principal benefits of our vision of the union of relations polyhedral representation is the *separation of domain from scheduling*. Programs may be transformed without modifying their iteration domain, only by the change of scheduling relation. Beside its elegance,

6. http://icps.u-strasbg.fr/~bastoul/development/clan/index.html

it makes scheduling relation manipulation easier and provides strong guarantees on the transformed program correctness if the schedule is *legal*, i. e. does not violate the dependences. Embedding the order information solely into the scheduling relation and leaving the iteration domain immutable allows also to reapply the precomputed transformation to domains of different shape and size and to delay the legality checks until after all modifications are complete, contrary to existing approaches that may require intermediate dependence checks [58].

However, the scheduling structure alone does not guarantee the *validity* of the scheduling relation. One statement instance may be scheduled for execution from zero to possibly unlimited number of times. The non-execution of an instance is likely to violate the original program semantics. Re-execution of a statement instance, although sometimes useful for avoiding communication and synchronization in parallel programs, may also violate semantics of the original program if this instance has a side effect, e.g. is accumulating values in a variable. To avoid these dangerous effects, the scheduling relation must guarantee the existence of a unique integer execution date for each point in any domain by respecting the following validity constraints.

EQUAL INPUT DIMENSIONALITY Since the scheduling relation establishes the correspondence between the iteration domain and the execution dates, the number of *input* dimensions for each relation in the scheduling union must correspond to the dimensionality of the iteration domain. We refer to the $(\dim \vec{\iota} \to \dim \vec{\sigma})$-dimensional scheduling relation applicable to d-dimensional domain, i. e. $d = \dim \vec{\iota}$ as d-*compatible*. All relations in the union are required to have identical *input* dimensionality as they are applicable to the same domain. On the other hand, output dimensionality may vary within the union. If a valid scheduling relation has more input dimension than the domain, it becomes partially *undefined*. If it has less input dimensions than the domain, part of the domain may be ignored violating other conditions.

SCHEDULE EXISTENCE Any *d-compatible* scheduling relation union must define a date for any d-dimensional integer point. An individual relation may include inequations that restrict its applicability to the part of the iteration domain as long as the whole union maps the entire integer space $\mathbb{Z}^d$ to the corresponding logical dates.

LOGICAL DATE INTEGRALITY All dates defined by the scheduling relation must be integer. In the *output form*, we require *explicit definitions* of the output dimensions to have a unit coefficient for the dimension itself and be linear functions with integer coefficients, i. e.

have the form $t_x = \vec{k}_\iota^T \vec{\sigma}$. This guarantees that the output dimension values are integer. Input dimension values are integer given that they come from a domain defined in $\mathbb{Z}^{\dim \vec{\iota}}$.

LOGICAL DATE UNIQUENESS Each instance in the d-dimensional iteration domain must be executed exactly once and its logical execution date should remain unique throughout the entire schedule of the program. Thanks to unique β-vectors in each relation, it is sufficient to ensure date uniqueness within an individual relation.

Combining *existence* and *uniqueness* constraints, we obtain that a scheduling relation should have at least $\dim \vec{\iota}$ explicitly defined (linearly independent) dimensions.

We call *globally valid* scheduling relation unions that respect all these conditions. Globally valid d-compatible schedulings allow to assign a unique integer date for any point in $\mathbb{Z}^d$. Scheduling relation unions that respect the conditions only for a subset of $\mathbb{Z}^d$ are called *conditionally valid*, where conditions define the subset. Finally, a scheduling relation union is called *invalid* if there is no point $\mathbb{Z}^d$ for which it respect all validity conditions.

↩ Global and conditional validity.

For example a one-dimensional scheduling that assigns execution dates only for the first ten iterations is conditionally valid since it fails to schedule all iterations of a larger domain (existence constraint). A relation with implicitly defined dimension bounded by two different explicitly defined dimensions is invalid since it assigns multiple dates for the same point (uniqueness constraint). Even though some polyhedral frameworks allow fractional *explicitly defined* dimensions, we consider them invalid (integrality constraint). We argue that they do not improve expressive power of the scheduling as an *equivalent* transformation that multiplies all fractional dates by a constant factor makes them integer.

3.6 DISCUSSION

STRUCTURED SCHEDULING TO IMPROVE MANIPULATION Imposing a structure on the scheduling relation union in the polyhedral representation allows to reason on a higher-level: rather than changing individual coefficients in the (in)equations, it is possible to identify specific statements, statement instances or groups thereof, and loops.

EQUALITY, EQUIVALENCE AND IDENTICAL SHAPE In their normalized form, equal scheduling relation unions are likely to have identical shape. However, this is not guaranteed. Consider, for example, a scheduling relation union consisting of two parts that are identical except a pair of complementary inequations ($i < 0$ and $i \geqslant 0$

respectively). Although these two scheduling relation unions yield equal dates for any compatible domain, i. e. they are equal by definition, they do not have the same shape. Specific equality and equivalence tests should be developed. We suggest that an equivalence test may be based on the dependence violation test. Namely, one may artificially create a dependence between a statement instance and its lexicographical predecessor. Dependence violation in this case will correspond to the change of instance order that makes two scheduling relations not equivalent.

SCHEDULING EQUIVALENCE AND PROGRAM EQUIVALENCE Scheduling relation equivalence leads us to the more general problem of program equivalence that is known to be undecidable in general case [133]. Using the data dependences, we can demonstrate that a change in the program *execution order*, or a transformation, leads to an equivalent program [30] However, it does not allow to reason more about program equivalence or polyhedral programs with different iteration domains or different statements that are equivalent to the given program.

RAISING AND LOWERING AS EQUIVALENT TRANSFORMATIONS Raising a program to the polyhedral model and lowering it back to the code should be considered similarly to program transformations: multiple equivalent representations of the program are possible within the polyhedral model and multiple codes may correspond to the same polyhedral representation. While the former case can be mitigated by the normalized structure of the polyhedral representation, the latter is entirely subject to the code generation algorithm. This unhandled equivalence gives code generators the flexibility necessary to adapt the code to the target system or to use more efficient generation algorithms. On the downside, formally proving the equivalence of the generated code to the polyhedral representation and, transitevely, to the original code remains an open problem.

SCHEDULING FUNCTIONS AND RELATIONS Earlier works on the polyhedral model used affine *functions* rather than relations or relation unions to perform scheduling and represent data accesses [6, 41, 77]. In addition to the homogenization of the representation, relations allow a wider range of possible schedules to be expressed in the model. Functions are, in fact, particular cases of relations that are left-total and right-unique (or functional), i. e. have a single output value for all possible input values. Relation-based scheduling remove these constraints. At the same time, our global validity conditions correspond to specific properties of relations: schedule existence implies that the relation is left-total and logical date uniqueness implies that it is injective and functional. Relations also help to express multidi-

mensional schedulings or non-unit strides thanks to the additional dimensions that can be added and quantified if necessary.

While it seems that a left-total injective functional relation is a more restricting than a function, in our representation, global validity should be respected by the entire relation *union* rather than each individual relation having all these properties. In particular, this allows for different dimensionalities of individual relations in the same union. Furthermore, thanks to the generalized change of basis method, requirements for relation bijectivity (or function invertibility) can be relaxed.

REDUCING COMPUTATIONAL COST WITH GLOBAL VALIDITY Operations on polyhedra, extensively used by the program analysis engine in the model, are notorious for their computational complexity [247]. This complexity is primarily caused by the combinatorial explosion due to pairwise analysis of the inequations that define the polyhedra (present in the iteration domain, scheduling and access relations). This complexity is alleviated by the fact that even large-scale programs, when written by human programmers, rarely have numerous conditions or deep loop nests as these constructs are also hard to handle by the programmer in the code representation.

Nevertheless, decreasing the number of costly computations in the polyhedral model remains important. Our union-of-relations representation allows to encode all the information about the execution in the scheduling relation union thus avoiding its combination with the domain relation union for the sake of intermediary correctness checks. Given that the scheduling global validity conditions are respected, one may operate exclusively on scheduling relations as systems of inequations avoiding computationally expensive polyhedral operations or dependence graph manipulations until the step.

MANIPULATING CONDITIONALLY VALID SCHEDULINGS Fully decoupling iteration domain from the scheduling relation may have its benefits, but domain information may be precious for constructing more efficient tailored schedules. For example, having loops with fixed number of iterations allows to perform full unroll transformation. Global validity constraints ensure that the scheduling relation union is applicable to any compatible domain. If a particular feature of the domain, e. g. its size or shape, may improve the scheduling, the conditions defining this feature may be included in the definition of compatibility. The scheduling will then have *conditional validity* that is subject to specific domain features.

A conditionally valid scheduling union can be easily turned into a globally valid: it suffices to include in it the relations with inequations complementary to the validity conditions. These extra relations may implement another, less optimal yet suitable in the general case,

scheduling. However, this operation will significantly increase the number of inequations defining the scheduling union and therefore its analysis time. Therefore, resorting to conditionally valid scheduling unions should be considered as a trade-off between the analysis time and the execution time of the generated program.

GLOBAL VALIDITY BEYOND 2D+1 REPRESENTATION Polyhedral scheduling relations expressed as relations with $2d+1$ structure support the definition and verification of global validity conditions. $d+1$ structure, on the contrary, relies on the domain properties to express the lexical order of the statements and loops: if one loop follows another at the same depth, its logical execution date is computed as a function of the number of iterations that the first loop does, which embeds a part of the domain information in the scheduling. Schedule trees provide a structure of the polyhedral model that combines both domains and scheduling rules. Such tree may contain partial domain information as a node, connected to partial scheduling information in its child nodes potentially followed by another part of the domain information and so on. While it reduces description redundancy, it does not guarantee that the scheduling information is separable from the domain information. Even though global validity conditions can be translated to the schedule tree form, verifying them requires considering the whole tree at once.

3.7 CONCLUSION

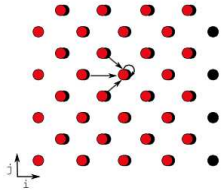
In this chapter, we presented the *polyhedral model*, the state-of-the-art program representation for loop-level modeling. Contrary to other program models, it represents individual loop executions and allows for substantial program transformations. We proposed a structure of the polyhedral representation, $2d+1$ union of relations, and a set of conditions, ensuring global validity, that enable efficient program transformation using the polyhedral model. First, it fully encodes the program and allows to reconstruct the code after transformation. Second, it provides information about individual loop executions and dependences between them that is not easily accessible in other program representations. Finally, thanks to our global validity conditions, it enables higher-level program manipulation through changes to scheduling relation union with verifiable safety.

However, before applying the program transformation on the statement instance level, the user has to understand and analyze the program with the same granularity. Using mathematical notation of polyhedra is challenging even for the experts and brings them far away from the code. Fortunately, the polyhedral model has a geometrical interpretation that can be used as a basis for an interactive program restructuring tool resorting to visualization for program analysis.

4 VISUALIZING POLYHEDRAL PROGRAMS

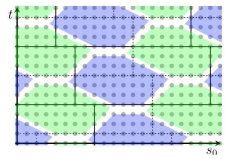
4.1 VISUALIZATION TECHNIQUES FOR POLYHEDRAL PROGRAMS

Programs amenable to the polyhedral model are often represented visually in the literature using various forms of scatter plots where points correspond to individual statement instances. This representation reifies these instances allowing to interact with them, contrary to the code that does not allow accessing individual statement instances. Furthermore, multiple terms in polyhedral program-optimization refer to the graphical nature of the representation suggesting that experts rely on visual representation and intuition to create, verify and evaluate program transformations. For example, loop *tiling* — a frequently used loop optimization technique that consists in introducing an extra loop with restricted number of iterations in the nest — is better explained visually as a tessellation of loop iteration space.

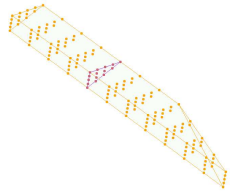


Scatterplot-like representation for the polyhedral construct `xfor`, reproduced from [92].

At the same time, points of the scatter plot may serve as nodes in node-link diagram where links correspond to dependences, ubiquitous for exploiting parallelism of the program. Therefore, several compilation-related polyhedral libraries include tools for visualizing polyhedra that describe programs, in particular iteration domains. PolyLib [167] includes a VisualPolylib[1] component for generating three-dimensional visualizations. The automatic loop parallelizer LooPo[2] [109] visualizes polyhedral domains and dependencies between points inside them.

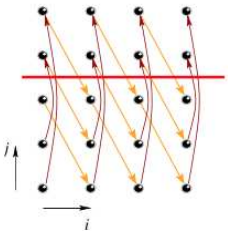


Hexagonal tiling, reproduced from [113].



VisualPolylib visualization.

Taken from a higher-level perspective, the polyhedral model uses a representation alternative to the code in order to reason about program behavior contrary to syntax tree-based approaches more tightly connected to the textual form of the program. While human programmers are more used and better trained to work with the code, alternative representations may be useful when the relevant information is not easily accessible in this form, which is the case with instance-wise data dependences and reuse. Existing polyhedral visualization techniques may be used as a basis for interactive program visualization. In order to design a representation according to the interactive program restructuring principles, we explore the parts of visual vocabulary commonly used by experts in polyhedral compilation and to which extent they are understandable by these experts. These observations allow us to design a visual representation for polyhedral programs that is understandable for the users and suitable for program



LooPo visualization.

1. https://icps.u-strasbg.fr/polylib/
2. http://www.infosun.fmi.uni-passau.de/cl/loopo/

manipulation. We propose several design alternatives and evaluate the visualization experimentally.

4.2 PARTICIPATORY DESIGN WORKSHOP

In participatory design, target audience members are participating in creating prototypes of the future interactive system.

We conducted a participatory design workshop in order to gather insights about the use of visual representation in communicating program parts in the polyhedral model. We were interested in finding both commonalities and divergences in visual representations. Our hypothesis was that experts have implicit rules and graphic elements for creating such representations in order to make them understandable to each other, but each of them uses particular parts of the representation to represent information specific to their interest.

4.2.1 *Protocol*

PARTICIPANTS Through the thematic mailing list, we recruited 4 participants (all male, aged 23-39) that have previously worked with the polyhedral model on both applicative programming and theoretical level. Each of participants used a polyhedral library in a software project and co-authored at least one paper on the subject. We divided them in two pairs. Participants in a pair did not previously work together on a visual representation. One of the experts in each pair reported that they have manually created a visual representation for their research project at least once.

PROCEDURE We held a 2-hour workshop with three activities focused on communicating program parts by means of polyhedral-based visualizations.

Activity 1 — creating the representation. Participants were asked to create a visualization for an abstract code part including at least one loop nest in order to communicate their execution to the other member of the pair. They were asked to include as many relevant details as necessary. They were not given any specific code snippets, but were allowed to create their own. The activity was being done on paper, with no computers or printed material available.

Activity 2 — interpreting the representation. Once visualization complete, participants in the pair exchanged their representations. They would then, in turn, explain how they understand each other's representation. The author of the representation was not allowed to correct his peer unless explicitly asked for clarification. After the end of explanation, the author could complete or correct it.

Activity 3 — improving the representation. For the final activity, participants were given a code snippet to represent visually (Listing 5) and asked to focus on communicating individual loop iterations, dependences between them and information relevant to parallelism. They

were asked to collaborate on this new version and provide a new representation that suits both of them. They were allowed to reuse their previous results or come up with a completely new one. This activity was also done on paper. Once completed, they explained it to the observer.

The code is intentionally useless so as to avoid participants suggesting algorithm-specific representations.

```
s = 0;
for (i = 0; i < N; i++) {
  a[i] = 0;
  for (j = i; j < M; j++) {
    for (k = 0; k < j; k += 2) {
      Z[i][j] += A[i][k] * B[k][j];
      Z[i][j] /= j / 2;
    }
    a[i] += Z[i][j] * s;
  }
  s = 0;
}
```

Listing 5 – Code snippet that participants had to visualize.

DATA COLLECTION We collected all artifacts produced by the participants including final versions and drafts. We photographed the artifacts as they were being made. The observer also took notes during the explanations and the discussion phases.

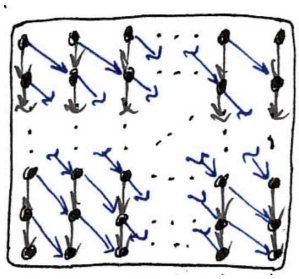
4.2.2 *Results and Discussion*

POINTS, SHAPES, ARROWS AND AXES All participants represented loops and iterations as polygons delimited by the lines that correspond to loop bounds. Three participants put points inside the polygons to represent individual iterations whereas the fourth one drew a lattice and shaded the polygon assuming grid intersection points correspond to individual executions. P4 created a 2-dimensional and a 3-dimensional representation, later explaining that he "*...would need to manipulate the 3D and take projections or sections*". All participants imagined loops with simple conditions: constant or depending on one outer iterator, resulting in rectangular and triangular shapes. No one assumed a loop nest with more than 3 loops.

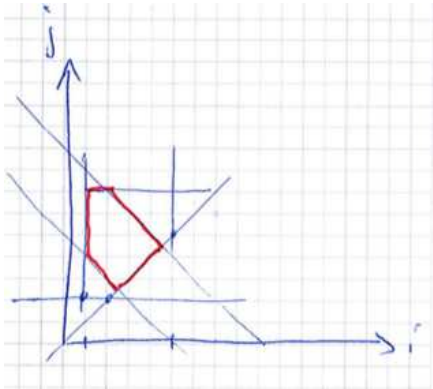
▷ Participants used polygonal shapes for loops and dots for instances.
We refer to participant 4 as P4.

Three participants (P1,P3,P4) used arrows to depict eventual dependences between iterations. Two of them used different colors to identify dependences or specify their properties. For example P1 referred to "*blue dependence*" and "*black dependence*". P2 decided to use arrows to represent the order of execution of the points. He commented "*with this matrix of points, you don't know if you go first by row or by column*".

All participants included axes in their representation. These axes formed a coordinate system for the polyhedral loop iteration domain.



(a) Using color coding to identify dependences.



(b) Intersecting shapes with grid support to show statement instances.

Figure 27 – Participants used points, shapes, arrows and axes in polyhedral representation.

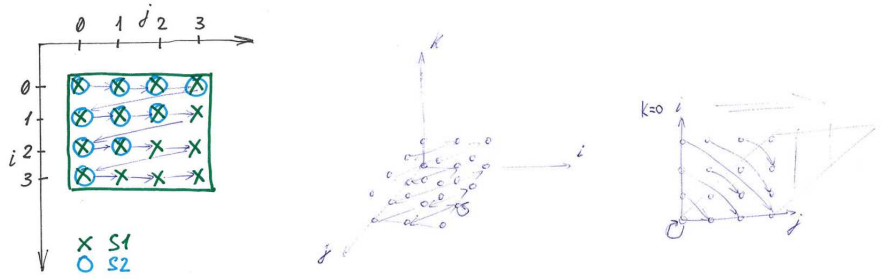
Each axis corresponded to a loop iterator. They put ticks on the axis and attached specific numerical values for the bounds. P3 depicted parametric upper bound by "breaking" the axis with dots and than continuing with a values $N-1$ and N.

MULTIPLE STATEMENTS AND NESTED LOOPS Only two of the participants (P1 and P2) had represented multiple statements in a loop nest. P1 created multiple coordinate systems, each of which hosted one polygon associated with one statement. He labeled this coordinate systems as "statement 1" and "statement 2". P2 used the same coordinate system for many statements and opted for color coding to tell them apart. In his representation, the distance between the points of different statements and different iterations was roughly the same. This was partly a reason to force him to use arrows to specify execution order.

P4 created a 3D representation for three nested loops and pair-wise projections of it. He commented later that "*projections should be enough in most cases, but 3D gives you better view of the domain size*".

When asked by the observer after the end of Activity 2 about multiple statements in the loop nest, P3 explained that he "*drew a common* [for all statements] *iteration domain*".

INTERPRETATIONS All participants unambiguously interpreted the axes as loop iterators, the polygonal shapes as loop bounds and the points as individual iterations. All of them also interpreted arrows as dependences, although it was not the case for P2's representation. P1 asked if P2 had "*everything depend on each other*" and received an explanation that the arrows depicted the order of execution. P2 than added that he "*thought about arrows for dependences, but the execution order of the statements was not clear*". He later suggested to have modes



(a) Using color coding to differentiate multiple statements.

(b) A 3D representation and one of its projections onto 2D.

Figure 28 – Participants used points, shapes, arrows and axes in polyhedral representation.

to switch between dependences and execution order, or introduce a special time axis to show the execution order.

DISAGREEMENTS Most disagreement came from the interpretation of the parts that are rarely relevant to present the research results, but necessary to have a complete representation of a program part. For example, P1 interpreted multiple coordinate systems of P2 as different loop nests rather than different statements inside the same nest. P3 labeled the bounds of polygon by the inequalities, e. g. $i \geqslant 0$ and $j \leqslant N$ explaining that "*for more complicated cases, it* [was] *more precise*". P4 disagreed and gave an example of a bound "*with multiple minimum functions*" like $\min(\min(\min(2*i+1,4),N-2),j-N+5)$.

▷ Participants agreed on general representation, and disagreed on minor details such as axis orientation.

Minor representation details also resulted in discussion, although were correctly interpreted by the participants. The vertical axis was pointing up for P1,P3,P4 motivated by "*it is how we do in mathematics*", and pointing down for P2 since "*it is similar to reading the code, you go down line by line*". Dependence arrows were directed from dependence sink to its source for P1 and P4. P1 explained it as "*the arrow shows that this* [iteration] *depends on that* [iteration]". On the other hand, P3 directed the arrows from source to sink motivating it as representation of "*how the data flows between iterations*". Execution order arrows used by P2 were also directed towards the next iteration, expressing flow of control.

The use of 3d for deep nests was also discussed. P2 suggested to "*differentiate between truly three-dimensional problems and loop nests that iteratively solve two-dimensional problems*". He justified that for iterative computations, it did not make sense to visualize the time loop, i. e. the one counting time steps, as computations performed on each step were identical. He further argued that it was impossible to optimize the time loop since each step was entirely dependent on the previous step. P4 proposed a similar distinction for "trivial" dimensions that

were not involved in dependences or for dimensions created during the program transformations such as tiling.

IMPROVEMENT Given a specific example, participants noticed the edge cases that they did not consider in their initial version, namely individual statements outside loops, imperfectly nested loops and non-unit loop strides. Both pairs kept the initial version with 2D representation as a basis. Due to the presence of non-unit strides they decided to use points rather than the intersection between a lattice and a polygonal shape. However, P3 and P4 kept the lattice as a substrate for placing and identifying points. P1 and P2 tried creating a 3D view, but found it too complex to identify boundaries and to put dependences. P4 drew a 3D shape, but was not sure about its correctness. Both pairs ended up with multiple projections of the loop nest on pair of dimensions. P1 and P2 used groups of stacked differently colored points to represent multiple statements. P3 and P4 used different marks (circle, rectangle, triangle, star, plus sign) to represent different statements. Both pairs depicted individual statements as a single mark outside coordinate systems.

4.3 THE CLINT VISUALIZATION

4.3.1 *Statement Instance Scatter Plot*

Based on the collected information, we created an interactive tool, *Clint*, for visualizing static control parts of the program. The visual representation is inspired by those typically used in the literature on the polyhedral compilation and ensures all program parts amenable to the polyhedral model can be visualized. It relies on the set of existing polyhedral tools and libraries to extract the polyhedral representation from the code. Beyond visualization, we intended to make the tool interactive, allowing to change the underlying program by changing its visual representation and having the code automatically regenerated.

Clint is a direct manipulation interface designed to:

— help programmers when parallelizing compute-intensive programs parts;
— ease the exploration of possible transformations; and
— guarantee the correctness of the final code.

Clint leverages the geometric nature of the polyhedral model by presenting code statements, their instances, and dependencies in a scatterplot-like visualization of iteration domains. The main components of its visualization are as follows.

↩ *Elements of Clint representation.*

POINT corresponds to an individual execution of a statement within the loop nest, i. e. a statement *instance*;

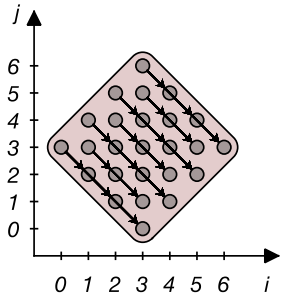
ARROW depicts a dependence between two statement instances, i. e. when both instances access exactly the same memory address and at least one of accesses writes to it;

POLYGON encloses all instances of a statement within the loop;

STACK OF POLYGONS represents the lexical order of the statements in the same loop;

AXIS corresponds to a loop iterator and its values;

COORDINATE SYSTEM corresponds to a 2D projection of a loop nest;




```
for (i = 0; i < N; i++)
  for (j = max(3-i,i-3);
       j <= min(i+3, 9-i); j++)
    Z[i+j] += f(i,j);
```

Figure 29 – Basic elements of *Clint* visualization: points represent individual iterations, arrows – dependences between them, axes correspond to loops.

Figure 29 gives an example of a loop nest with a single statement inside and its visual representation in *Clint*. The red shape delimits the iteration domain of the loop nest, the points correspond to individual executions and arrows depict the *data flow* between these executions. While the code snippet does not seem to feature parallelism, the visual representation of the data flow suggests, given parallel dependence lines, that some form of parallelism may be present. In fact, if we consider the order of statement instance execution, we notice that iterations of the inner loop (on `j`) inside a single iteration of the outer loop (on `i`) do not depend on each other, but rather on the previous iteration of the outer loop. Therefore, inner loop may be executed in parallel as long as the outer loop is sequential. This case is commonly referred to as *wavefront parallelism*.

A larger example is shown in Figure 30 that illustrates the code from Listing 6. Statements `S7,S8` share two loops and therefore visually share two axes `i` and `j` (a). Their stacking corresponds to the execution order with `S7` being topmost. Statements `S6` and `S7` share only the outer loop, but are nested in different inner loops, which corresponds to different coordinate systems aligned vertically so that the horizontal axis `i` is common (b). Statements `S4,S5` are nested in one loop only and, therefore, do not have a second axis (c). At the same time, they are placed in the same column and aligned to `i` values. Individual statements `S9` and `S10` are depicted without axes as they are not nested in loops (d). Finally, the statement `S11` with its own loop nest is visualized in a separate coordinate system that is not aligned

with the first column visually creating a different reference for the i loop (e).

```
for (i = 0; i < N; i++) {     // (0)
  for (j = 0; j < N; j++) {   // (0,0)
    S1(i,j);                  // (0,0,0)
    S2(i,j);                  // (0,0,1)
    S3(i,j);                  // (0,0,2)
  }
  S4(i);                      // (0,1)
  S5(i);                      // (0,2)
  for (j = 0; j < N; j++) {   // (0,3)
    S6(i,j);                  // (0,3,0)
  }
  for (j = 0; j < N; j++) {   // (0,4)
    S7(i,j);                  // (0,4,0)
    S8(i,j);                  // (0,4,1)
  }
}
S9;                           // (1)
S10;                          // (2)
for (i = 0; i < N; i++) {     // (3)
  for (j = 0; j < N; j++) {   // (3,0)
    S11(i,j);                 // (3,0,0)
  }
}
```

Listing 6 – A code snippet with multiple imperfectly nested loops and multiple statements.

↩ Representing order of statement instance execution in Clint.

The general order of statement instance execution is:

— stacked points, front to back;
— columns of points, bottom to top, crossing polygon boundaries and vertical boundaries of the coordinate systems;
— rows, left to right, crossing polygon boundaries, but not the coordinate system boundaries;
— vertical piles of coordinate systems.

For example, the first point inside the light red polygon representing `S4(0)` is executed after the topmost leftmost point of the green polygon representing `S3(0,5)`. The single point in light green polygon representing `S9` is executed after the topmost rightmost of the column on its left (`S8(5,5)`). *Clint* provides an animation that highlights each point in turn according to the execution order.

Statements with iteration domains featuring more than two dimensions are split up into two-dimensional projections and displayed as a scatter-plot matrix as shown in Fig. 31. If several statements have the same coordinates in a projection — e.g., $(2, 0, 0)$ and $(2, 0, 1)$ in the $\langle i, j \rangle$ projection — only one point is displayed and the intensity of its shade is proportional to the number of the underlying points.

Beyond participatory design feedback where participants had difficulties creating and interpreting three-dimensional representations,

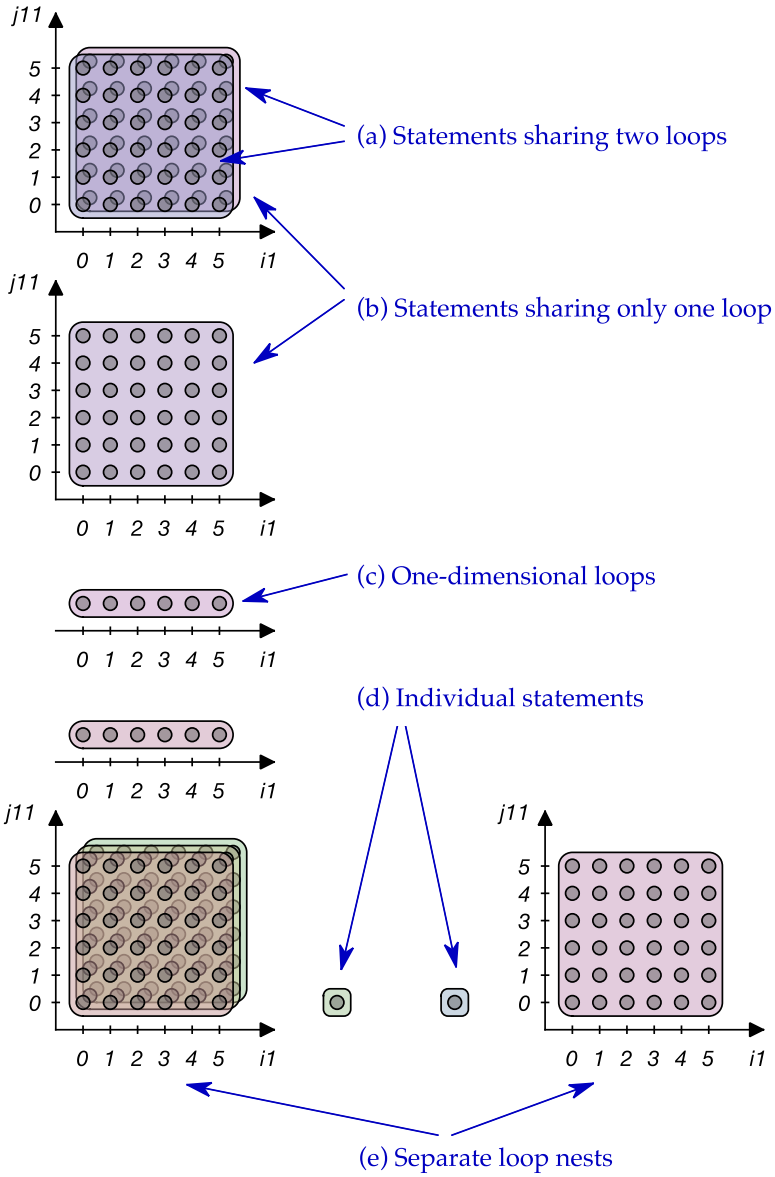


Figure 30 – Statements sharing a loop also share an axis. Separate loop nests are different coordinate systems. Individual statements outside loop depicted as points outside coordinate systems.

the choice of using only two-dimensional projections is motivated by the intended direct manipulation of the visual components with a standard 2D input device (e.g. mouse) [24]. It also allows to keep the visualization consistent regardless of the number of dimensions in the iteration domain. Finally, stacking multiple polygons is essentially equivalent to introducing another dimension interpreted as "depth" of the polygon.

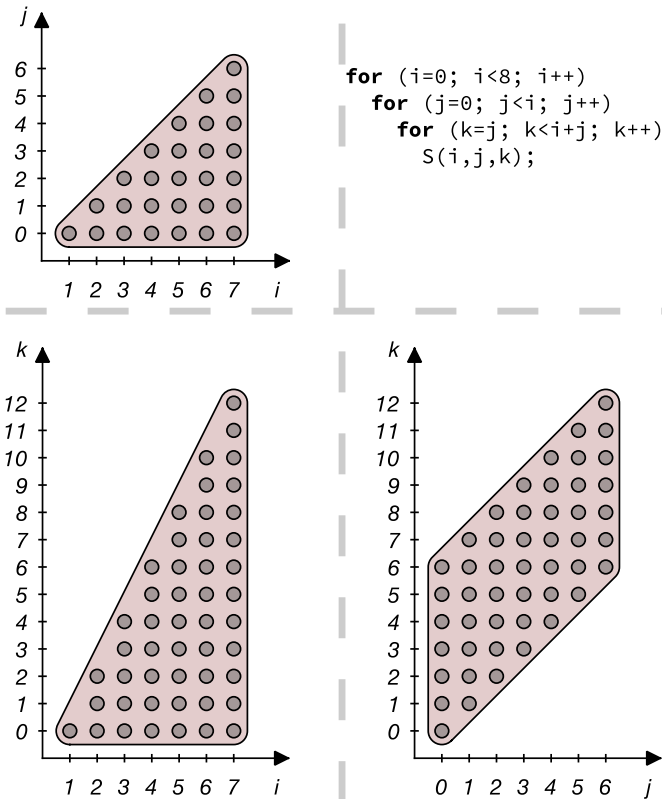


Figure 31 – Deep loop nests are visualized as a scatterplot matrix of two-dimensional projections corresponding to each pair of loops.

4.3.2 *Memory Accesses as Interactive Color Coding*

Reorganizing repeated memory accesses is another way to improve program performance along with exploiting parallelism. The polyhedral model allows to reason about data accessed by individual loop iterations. Given that data is structured as arrays or array-like constructs with affine subscripts, one is able to precisely identify the elements accessed by each individual execution of a statement nested in loops. These accesses are typically uniform across all executions for static control code. Non-uniform cases may arise from statements surrounded by branching control flow. However, it leads to a constant non-parametric number of branches, adequately representable in the code. The loop iteration domain is thus separated into several subdomains, each of which featuring uniform accesses. This uniformity allows us to speak about data access *patterns* rather than individual accesses.

Due to the definition of the dependence — two dependent statement instances access the same address —, our data dependence visualization partly presents the information about array accesses. Indeed, statement instances connected by an arrow access at least one common address with data. However, it does not give any information about accesses to adjacent elements. Most modern systems

use deep memory hierarchies with multiple caches significantly decreasing the delay time for accessing subsequent or close memory addresses [200], making performance gains from exploiting data locality important.

We enhanced our visualization with information about adjacent accesses. When the user hovers a specific point in the visual representation, points that correspond to statement instances accessing the same address are highlighted in the same dark red color. Points that access immediately preceding addresses are highlighted with lighter and less saturated shades of red using the metaphor of "hot addresses" meaning that they are likely to be still in the cache. Points accessing succeeding addresses are highlighted with shades of yellow. This highlighting is made across all available coordinate systems in order to hint the user about the data reuse between loops and the potential for loop fusion.

For example, the polynomial multiplication example from Listing 1 can be prepended with a data initialization step to demonstrate data reuse as shown in Listing 7. Figure 32 shows the *Clint* visual representation for this example with one point hovered. The right coordinate system corresponds to the main computation and shows the same diagonal pattern as the dependence lines. The left coordinate system also has several points highlighted suggesting that similar sets of addresses are accessed in two loops and fusing them in two loops may leverage the access locality to improve program performance.

```
for (i = 0; i < N + M - 1; i++)
  Z[i] = 0;
for (i = 0; i < N; i++)
  for (j = 0; j < M; j++)
    Z[i + j] += A[i] * B[j];
```

Listing 7 – Array initialization and polynomial multiplication

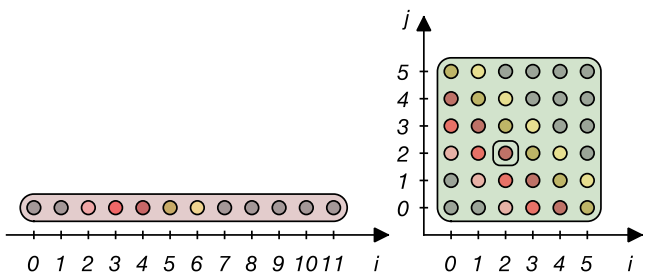


Figure 32 – Access pattern visualization: dark red iterations access the same address as the selected point, lighter reds access preceding addresses and yellows access immediately following addresses.

Although this representation of memory accesses is easily combinable with the dependence/parallelism view, it has various issues. First of all, it reacts to the user action and does not provide information statically. It does not take into account the fact that one state-

▷ *Color coding for memory accesses is limited to a specific instance, rather than gives a pattern.*

ment iteration may accesses multiple addresses during its execution. Finally, it is limited to the addresses adjacent to the selected iteration, rather than giving general information about all points. However, the last issue is partly alleviated by the access uniformity.

4.3.3 *Memory Accesses as Nested Parallel Coordinates*

We created another visual representation based on parallel coordinates [125] to address these issues. Contrary to the arbitrary multi-dimensional data, statement instances have special semantics associated with each dimension that we may embed in the visual representation. In particular, inner loop iterations are executed *inside* a particular outer loop iteration rather than being independent. At the same time, the amount of data points in an individual loop is substantially less than typical number of data points used in parallel coordinate application. Therefore, we propose to use *nested parallel coordinates*, a variant of parallel coordinates visualization where an axis is replicated for each value of the previous axis. We expect this technique to let the user trace individual lines and visually detect patterns in nested loop executions.

▷ *Nested parallel coordinates reflect the relation between dimensions in the polyhedral model.*

We visualize both α and β dimensions of the scheduling relation in order to separate executions of different statements nested in the same loop. For example, the first statement of Listing 7 is represented in nested parallel coordinates in Figure 33. Its first axis `i` corresponds to the loop, and the set of vertically aligned axes `b2` to the β-dimension inside this loop. The axis on the right corresponds to the subscripts of the array `Z` accessed by each execution of the statement in the loop.

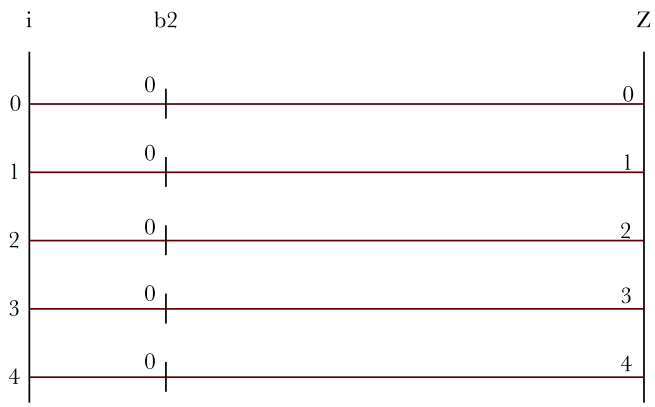


Figure 33 – Nested parallel coordinates access representation of access pattern. Each value of `t` is associated with a separate axis for `b2`.

Similarly, Figure 34 shows which `Z` subscripts are accessed by which iteration of the loop nest. The `j` axis is replicated for each pair of values `i` and `b2`. The `b3` is replicated for each triple `i`, `b2`, `j` and contains the single value as the loop nest contains only one statement. Follow-

ing the line back from the address axis, the user is able to identify all statements instances accessing the same data element.

Nested parallel coordinates representation allows to represent addresses accessed by all statement instances in the same view without having the user interact with the representation. If the statement accesses multiple addresses in the same array, multiple lines are drawn and color coded to identify each reference (see Figure 34). In case of multiple accesses to different arrays, extra axes are added on the right and connected with the statement instance lines. Thus parallel coordinates enables the user to visually analyze addresses commonly accessed together giving hints for data layout transformations such as transforming structure of arrays into array of structures.

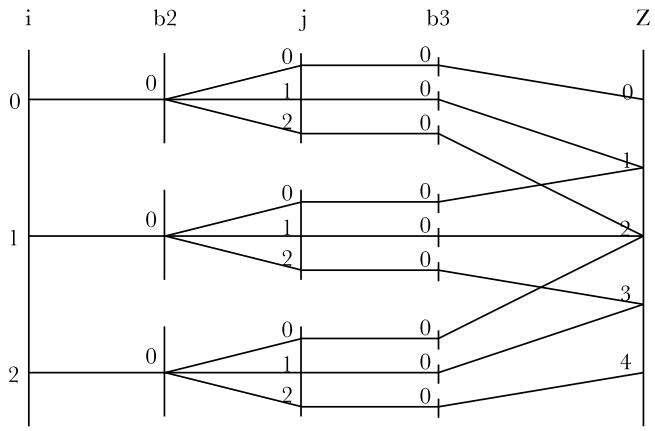


Figure 34 – Nested parallel coordinates access representation of access pattern. Lines join on the iterator axis only if the statements share the corresponding loop.

Multiple statements and loops are managed implicitly by this representation thanks to the presence of beta-dimensions that encode lexical order and nesting of statements. For example, the code from Listing 7 can be transformed to enable loop fusion as written in Listing 8. The transformed code contains an imperfectly nested loop, which the representation reflects by drawing a dashed line over the dimensions not present for the first statement in Figure 35. Note also that the `b2` axes now have two values each, but are still separated from each other.

▷ Allows to reason about loop parallelism, but less efficiently than scatter plot-based representation.

Contrary to conventional parallel coordinates, this representation maintains the hierarchical structure of the loops and the lexical order of the statement by using multiple axes. This hierarchy allows to reason about iteration independence and potential parallel execution. For example, iterations of the loop `i` are accessing the same addresses as seen on the Figure 34, making them dependent on each other and preventing parallel execution. However, iterations of the loop `j` access different addresses and may be executed in parallel. In the modified code visualized in Figure 35, different iterations of the `i` loop access different addresses and are visually separated, demonstrating their

independence and potential parallel execution. Although potentially helpful for identifying outer loop parallelism, this representation is hard to use for detecting inner loop parallelism as *all* replications of the inner loop axis should be analyzed visually.

```
for (i = 0; i < N + M - 1; i++) {
  Z[i] = 0;
  for (j = max(0, i - N + 1); j <= min(i, M - 1); j++)
    Z[i] += A[i-j] * B[j];
}
```

Listing 8 – Skewed and fused array initialization and polynomial multiplication

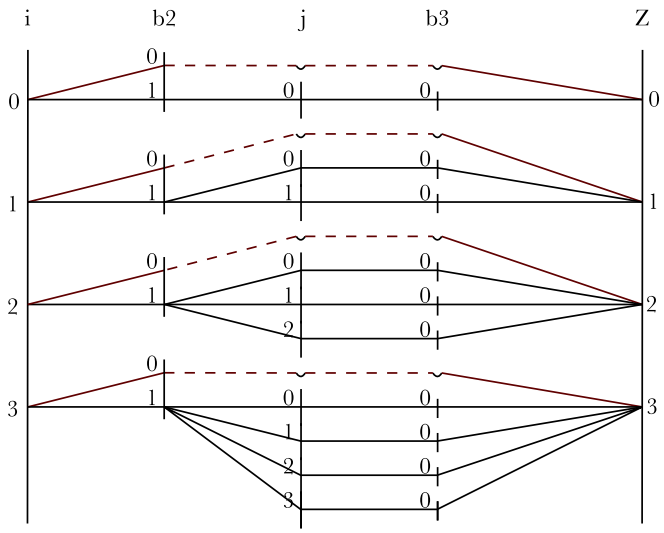


Figure 35 – Nested parallel coordinates access representation for multiple imperfectly nested loops. Dashed lines correspond to statements that are outside of inner loop.

4.3.4 *Clint Interface*

As shown in Fig. 36, *Clint* combines three components:

1. the visual representation of the program statements;
2. an editable history view; and
3. the source code editor, in which the program structure can be edited manually.

The source code editor includes ad-hoc syntax highlighting so that statements are displayed in the same color as their corresponding polygons in the visual representation. The three views are synchronized and updated according to user's manipulations. Changes made to the source code are immediately reflected on the visualization and added to the history. As these changes may intentionally modify the program semantics, dependences are recomputed but are not checked for violation against the previous version.

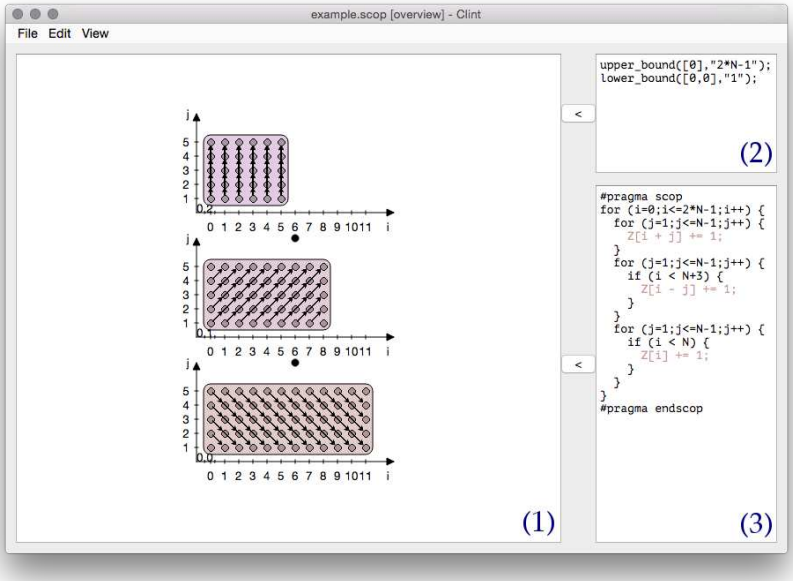


Figure 36 – The interface of *Clint* consists of three coordinated editable parts: (1) scatterplot-like visualization, (2) editable history view and (3) source code editor.

4.4 ARCHITECTURE AND IMPLEMENTATION

4.4.1 *Statement Instances*

Clint relies on a collection of research tools and libraries that are the building blocks for polyhedral compilers. These tools usually work together as a single black-box from the input source code to the output final code, with an internal flow based on a common polyhedral representation specification OpenScop [17]. Figure 37 illustrates the interplay between different parts of this tool chain. Specifically, *Clan* [20] raises a C program into the polyhedral model with the union of relations representation; *Candl* [250] performs the data dependence analysis in the polyhedral model; and *CLooG* [16] generates a C+OpenMP code that implements the current schedule. Expert programmers may use these tools separately to get feedback from either the data dependence analysis, e.g., to check whether a given code modification violates dependences or not, or the code generation to get the final code. Internally, *Clint* relies on the polyhedral libraries *isl* [253] and *piplib* [93] to perform projections and to create a set of points for the scatterplot from the iteration domain relations.

INSTANCE ENUMERATION In order to visualize the program from polyhedral relations, *Clint* combines the iteration domain relation union with the current scheduling relation union: the *scheduled domain* relation union is a result of combining inequalities from each

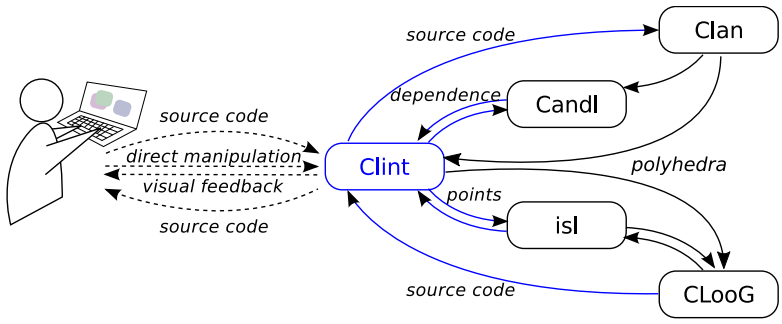


Figure 37 – *Clint* architecture: the user interacts only with *Clint* with the visualization and code being generated automatically by the polyhedral tool chain.

pair of domain and scheduling relations in a new relation. Each relation in the scheduled domain is than contextualized with user-defined values replacing symbolic constants introduced as extra equations to the relation definition. Finally, a polyhedral library transforms the contextualized scheduled domain into a set of points Rather than projecting the relation on the pair of dimensions, we compute the set of points belong to the corresponding polyhedron. Coordinates of these points are used for constructing the scatter plot. Having a set of points rather than individual projections allows *Clint* to properly display cases where different amounts of statement instances are projected to a single point. After placing all points, the surrounding shape is created by a Graham scan algorithm for convex hull computation knowing that the model requires individual polyhedra to be convex.

For example, the polynomial multiplication iteration domain (3) with the transformed scheduling (7) and fixed parameters results in a single relation with all the (in)equations combined (18).

$$\mathcal{D}_A = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \rightarrow \begin{pmatrix} b1 \\ a1 \\ b2 \\ a2 \\ b3 \end{pmatrix} \middle| \begin{array}{l} b1 = 0 \\ a1 = i + j \\ b2 = 0 \\ a2 = j \\ b3 = 0 \\ 0 \leqslant i < N \\ 0 \leqslant j < M \\ N = 11 \\ M = 14 \end{array} \right\} \tag{18}$$

STATEMENT NESTING IN LOOPS Beta-vectors of the statement's scheduling guide the distribution of individual polygons to the coordinate systems and their layout. First, the size of the beta-vector is

compared with the dimensions the statement is being projected onto: if the vector size is less than the horizontal dimension, the statement is visualized as a single point (the loop nest is not deep enough), if it is less than vertical dimension, the statement is visualized as a line, otherwise the statement is visualized as a 2D polygon. The beta-vector is than split into three parts between the beta-dimensions surrounding projection dimensions. The lexicographical order of the first part defines the order of "columns" of coordinate systems arranged horizontally. Statements with identical first part are placed in the same column. The lexicographical order of the second part, given identical first parts, defines the order of coordinate systems in the column. The lexicographical order of the last part defines, given first two parts being identical, the stacking order of statements within the coordinate system. For example, the statements from Listing 6 are distributed in Figure 30 according to their beta-vectors (provided in comments for readability) as shown in Figure 38.

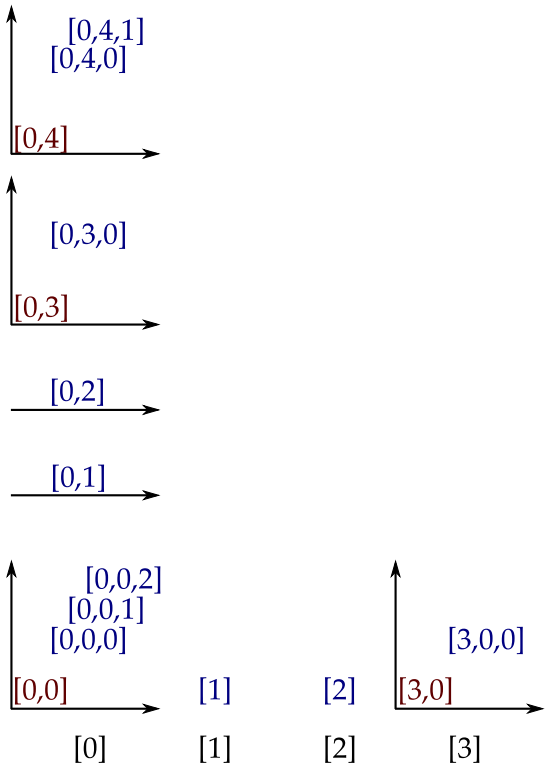


Figure 38 – Polygon alignment and distribution to coordinate systems is defined by statement beta-vectors. Blue numbers are beta-vectors, red numbers are beta-prefixes of coordinate systems. Black numbers are beta-prefixes of columns.

4.4.2 *Memory Accesses*

ADDRESS ENUMERATION *Clint* computes the set of addresses accessed by each statement instance similarly to the execution dates. It

takes the *scheduled iteration domain* and combines it with access relation unions resulting in *scheduled access relations*. If a statement accesses multiple different addresses, it is described by multiple scheduled access relations. After replacing the parameters by constant values, *Clint* uses *isl* to enumerate all integer points inside the polyhedron defined by the scheduled access relation. Once the set of integer points is computed, the scheduling dimensions and the array name are used as an index to group all addresses accessed by a statement instance.

For example, the second statement of the transformed polynomial multiplication snippet in Listing 7 has three distinct scheduled access relations: read/write on `Z[i+j]`, read on `A[i]` and read on B[j]. The relation (19) maps original iterators to the new ones and to the array subscript a_1 of `Z`.

$$\mathcal{A}_S^{Z,\mathrm{Read},\mathrm{Write}} = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to \begin{pmatrix} b1 \\ a1 \\ b2 \\ a2 \\ b3 \\ id \\ a_1 \end{pmatrix} \middle| \begin{array}{l} b1 = 1 \\ a1 = i \\ b2 = 0 \\ a2 = j \\ b3 = 0 \\ 0 \leqslant i < N \\ 0 \leqslant j < M \\ id = Z \\ a_1 = i + j \\ N = 11 \\ M = 14 \end{array} \right\} \tag{19}$$

DOMAIN SPLITTING As one statement may access multiple addresses and, conversely, one address may be accessed by numerous statement instances, the nested parallel coordinates visualization quickly becomes visually cluttered and difficult to interpret. At the same time, loop-based computation amenable to the polyhedral model features uniform access to the data, i. e. the access relation has a concise form mapping iteration variables to addresses. Therefore, it should be enough to only show the access *pattern*, repeated across all iterations rather than showing all accesses. Within the polyhedral model, these patterns can be expressed as piecewise affine functions. To reduce the amount of data represented, *Clint* computes the number of different addresses accessed by individual iterations. It finds the regions where this number is either constant or changes monotonously (increases or decreases with a constant factor) and only visualizes a limited number of iterations in these regions, e. g. 4 points in Figure 35. This approach reduces the amount of presented information, but still allows to understand the general pattern.

For the dependence visualization, the splitting approach may also be beneficial in order to reduce clustering, but requires a specific algorithm to find iteration domain parts with uniform dependences after transformation [108].

4.5 EVALUATION OF CLINT VISUALIZATION

Although similar visualizations have been already used for descriptive or pedagogical purposes, there is no empirical evidence of their appropriateness for conveying program structures. We designed an experiment to assess the suitability and completeness of our visual representation in the polyhedral model for programmers. We are testing whether programmers with different level of expertise are able to establish a bidirectional mapping between our visualization and the code with different levels of complexity.

4.5.1 *Protocol*

PARTICIPANTS We recruited 16 participants – 12 male, 4 female, aged 18-53 – from our organizations. All of them have experience in programming using imperative languages with C-like languages. Six participants already used iteration domain visualizations in their work and have some previous knowledge about the polyhedral model. Therefore they were considered as experts. In this study, we focus on the scatter plot-based visualizations as they provide information for both parallelism and data locality and are based on the participatory design.

In a controlled study, the environment is maintained consistent and only the factors change from task to task. Each participants sees all values (levels) of a within-factor and only one value of between-factor.

PROCEDURE We conducted a controlled experiment with a $[3 \times 2]$ mixed design having two factors:

— TASK: mapping direction (between participants)
  VC – writing a code snippet which corresponds to the given visualization using a C-like programming language featuring loops and branches with affine conditions;
  CV – drawing an iteration domain visualization given the corresponding code.
— DIFFICULTY: problems may be (within participants)
  Simple – two-dimensional with constant bounds;
  Medium – multi-dimensional with constant bounds;
  Hard – two-dimensional with branches and loops having mutually dependent bounds.

From the participatory design output, we expected deep loop nests to be the most problematic for users. However, during pilot testing we found that the the nest depth is not a source of difficulty itself, rather it creates more room for complex mutually-dependent loop bounds, that are difficult to envision. Therefore, we readjusted the difficulties

Pilot testing consists in running the preliminary version of the experimental protocol with a small sample of users in order to evaluate the protocol.

so as to avoid complex loop bounds in *Simple* and *Medium* conditions as well as to render the *Hard* conditions still feasible by limiting the dimensionality.

Performing the tasks in a certain order may affect results. In particular, participants can learn to use one representation better than others. Therefore, the order of tasks is changed, or counterbalanced, for different participants.

In order to avoid learning effect and to ensure consistent difficulty over tasks, participants were divided in two groups with the same number of experts. Group 1 was asked to perform the visualization to code task (*VC*), and group 2 the code to visualization task (*CV*). The order of task difficulty was counterbalanced across participants. Both tasks were performed on paper, with squared graph paper for the *CV* condition. Participants were presented with the visualization and did two practice tasks at the beginning of the session. They were instructed to perform the tasks as accurately as possible without time limit and were allowed to withdraw from a task. Expected solutions were shown at the end of the experiment. Each session lasted about 20 min.

DATA COLLECTION For each trial, we measured *Completion Time*, *Error* and *Abandon* rates. The errors were split in two categories: *Parameter Errors*, the shape of the resulting polyhedron was drawn correctly, but linear sizes or position were wrong; *Shape Errors*, the shape of the polyhedron was incorrect. Codes describing the same iteration domain were considered equivalent (e.g. `i <= 4` and `i < 5`). We also videotaped participants activity and collected the materials they produced. After they completed the study, participants filled in a questionnaire with questions regarding their strategies to accomplish the task and suggestions for improving the visualization.

DATA PROCESSING We performed *log-transformation* of the *Completion Time* during the analysis to compensate the positive skew of its distribution, natural to non-negative time measurements. This transformation results in asymmetric confidence intervals.

DATA ANALYSIS Due to growing concerns in various research fields over the limits of null hypothesis significance testing for reporting and interpreting experimental results [73, 82], we base all our analyses and discussions on estimation, i.e., effect sizes with confidence intervals [74]. Furthermore, we argue that estimation results and accompanying graphs are better suitable for communicating results in a work mixing methodologies from different disciplines. We describe the computation and interpretation of effect sizes and confidence intervals in Appendix B.

4.5.2 *Results*

We did not observe any significant learning effect on the *Error Rate* or *Completion Time*. We discarded 7 trials in which the non-expert

participants produced syntactically incorrect or not static control code as we could not unambiguously interpret participants intention with this code and attribute these problems to the visualization problems or the lack of expertise of the participants.

COMPLETION TIME TASK did not strongly affect the *Completion Time*. *VC* transition took 182s (95%CI = [127s, 262s]) on average while *CV* transition took 215s (95%CI = [156s, 296s]) on average, resulting in an effect size of 16.3% (95%CI = [−39.2, 50.9]). Although the previous exposure to similar visualization techniques was used as an expertise criterion, no interaction between expertise and TASK was observed, mostly due to large inter-participant variability and small number of tasks. This result may also suggest that only a short training may be sufficient to introduce *Clint* visualization to Non-Experts.

▷ *Experts completed only Hard tasks faster than Non-Experts.*

Expertise difference affected *Completion Times* only for *Hard* tasks, where Non-Experts took 539s (95%CI = [497s, 585s]) on average while Experts took 301s (95%CI = [201s, 451s]), with an effect size of 56.7% (95%CI = [26.8, 98.8]). Both Experts and Non-Experts spent similar amounts of time for *Easy*, 98s (95%CI = [49s, 199s]) and 82s (95%CI = [62s, 113s]) on average respectively, and *Medium* tasks, 208s (95%CI = [95s, 455s]) and 199s (95%CI = [151s, 262s]) on average respectively. In general, *Completion Time* is more consistent across Non-Expert participants than across Expert participants (Figure 39). Combined with our observations, this can be explained by Experts being substantially faster on some trials thanks to their expertise and spending more time on some other trials by looking for a potential caveat. On the other hand, Non-Experts were performing the task following the instruction without detailed reflections. Overall, similar *Completion Time* results suggest that our representation is suitable for both experts and non-experts if the complexity of the task remains limited.

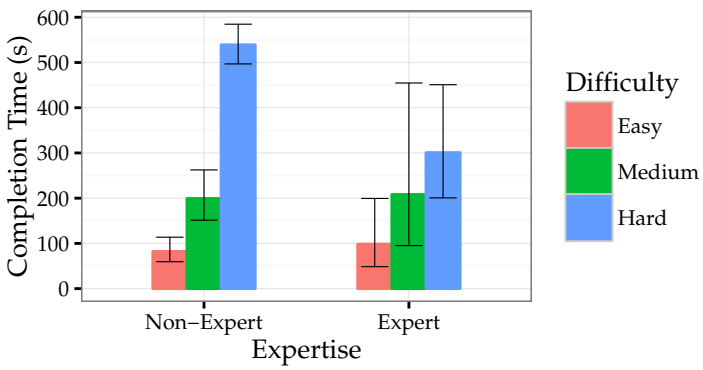


Figure 39 – *Completion time* increases with task difficulty but is lower for *experts*. Error bars show 95% confidence intervals.

▷ *Completion Time increases with Difficulty.*

The analysis of *Completion Times* also confirms our estimation of task DIFFICULTIES. On average, *Easy* tasks required 88s (95%CI = [66s, 117s]), *Medium* tasks required 202s (95%CI = [153s, 268s]), and *Hard*

tasks required 433s (95%CI = [354s, 530s]). Effect sizes between subsequent DIFFICULTY levels reach 78.8% (95%CI = [47.7, 99.2]) for *Easy* → *Medium* and 72.5% (95%CI = [39.9, 94.1]) for *Medium* → *Hard* showing a consistent growth. However, for Experts, the difference between DIFFICULTY levels is less visible due to large variability.

ERRORS AND ABANDONS Participants performed the tasks with very low error rates, 8.3% (95%CI = [−3.6%, 20.3%]) for *VC* tasks and 4.2% (95%CI = [−4.5%, 12.8%]) for *VC*. As for the type of errors, two Non-Experts proposed wrong code for *Hard VC* tasks, one *Parameter* and one *Shape Error*. One Expert made a *Parameter Errors* for a *Medium*. However, it is hard to conclude on the causes of errors with such low error rates.

▷ *Participants map Clint visualization to code reliably.*

We observed only two withdrawals during a trial (one *Hard VC* and one *Hard CV* task), both from non-experts, and after more than 500s. In both cases, participants tried to perform the task and explicitly stated their lack of confidence in the result as a reason for abandon. In general, these result suggests that the users can reliably map *Clint* visual representation to the code and vice versa (Figure 40).

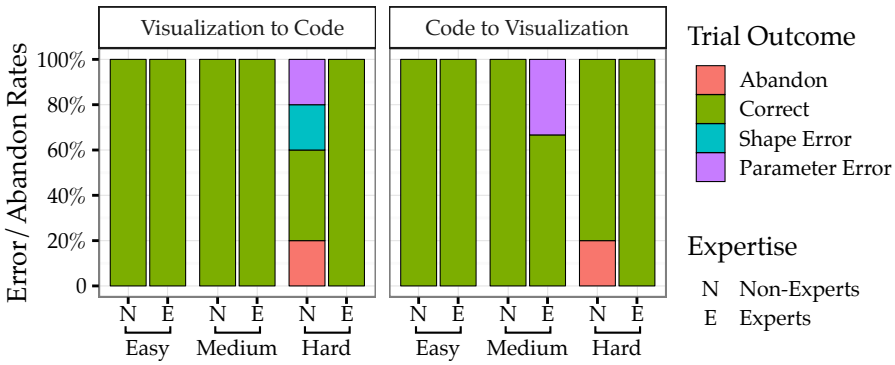


Figure 40 – Percentage of errors and withdrawal: experts were slightly more successful than non-experts, but failed at simpler tasks. Only non-experts abandoned tasks. The overall error rate is less than 10% for each task.

QUALITATIVE DATA Both Expert and Non-Expert participants learned the visualization approach rapidly and were confident in its interpretation most of the time. The majority of the feedback was concentrated on the disagreements observed before: directions and position of the axes and the traversal order. At the same time, when asked if he would like to have a configuration option for changing the axis direction, an Expert participant stated that "[he] *doesn't agree with axis pointing up, but will stick with the default representation and will not go to the depth of settings to change it.*" 50% of the participants stated that the visualization substantially helps to understand the loop structure or

execution, 31% that it rather helps and 19% that it does not change their level of understanding, but does not harm.

Overall, these results suggest that both expert and non-expert programmers could reliably map our visualization to the corresponding code, most of them stating that it has potential in assisting them in understanding programs. While the task they performed in this study does not belong to the program parallelization process, it shows that the scheduled iteration domain visualization is an efficient representation of static control parts of the program.

4.6 DISCUSSION

VISUALIZING INFORMATION NOT AVAILABLE IN CODE We designed a visualization tool for representing the execution of loop-based program parts. The base of its visualization technique is a combination of scatter plots and node-link diagrams widely used in the polyhedral compilation community. However, most existing works feature ad-hoc images with subtle differences in order to explain complex concepts of the program optimization in the polyhedral model or the model itself. After a participatory design session with experts in the model, we created a visual representation sharing the commonly used features and taking decisions for previously omitted edge cases such as individual statements or imperfectly nested loops. Our tool, *Clint*, is able to visualize any program part amenable to the polyhedral model.

Clint visualizes *information available through the polyhedral analysis, but not immediately accessible in the code*. It models the dynamic program behavior statically and allows to reason about individual statement instances executed within loop nests and dependences between them. This information, compactly folded in the code, is however crucial for loop-level optimization and parallelism extraction. Representing dependences as lines appeals to user's intuition: parallel lines correspond to potentially parallel loop.

▷ *Clint visualizes the information that is not immediately accessible in the code.*

TARGETING EXPERTS OR NOVICES Although based on feedback from experts, *Clint* visual representation is accessible for ordinary programmers with no specific background in loop optimization or optimizing algorithms development. During the experimental evaluation, both experts and non-experts were able to establish a reliable bidirectional mapping between the code and the visualization suggesting that they obtain equivalent information about program execution from both representations. Although experts in the polyhedral model often rely on visualization, we observed them to be reluctant to visual representations of simple program parts, supporting earlier studies on visual programming complexity [107, 202]. How-

ever, for complex cases, e. g. automatically optimized programs with regenerated code that features complex loop and branching conditions, visual representations preserving a clear geometric shape were widely requested. While *Clint* is useful for experts to manage complex cases, it may also have a potential in teaching programming, namely explaining loops and loop-level optimization on the simple cases, similarly to algorithm animation techniques.

SINGLE OR MULTIPLE REPRESENTATIONS Beside parallelism, the polyhedral model allows to reason about memory addresses accessed by individual statement instances and data reuse across loop iterations. Memory access optimization is often orthogonal to, if not hindering, the parallelism extraction. After trying to integrate memory access information into the parallelism-targeted visualization, we opted for a completely different representation, more appropriate for this information. Contrary to parallelism extraction where the independence of statement instances is sufficient, memory access analysis requires identifying groups of statement instances accessing the same or adjacent addresses. Therefore, the address, possibly multidimensional, should be explicitly represented, which is possible in parallel coordinates form. At the same time, our representation maintains the dimension hierarchy imposed by the lexicographic ordering making the link to the program structure clearer.

▷ *Both visualization techniques allow to discover parallelism and locality. Yet it requires more effort to discover information in a non-adapted representation.*

Scatter plot and parallel coordinates-based representations are targeted at presenting *different* information, statement instance execution and memory access, respectively, extracted from the program by polyhedral analysis. Nevertheless, each of these representations allows to discover another type of information: interactive highlighting hints the user about data access patterns in the scatter plot and non-intersecting lines imply parallelism in the nested parallel coordinates display. While this information can be also accessed by analyzing the source code, the time required to obtain it is substantially longer. In a more general case, the user may want to chose the representation that is appropriate to the analysis task at hand depending not only on the information that is available in this representation, but also on how easily it can be accessed and interpreted. Multiple representations may be combined in a coordinated multiple view interface letting the user select, but at the same time reducing the size of each representation and potentially the amount of information presented. It may also increase the cognitive load on the user as he would have to take decisions and potentially maintain the mapping between the representations [72, 203].

OPPORTUNITIES FOR INTERACTION While these program visualization techniques may be used to understand program execution and statement instance level, they remain static and do not allow to

modify the program by modifying the visual representation. These representation may serve as a basis for comparison between the original and the optimized program. Considering the potential for using graphical representation to manipulate programs in such comparison, we notice that both transformations for parallelism and for memory access optimization are better visible in the scatter plot form where they take form of polygon displacement and deformation. Indeed, basic *Clint* visualization uses a multi-level mapping of position to the execution order of statement instances as well as a mapping between polygon shapes and loop bounds. One may thus directly manipulate these properties and expect the corresponding modification of the code. On the contrary, parallel coordinates representation is manipulable mostly along the axes that span in one dimension reducing the opportunities for intensive direct manipulation. Finally, the loop-level optimization vocabulary draws inspiration from polygonal representation of loop nests proposing, for example, program transformations called loop skewing and shifting, easily transferable to our visual representation.

4.7 CONCLUSION

In this chapter, we detail the design and implementation of *Clint*, a visualization tool for representing the execution of loop-based program parts. The base of its visualization technique is a combination of scatter plots and node-link diagrams widely used in the polyhedral compilation community. We used participatory design sessions with experts in the polyhedral model to collect commonly used features and to design around previously omitted edge cases. *Clint* visualizes any program part amenable to the polyhedral model. It provides information available through the polyhedral analysis, but not immediately accessible in the code. An empirical evaluation demonstrated that both expert and non-expert programmers can reliably map between the visual representation and the code. While the visualization is designed to support direct manipulation, the tool requires a high-level transformation engine in the polyhedral engine.

In order to enable program transformation by directly manipulating the visual representation, we need a mapping between graphical actions and modifications to the scheduling relation union in the polyhedral model. However, the visual representation is not directly related to the relational form of the scheduling, nor is the relational form helpful for the user. Therefore, we suggest the mapping to be based on a set of higher-level program transformation *directives* that, when combined, allow arbitrary modifications of the scheduling relation union.

5 HIGH-LEVEL PROGRAM MANIPULATION

5.1 SEMI-AUTOMATIC PROGRAM RESTRUCTURING

Software visualization techniques simplify program analysis while searching for optimization, but offer no support for program restructuring that continues to rely on cumbersome and error-prone manual code modification. Automatic program optimization alleviates this problem, but introduces another one: if an optimization heuristic fails in a particular case, the user ends up with a barely readable and hardly modifiable automatically generated code with no speedup and no feedback about the reasons of failure. Instead of an automatic optimization, the developer may opt for a directive-based program transformation framework. In this case, the developer chooses a program transformation procedure from the list of available directives, e. g. loop fusion, distribution, tiling, etc. [265], and demands the framework to perform it by rewriting the code. Directive-based approach gives the developer more control over the optimization process, but requires to take all optimization decisions, even those which could be handled automatically in an optimal way. It also allows to decouple the optimizing restructuring from the program itself. A sequence of transformation directives can be stored separately from the code and reapplied as a part of the compilation process. This is especially important for sophisticated scientific computation codes that should maintain a decent level of code readability while offering high performance.

We propose to use a combined *semi-automatic* approach to program restructuring. A program optimization algorithm computes a transformation that satisfies a certain predefined criterion. Instead of modifying the program directly, it yields a sequence of program transformation directives that may be stored along the code. Once computed, this sequence can be applied by a directive-based framework or modified by the developer. Transformation directive essentially become a first-class object in developer-compiler interaction. Transformation sequences can be stored, analyzed, modified and reused. Optimization may be performed separately, using more time and computation resources than a relatively cheap transformation application, enabling more complex and precise algorithms. However, within the polyhedral model, the optimization algorithm *does not operate in terms of transformation directives* at any point, see Figure 41. as it relies on mathematical representations of the program and uses, e. .g., integer linear programming to compute optimized schedules.

Practical polyhedral optimizers often apply syntactic transformations separately from the main algorithm, e.g. Pluto has "–tile" option.

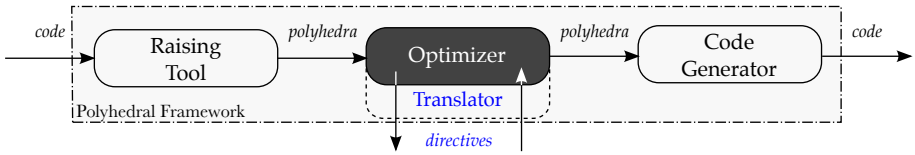


Figure 41 – Polyhedral framework is essentially a black box offering user little control over the optimization process. We propose to translate polyhedral relation changes to high-level transformation directives (light blue) to interact with the optimizer.

In order to enable semi-automatic program transformation in the polyhedral model, which, in turn, enables interactive program restructuring, we need to, first, express a set of program transformation directives as operations on polyhedral unions of relations and, second, develop an algorithm that identifies a sequence of these directives, equivalent to the modification of a given polyhedral scheduling relation union. To ensure such sequence exists, the transformation set should be complete, i. e. allow transforming any polyhedral scheduling union into any other. Combining transformation sequence identification algorithm with the directive-based polyhedral transformation engine, the user will be able to "reverse engineer" the functionality of a polyhedral optimizer, understand what transformations are performed, analyze why they may fail and manually modify the sequence only when necessary. Finally, transformation directives may be mapped to graphical element manipulations in *Clint* enabling direct manipulation for program transformation and interactive animation for program transformation visualization and analysis.

5.2 FROM TRANSFORMATION DIRECTIVES TO POLYHEDRA

5.2.1 *Loop Transformations Expressed in Polyhedral Model*

The polyhedral model is notoriously difficult to handle directly. Over the years of model development, various algorithms and tools were proposed to simplify routine tasks including semi-automatic program transformation. Contrary to fully automatic optimization where a transformation is selected and performed to reach a specific objective, a semi-automatic approach is based on user input requesting program manipulations rather than defining objectives. In this case, the program transformation framework only performs the operation and provides feedback on its effect (semantics preservation, parallelism, expected time gain) without deciding on the nature of transformation.

Several frameworks expose a high-level interface on top of a polyhedral engine, UTF [138] being arguably the first of them. The URUK framework [104] enables the composition of complex sequence of classical loop transformations [265] decoupled from any syntactic form

of the program. CHiLL [58, 221] provides an alternative mapping between syntactic loop transformations and polyhedral schedulings with *alignment* pre-processing step to ensure iteration domains have identical dimensionality. AlphaZ provides directives for user-guided transformation of the data layout and access patterns in the classical loop transformations [271]. These frameworks provide mappings of well-known transformations, such as *loop fusion* or *loop tiling*, to the polyhedral representation. However, they do not strive for completeness of the transformation set, i. e. the possibility to express an arbitrary polyhedral scheduling using only the high-level transformation. To alleviate this limitation, CHiLL and AlphaZ feature so called *linear transformations* that allow to apply an affine function to the program scheduling. A composition of such linear transformations allows to express most modifications to the polyhedral scheduling except those affecting its dimensionality. Linear transformations are in fact a workaround for schedules that cannot be expressed otherwise. They still require at least basic understanding of the polyhedral model by relying on a specific representation such as affine scheduling functions or scheduling relations. Furthermore, some loop transformations including *loop tiling* and *index-set splitting* require modification of the iteration domain within this framework. Changes to the domain require extra care after transformation in order to preserve original program semantics since they may affect the number of statement instances executed in a loop. For example, an improperly constructed strip-mining transformation may result in some instances not being executed if the number of loop iterations is not evenly divisible by the tile size.

We present a new revisiting of classical loop transformations in the polyhedral model, called after its implementation *Clay* [1]. Contrary to many existing approaches that also expose high-level transformation directives on top of a polyhedral engine, it is based on the more general *union of relations* abstraction described in Section 3.3 rather than on scheduling functions. This abstraction encodes all information about the program schedule in the scheduling relation union allowing us to delay the complex and expensive semantics preservation check until after the entire transformation sequence is applied. Transformations from a sequence are applied one by one and only modify the scheduling relation union. As long as the *global validity* requirements (see Section 3.5.8) are met, each individual statement instance is always assigned a unique integer execution date and place. Therefore, the semantics preservation check based on date comparison is always possible. Global validity conditions also require the scheduling relation union to be compatible with the domain. Since the domain is not modifiable in the union of relations abstraction,

1. Chunky Loop Alteration wizardrY

there is no need to verify if it combines with the scheduling after every transformation.

In *Clay*, we strive to provide a *complete* set of syntactic-level transformations allowing to transform any well-formed scheduling relation union to any other well-formed scheduling union. These transformations, inspired from existing mappings and sets of classical loop manipulations, can be expressed syntactically as well as in the polyhedral model. Given the low cost of transformation composition, we avoid *linear transformations* and propose to decompose them into primitive transformations with stronger semantics. Finally, we define the transformations so as to respect the *global validity* requirements by construction. With this conditions, *Clay* relaxes unimodularity and invertibility limitations present in the some previous work. *Clay* embeds the complete scheduling information in a single relation union per statement, even for the transformations previously requiring iteration domain modifications (StripMine and IndexSetSplit), thus removing the previously necessary intermediate data dependence graph updates and checks.

Clay relies on the scheduling structure with β-dimensions described in Section 3.5.6 in order to specify transformation target. Even scheduling dimensions, denoted α, define the execution order within the loop. Odd scheduling dimensions, denoted β, encode the lexical order of statements and their nesting in loops. Hence, the general form of the scheduling output dimension vector is $\vec{\sigma} = (\beta_1, \alpha_1, \beta_2, \alpha_2 \ldots, \beta_n, \alpha_n, \beta_{n+1})^T$. The original scheduling of a program can be constructed as follows. For a statement enclosed in an n-dimensional loop, we introduce $(2n+1)$ logical date dimensions. The β_i dimension denotes the lexical position of loops iterating over i^{th} nesting level, while the α_i denotes iteration variables of such loops. Last β dimension denotes the lexical position of the statement in the most nested loop. Beta-vectors, that consist of non-parametric constant values involved in definitions of β-dimensions, define a statement targeted by a particular transformation.

To express classical loop transformations using the relation formalism and the union of relations structure, we use notations and operators shown in Figure 42. They are used to represent specific subsets of scheduling union components and relation dimensions. The notion of β-prefix is paramount. It is a prefix of a β-vector, i.e. it contains some leading elements of the β-vector, but not all of them. β-prefixes are used to select specific subsets of relations to be affected by the transformation. From a syntactic point of view, a β-prefix addresses a specific loop (or, equivalently, the set of statements enclosed in that loop). The empty vector is a particular β-prefix used to select all scheduling relations, or, from a syntactic point of view, the root of the program.

θ	scheduling relation union with parameters $\vec{p}$: $\theta(\vec{p}) = \bigcup_i \mathcal{T}_i(\vec{p})$
$\mathcal{T}$	scheduling relation, defined by a set (system) of affine constraints $\mathcal{C} = \{af \geqslant 0\}$
$\mathcal{C}$	set of affine constraints defining the scheduling relation $\mathcal{T}$
af	an arbitrary multidimensional affine function with integer coefficients $af(\vec{x}) = \vec{k}\vec{x}^T + c$, $\vec{k} \in \mathbb{Z}^{\dim \vec{x}}, c \in \mathbb{Z}$
$\vec{p}$	vector of constant parameters
$\vec{\iota}_{\mathcal{T}}$	vector of input dimensions of $\mathcal{T}$
$\vec{\sigma}_{\mathcal{T}}$	vector of output dimensions of $\mathcal{T}$ (includes interleaved α and β dimensions)
$\vec{\alpha}_{\mathcal{T}}$	α-vector of $\mathcal{T}$, i.e., the vector of even dimensions of $\vec{\sigma}_{\mathcal{T}}$
$\vec{\beta}_{\mathcal{T}}$	β-vector of $\mathcal{T}$, i.e., the vector of odd dimensions of $\vec{\sigma}_{\mathcal{T}}$
$\vec{\rho}$	β-prefix; if empty, corresponds to the root of the SCoP
$\vec{\square}_{\mathcal{T},i}$	symbolic i^{th} element of any vector $\vec{\square}$; corresponds to the symbolic expression of the ith dimension of $\mathcal{T}$
$\mathcal{T}_*$	set of all scheduling relations
$\mathcal{T}_{\vec{\rho}}$	subset of $\mathcal{T}_*$ restricted to relations such that $\vec{\rho}$ is a β-prefix, i.e. $\vec{\beta}_{\mathcal{T},1..\dim \vec{\rho}} = \vec{\rho}$
$\mathcal{T}_{\vec{\rho},next}$	subset of $\mathcal{T}_*$ such that $\vec{\rho}_{1...\dim \vec{\rho}-1}$ is the β-prefix and $\beta_{\mathcal{T},\dim \vec{\rho}} = \vec{\rho}_{\dim \vec{\rho}} + 1$; denotes all scheduling relations inside the loop that is immediately following the one defined by $\vec{\rho}$
$\mathcal{T}_{\vec{\rho},>}$	subset of $\mathcal{T}_*$ restricted to relations such that $\vec{\rho}_{1..\dim \vec{\rho}-1}$ is the β-prefix and $\beta_{\mathcal{T},\dim \vec{\rho}} > \vec{\rho}_{\dim \vec{\rho}}$; denotes all scheduling relations for statements and loops following the one defined by $\vec{\rho}$ within the same enclosing loop
$a \mapsto b$	substitution operator: symbolically replaces all occurrences of a with b throughout scheduling relations

Figure 42 – Notations and Operators used in the Clay Formalism

5.2.2 *Revisiting Classical Transformations in Clay*

We revisit classical loop transformations [265] using the union of relations abstraction and notations in Fig 42. In addition to existing transformations, we propose new loop-level transformations, RESHAPE, COLLAPSE, DENSIFY and LINEARIZE. These transformations are required to ensure the completeness of the *Clay* transformation set. Most of them are complementary to existing transformations, e.g. LINEARIZE undoes STRIPMINE.

Each transformation is presented as a primitive, arguments of which can be integers (i), integer vectors (v) or affine inequalities (a). It

also includes preconditions for the parameters in order to enforce *global validity* of the scheduling after the transformation if the initial scheduling respected it. The rest of this Section describes *Clay* transformations, their preconditions and effects.

First precondition implies that $\vec{\rho}$ is a strict prefix, which targets a loop and not an individual statement.

REORDER $(\vec{\rho}, \vec{v})$

Preconditions: $\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho}}, \dim \vec{\rho} < \dim \vec{\beta}_{\mathcal{T}}$;
$\dim \vec{v} = \max_{\mathcal{T} \in \mathcal{T}_{\vec{\rho}}} \left(\vec{\beta}_{\mathcal{T}, \dim \vec{\rho}+1} \right) + 1$;
$\forall i \;\; 1 \leqslant i \leqslant \dim \vec{v}, \;\; 0 \leqslant \vec{v}_i \leqslant \dim \vec{v} - 1$;
$\forall i, j : i \neq j, \;\; \vec{v}_i \neq \vec{v}_j$;
β-vectors are normalized.

Effect: $\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho}}, \;\; \vec{\beta}_{\mathcal{T}, \dim \vec{\rho}+1} \leftarrow \vec{v}_{\vec{\beta}_{\mathcal{T}, \dim \vec{\rho}+1}}$;

reorganizes statements and loops inside the loop defined by $\vec{\rho}$ according to the vector $\vec{v}$. The ith element of $\vec{v}$ corresponds to the new position of i-th statement (loop), sorted lexically.

FUSENEXT $(\vec{\rho})$

Preconditions: $\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho}}, \dim \vec{\rho} < \dim \vec{\beta}_{\mathcal{T}}$;
$\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho}}, \exists \vec{\beta}_{\mathcal{T}} : \vec{\beta}_{\mathcal{T}, 1..\dim \vec{\rho}-1} = \vec{\rho}_{1..\dim \vec{\rho}-1} \wedge$
$\wedge \vec{\beta}_{\mathcal{T}, \dim \vec{\rho}} = \vec{\rho}_{\dim \vec{\rho}} + 1 \wedge \dim \vec{\beta}_{\mathcal{T}} > \dim \vec{\rho}$;
β-vectors are normalized.

Effect: $\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho}, \mathtt{next}}$,
$\vec{\beta}_{\mathcal{T}, \dim \vec{\rho}+1} \leftarrow \vec{\beta}_{\mathcal{T}, \dim \vec{\rho}+1} + \max_{\mathcal{T} \in \mathcal{T}_{\vec{\rho}}} \left(\vec{\beta}_{\mathcal{T}, \dim \vec{\rho}+1} \right) + 1$;
$\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho}, >}, \vec{\beta}_{\mathcal{T}, \dim \vec{\rho}} \leftarrow \vec{\beta}_{\mathcal{T}, \dim \vec{\rho}} - 1$

fuses the loop corresponding to $\vec{\rho}$ with its direct successor. Keeps the original order of nested loops and statements.

DISTRIBUTE $(\vec{\rho}, n)$

Preconditions: $\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho}}, \dim \vec{\rho} < \dim \vec{\beta}_{\mathcal{T}}$;
$n \in \mathbb{N} \wedge 1 \leqslant n < \max_{\mathcal{T} \in \mathcal{T}_{\vec{\rho}}} \left(\vec{\beta}_{\mathcal{T}, \dim \vec{\rho}+1} \right)$;
β-vectors are normalized.

Effect: $\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho}, >}, \;\; \vec{\beta}_{\mathcal{T}, \dim \vec{\rho}} \leftarrow \vec{\beta}_{\mathcal{T}, \dim \vec{\rho}} + 1$;
and normalize β-vectors using algorithm described before in Figure 26.

distributes statements in the loop between two succeeding loops, the first containing first n statement of the original loop, i. e. those with $\vec{\beta}_{\mathcal{T}, \dim \vec{\rho}+1} < n$, and the second loop containing the remaining statements.

SHIFT $(\vec{\rho}, i, \mathit{amount})$

Preconditions: $\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho}}, i \in \mathbb{N} \wedge 1 \leqslant i \leqslant \dim \vec{\rho} < \dim \vec{\beta}_{\mathcal{T}}$ and
$\mathit{amount} = \vec{v} \cdot \vec{p} + C, \vec{v} \in \mathbb{Z}^{\dim \vec{p}}, C \in \mathbb{Z}$

Effect: $\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho}}, \vec{\alpha}_{\mathcal{T}, i} \mapsto \vec{\alpha}_{\mathcal{T}, i} + \mathit{amount}$

moves all instances of statements with β-prefix $\vec{\rho}$ in the iteration space by constant (parametric) *amount* in the ith output loop. For relations with only *explicitly defined* dimensions, it may be performed by modifying the parameters and the constant in the definition by $-$*amount*. Substitution is required to preserve inequalities, including those *implicitly defining* the dimension.

Explicitly defined dimensions are expressed as equations while implicitly defined dimensions are expressed by a set of inequalities.

SKEW $(\vec{\rho}, i, k)$

Preconditions: $\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho}},\ i \in \mathbb{N} \wedge 1 \leqslant i < \dim \vec{\beta}_{\mathcal{T}} \wedge i \neq \dim \vec{\rho}$;
$k \in \mathbb{Z}$

Effect: $\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho}}, \vec{\alpha}_{\mathcal{T},\dim \vec{\rho}} \mapsto \vec{\alpha}_{\mathcal{T},\dim \vec{\rho}} + k \cdot \vec{\alpha}_{\mathcal{T},i}$

makes the loop iterator at depth $\dim \vec{\rho}$ traverse the values of the loop iterator at depth i with a coefficient k (*skew factor*). This operation takes into account the *output* dimension, i. e. the ith loop iteration variable in the transformed code.

REVERSE $(\vec{\rho})$

Preconditions: $\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho}},\ \dim \vec{\rho} < \dim \vec{\beta}_{\mathcal{T}}$

Effect: $\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho}}, \vec{\alpha}_{\mathcal{T},\dim \vec{\rho}} \mapsto -\vec{\alpha}_{\mathcal{T},\dim \vec{\rho}}$

reverses the iteration order of the loop at depth $\dim \vec{\rho}$ for all statement instances with β-prefix $\vec{\rho}$.

INTERCHANGE $(\vec{\rho}, i)$

Preconditions: $\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho}}, \dim \vec{\rho} < \dim \vec{\beta}_{\mathcal{T}}$;
$i \in \mathbb{N} : 1 \leqslant i < \dim \vec{\rho}$

Effect: $\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho}},\ \ \vec{\alpha}_{\mathcal{T},i} \mapsto \vec{\alpha}_{\mathcal{T},\dim \vec{\rho}} \wedge \vec{\alpha}_{\mathcal{T},\dim \vec{\rho}} \mapsto \vec{\alpha}_{\mathcal{T},i}$,
both substitutions are done simultaneously

in a loop nest including statements with β-prefix $\vec{\rho}$, swaps the loop at depth i with the loop at depth $\dim \vec{\rho}$.

RESHAPE $(\vec{\rho}, i, k)$

Preconditions: $\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho}}, \dim \vec{\rho} < \dim \vec{\beta}_{\mathcal{T}}$;
$i \in \mathbb{N} : 1 \leqslant i < \vec{\rho}; k \in \mathbb{Z}, k \neq 0$.
Either one of $\alpha_{\mathcal{T},i}$, $\alpha_{\mathcal{T},\dim \vec{\rho}}$, is *implicitly defined*.
or, for $\alpha_{\mathcal{T},i}$, $\alpha_{\mathcal{T},\dim \vec{\rho}}$ *explicitly defined* as

$$\alpha_{\mathcal{T},i} = \vec{v} \cdot \vec{\iota}^{T} + af_1(\vec{p}) + C_1 \text{ and}$$
$$\alpha_{\mathcal{T},\dim \vec{\rho}} = \vec{w} \cdot \vec{\iota}^{T} + af_2(\vec{p}) + C_2,$$
$$\left[\begin{array}{l} \nexists d \in \mathbb{R} : \forall j \neq i, d = \frac{\vec{v}_j}{\vec{w}_j} \\ \left(\exists d \in \mathbb{R} : \forall j \neq i, d = \frac{\vec{v}_j}{\vec{w}_j}\right) \wedge \left(\frac{\vec{v}_j}{\vec{w}_j + k\vec{v}_j} \neq d\right) \end{array}\right.$$

In addition, for $\alpha_{\mathcal{T},\dim \vec{\rho}}$ *explicitly defined* as

$$\alpha_{\mathcal{T},\dim \vec{\rho}} = w \cdot \vec{\iota}_i + f_2(\vec{p}) + C_2,$$
$$w \neq -k,$$

These preconditions ensure that the number of linearly independent explicit definitions remains the same after the transformation.

Effect: $\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho}}, \vec{\alpha}_{\mathcal{T},\dim \vec{\rho}} \mapsto \vec{\alpha}_{\mathcal{T},\dim \vec{\rho}} + k \cdot \vec{\iota}_{\mathcal{T},i}$

reshapes the iteration space so that the loop iterator at depth $\dim \vec{\rho}$ depends on the *original* loop iterator at depth i with a coefficient k

(*reshape factor*). This reshape operation takes into account the *input* dimension, i. e. the ith loop iteration variable as it was present in the original code. Useful for expressing complex transformations, such as skewing inner loop by a fraction of the outer loop. The conditions prevent originally *explicitly defined* dimensions from becoming linearly dependent or constant, which would violate the *scheduling existence* constraint since the transformed loop would no longer fully traverse explicitly associated input dimensions.

INDEXSETSPLIT ($\vec{\rho}$, constraint)

Preconditions: $\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho}}$, constraint: $\vec{u} \times \vec{\alpha}_{\mathcal{T}}^{T} + \vec{v} \times \vec{\iota}_{\mathcal{T}}^{T} + \vec{w} \times \vec{p}^{T} + C \geqslant 0$, $\vec{u} \in \mathbb{Z}^{\dim \vec{\alpha}}, \vec{v} \in \mathbb{Z}^{\dim \vec{\iota}}, \vec{w} \in \mathbb{Z}^{\dim \vec{p}}, C \in \mathbb{Z}$

Effect: $\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho},>}, \vec{\beta}_{\mathcal{T},\dim \vec{\rho}+1} \leftarrow \vec{\beta}_{\mathcal{T},\dim \vec{\rho}+1} + 1$;
$\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho}}, \mathcal{T} \mapsto \mathcal{T}' \cup \mathcal{T}''$:
$\mathcal{T}' = \mathcal{T} \cap \text{constraint}$, $\mathcal{T}'' = \mathcal{T} \cap \neg\text{constraint}$,
$\vec{\beta}_{\mathcal{T}'',\dim \vec{\rho}+1} \leftarrow \vec{\beta}_{\mathcal{T}'',\dim \vec{\rho}+1} + 1$

replaces every scheduling relation with β-prefix $\vec{\rho}$ by a union of two disjoint relations depending on the constraint and having unique β-vectors; further transformations may target either relation.

COLLAPSE ($\vec{\rho}$)

Preconditions: $\exists \mathcal{T}', \mathcal{T}'' \in \mathcal{T}_{\vec{\rho}}$:
$\exists \mathcal{T} : \mathcal{T}' = \mathcal{T} \cap \text{constraint} \wedge \mathcal{T}'' = \mathcal{T} \cap \neg\text{constraint} \wedge$
$\wedge\, \vec{\beta}_{\mathcal{T}',1..\dim \vec{\rho}} = \vec{\beta}_{\mathcal{T}'',1..\dim \vec{\rho}} = \vec{\rho} \wedge$
$\wedge\, \vec{\beta}_{\mathcal{T}',\dim \vec{\rho}+1} + 1 = \vec{\beta}_{\mathcal{T}'',\dim \vec{\rho}+1}$
with constraint : $\vec{u} \cdot \vec{\alpha}_{\mathcal{T}} + \vec{v} \cdot \vec{\iota}_{\mathcal{T}} + \vec{w} \cdot \vec{p} + C \geqslant 0$,
$\vec{u} \in \mathbb{Z}^{\dim \vec{\alpha}}, \vec{v} \in \mathbb{Z}^{\dim \vec{\iota}}, \vec{w} \in \mathbb{Z}^{\dim \vec{p}}, C \in \mathbb{Z}$

Effect: $(\mathcal{T}' \bigcup \mathcal{T}'') \mapsto \mathcal{T}$;
$\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho}} \vec{\beta}_{\mathcal{T}} \leftarrow \vec{\beta}_{\mathcal{T}'}$;
and normalize β-vectors using algorithm described before in Figure 26.

squashes immediately following statements with β-prefix $\vec{\rho}$ scheduled identically by a disjoint pair of relations, replacing it with a single relation.

GRAIN ($\vec{\rho}$, i, g)

Preconditions: $\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho}}, 1 \leqslant i \leqslant \dim \vec{\rho} < \dim \vec{\beta}_{\mathcal{T}}$ and $g \in \mathbb{N}, g > 1$

Effect: $\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho}}, \forall$(in)equation $\left(\vec{u} \cdot \vec{\alpha}^{T} + af(\vec{\iota}, \vec{p}) + C \geqslant 0\right) \in \mathcal{C} : \vec{u}_i \neq 0$, replace this (in)equation with
$\left(g\vec{u} \cdot \vec{\alpha}^{T} - (g-1)\vec{u}_i\vec{\alpha}_i + g \cdot af(\vec{\iota}, \vec{p}) + gC \geqslant 0\right)$

changes, for each statement with β-prefix $\vec{\rho}$, the number of iterations n between two consecutive executions of the statement along the $\vec{\alpha}_{\mathcal{T},i}$ dimension to $n \times g$.

DENSIFY $(\vec{\rho}, i)$

Preconditions: $\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho}}, 1 \leqslant i \leqslant \dim \vec{\rho} < \dim \vec{\beta}_{\mathcal{T}}$

Effect: $\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho}}, \forall$(in)equality $: \exists g \in \mathbb{N} : g \in \mathbb{N}, g > 1$
$\left(g\vec{u} \cdot \vec{\alpha}^{T} - (g-1)\vec{u}_i \vec{\alpha}_i + g \cdot af(\vec{\iota}, \vec{p}) + gC \geqslant 0\right) \in \mathcal{C}$,
$\vec{u} \in \mathbb{Z}^{\dim \vec{\alpha}}, C \in \mathbb{Z}$,
replace this (in)equation with
$\left(\vec{u} \cdot \vec{\alpha}^{T} + af(\vec{\iota}, \vec{p}) + C \geqslant 0\right)$

removes, for each statement with β-prefix $\vec{\rho}$, the gap between two consecutive executions of the statement along the $\vec{\alpha}_{\mathcal{T},i}$ dimension.

STRIPMINE $(\vec{\rho}, s)$

Preconditions: $\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho}}, \dim \vec{\rho} < \dim \vec{\beta}_{\mathcal{T}}; s \in \mathbb{N}$

Effect: let $i \leftarrow \dim \vec{\rho}$;
$\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho}}$,
$\mathcal{T} \leftarrow ((\uparrow_i (\mathcal{T})) \cap (s \cdot \vec{\alpha}_{\mathcal{T},i} \leqslant \vec{\alpha}_{\mathcal{T},i+1} \leqslant s \cdot \vec{\alpha}_{\mathcal{T},i} + s - 1))$
where unary operator $\uparrow_i (\mathcal{T})$ inserts a new α-dimension and a new β-dimension before the ith α-dimension

decomposes the loop at depth $\dim \vec{\rho}$, for all statement instances with β-prefix $\vec{\rho}$, into two nested loops such that, for each iteration of the first loop, the second loop iterates over a chunk of at most s consecutive iterations of the original loop.

LINEARIZE $(\vec{\rho}, i)$

Preconditions: $\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho}}, \dim \vec{\rho} < \dim \vec{\beta}_{\mathcal{T}}$;
$\exists s \in \mathbb{N}, s > 1 :$
$\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho}}, (s \cdot \vec{\alpha}_{\mathcal{T},i} \leqslant \vec{\alpha}_{\mathcal{T},i+1} \leqslant s \cdot \vec{\alpha}_{\mathcal{T},i} + s - 1) \in \mathcal{C}$;

Effect: $i \leftarrow \dim \vec{\rho}$;
$\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho}}, \mathcal{T} \leftarrow (\downarrow_i (\mathcal{T}))$
where unary operator $\downarrow_i (\mathcal{T})$ removes the α-dimension and the β-dimension before the $i+1^{th}$ α-dimension and removes all (in)equations which contain the old i^{th} α-dimension

integrates iterations of the nested loop into the host loop by extending its iteration variable span. This transformation remains *globally valid* only if applied to *implicitly defined* dimensions with non-parametric bounds, e. g. a dimension created by *stripmine*, since it effectively multiplies loop bounds.

PARALLELIZE $(\vec{\rho}, i)$

Preconditions: $\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho}}, 1 \leqslant i \leqslant \dim \vec{\rho} < \dim \vec{\beta}_{\mathcal{T}}$

Effect: Adds *independence* semantics to the ith α-dimension.

is an example of pseudo-transformation adding semantic information to the ith output dimension of scheduling relations for each statement with β-prefix $\vec{\rho}$. See Section 3.5.3 for other possible dimension

semantics. For example, pseudo-transformations may allow code generator to unroll loops or introduce vector operations.

5.2.3 *Transforming between Arbitrary Scheduling Relations in Clay*

Clay transformation set was designed to enable transforming any *globally valid* scheduling to any other *globally valid* scheduling. This transformation subsumes changing both α-dimensions, involved in equations and inequalities, and β-dimensions arbitrarily within the limits of *global validity*.

CHANGING EQUATIONS we may transform the scheduling relation to its *output* form as described in Section 3.5.5. After this equivalent transformation, each *explicitly defined* dimension will be involved in exactly one equation (its definition) of the form

$$\alpha_{\mathrm{explicit}} = \vec{u} \cdot \vec{\alpha}^{\mathsf{T}}_{\mathrm{implicit}} + \vec{v} \cdot \vec{\iota}^{\mathsf{T}} + \vec{w} \cdot \vec{p}^{\mathsf{T}} + C.$$

RESHAPE transformation allows to arbitrarily change coefficients in $\vec{v}$ and SHIFT changes coefficients in $\vec{w}$ and C. Non-zero $\vec{u}$ coefficients may appear after a SKEW by an implicitly defined dimension replacing the unique appearance of $\alpha_{\mathrm{explicit}}$ by $(\alpha_{\mathrm{explicit}} - u \cdot \vec{\alpha}_{\mathrm{implicit}})$. It suffices to take a difference between the current and the target value before applying the transformation. Note that SKEW preconditions forbid substituting an *implicitly defined* dimension by a linear form involving an *explicitly defined* dimension as this substitution turns the explicit definition into implicit. In the output form, equations arise only from explicit definitions of the output dimensions. These definitions can be arbitrarily modified with *Clay* transformations. Hence all equation coefficients can be modified arbitrarily as long as the global validity conditions are respected.

CHANGING INEQUALITIES *existence* and *uniqueness* constraints substantially reduce the possible structure of inequalities defining the scheduling relation. An arbitrary inequality makes the scheduling *conditionally valid* unless other relations in the scheduling union cover the remainder of the iteration domain. INDEXSETSPLIT transformation adds an arbitrary inequality and enforces the *existence* constraint. Sequencing such transformations allows to build an arbitrary disjoint scheduling union. COLLAPSE transformation, on the other hand, joins two disjoint parts preserving domain coverage. A combination of these transformations may be used to redistribute domain points between scheduling relations. Implicitly defined dimensions are another source of inequalities. However, the only dimensions that enforce *uniqueness* for all compatible domains, are those created to express integer division as suggested by Bastoul [16]. STRIPMINE allows to create such dimensions that do not omit or duplicate instances, and

LINEARIZE allows to remove them. Inequalities that are not involved in extra relations or dimensions render the scheduling *conditionally valid.* Hence *Clay* transformations suffice to express globally valid operations on inequalities.

Contrary to equations, *Clay* does not allow to modify a specific coefficient of an inequality. On one hand, it cannot address a single inequality in the relation in the same way as equations associated with unique explicitly-defined dimension. On another hand, such non-paired modification is likely to violate the global validity conditions, at least until the subsequent transformation restores the validity. Therefore, inequality modification in *Clay* is done by first removing the previous inequality and then introducing a new one. While this may look contradicting the "transformation primitive" principle in the transformation set design, it is a fundamental limitation of the model. An inequality cannot be expressed as a linear combination of simpler inequalities, e. g. one cannot achieve a constraint $i + j \geqslant k$ by using $i \geqslant k$ and $j \geqslant k$ constraints. Bypassing this limitation would require a way to identify and modify individual constraints.

CHANGING BETA-VECTORS statement position in the code (lexical order and nesting in loops) is encoded by β-vectors. These β-vectors, together with respective β-prefixes, form an ordered forest. Each tree in the forest defines a loop nest at the root of the SCoP. We consider individual statements outside loops to be zero-depth loop nest to maintain consistency of the notation. Nodes in the tree correspond to loops, and leaves correspond to statements. The depth of the leaf corresponds to the number of loops surrounding the respective statement. INDEXSETSPLIT and COLLAPSE transformation allow to increase or decrease the set of leaves. Leafs are duplicated or merged together given the common parent node. They can be moved around the forest by other transformations. DISTRIBUTE transformation allows to split a node at any level into two separate nodes, each of which retains part of the original node's leaves. Repeatedly applying this transformation to all nodes that have more than one child will result in the forest of degenerate trees, where each node has exactly one child. It allows to reduce the lexicographic ordering of β-vectors into the simple ordering of first components of the β-vectors and use REORDER once to establish an arbitrary total order. At this point, one may add or remove nodes using STRIPMINE or LINEARIZE. These transformations allow to control the depth of a tree. LINEARIZE lifts the statement in the loop nest removing one of the surrounding loops while STRIPMINE plunges it back by creating a new surrounding loop. Finally, FUSENEXT transformation allows to merge adjacent nodes inside a tree so that children of both nodes become children of the new merged node. Being applied to the forest of degenerate trees, it allows to establish any parent-child links. For simplicity, we

may transform a forest into a tree by adding a virtual root node and connecting it to all root nodes of the trees. This node will correspond to an empty β-vector or, in the code, to the entire SCoP. Thus same transformations apply to nodes and trees.

▷ *Clay allows to transform any globally valid scheduling into any other globally valid scheduling by changing coefficients of (in)equations that define the scheduling.*

Combining these transformations *Clay* allows to modify arbitrarily the number of leaves, nodes and the parent-child links that, together, fully define the forest. When β-forest represents a real program, all transformations are only applicable if the resulting scheduling remains *globally valid* and amenable to the polyhedral model. For example, LINEARIZE is only applicable to implicitly-defined dimensions given that it is not parametric or the previous dimension is not parametric. Otherwise, it would result in loop bounds expressed as a product of parameters, which is not representable in our relational structure.

5.2.4 *Clay Directives in Practice*

Let us consider the polynomial multiplication kernel in Fig. 43. The outer loop `L1` is not directly parallelizable because updates to array `Z` depend on both iteration variables `i` and `j`.

```
for (i = 0; i <= N; i++)      /* L1 */
  for (j = 0; j <= M; j++)    /* L2 */
    Z[i + j] += X[i] * Y[j]; /* S(i,j)*/
```

Figure 43 – Polynomial multiplication kernel

Fig. 44(a) shows the original scheduling of this kernel and the visual intuition of the new iteration domain shape for each transformation step. The dependence on `Z` may be resolved by applying a classical *skew* transformation such that one of the loops iterates over `i + j`. In *Clay*, this is expressed as $\textsc{Skew}((0,0)^T, 1, 2, -1)$. Following its definition in our formalism, this transformation modifies the scheduling relation so that $\alpha_1 = i + \alpha_2$ resulting in the new relation Fig. 44(b). The statement instances would now be executed in a different order, the outer loop iterating over different indices of `Z`. The iterations of the outer loop are now independent of each other and can be executed in any order or in parallel.

The parallelism is still not well balanced: only one statement instance is executed at the first and the $(2N+1)^{th}$ iteration while M are executed at N^{th} iteration, which will limit the efficiency of a parallel execution. Since we are free to choose an arbitrary execution order of the outer loop iterations, we may execute statement instances of the ith and $(i+N)^{th}$ iterations in one iteration of the new outer loop. Exactly M statement instances will be executed in each iteration of this loop making it well balanced. This transformation is performed

$$\text{(a)}\quad \theta_S = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to \begin{pmatrix} \beta_0 \\ \alpha_1 \\ \beta_1 \\ \alpha_2 \\ \beta_2 \end{pmatrix} \middle| \begin{array}{ll} \beta_0 & = 0 \\ \alpha_1 & = i \\ \beta_1 & = 0 \\ \alpha_2 & = j \\ \beta_2 & = 0 \end{array} \right\}$$

$$\text{(b)}\quad \theta'_S = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to \begin{pmatrix} \beta_0 \\ \alpha_1 \\ \beta_1 \\ \alpha_2 \\ \beta_2 \end{pmatrix} \middle| \begin{array}{ll} \beta_0 & = 0 \\ \alpha_1 & = \alpha_2 + i \\ \beta_1 & = 0 \\ \alpha_2 & = j \\ \beta_2 & = 0 \end{array} \right\}$$

$$\text{(c)}\quad \theta''_S = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to \begin{pmatrix} \beta_0 \\ \alpha_1 \\ \beta_1 \\ \alpha_2 \\ \beta_2 \end{pmatrix} \middle| \begin{array}{ll} \beta_0 & = 0 \\ \alpha_1 & = \alpha_2 + i \\ \beta_1 & = 0 \\ \alpha_2 & = j \\ \beta_2 & = 0 \\ \alpha_1 & \leqslant N \end{array} \right\} \cup \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to \begin{pmatrix} \beta_0 \\ \alpha_1 \\ \beta_1 \\ \alpha_2 \\ \beta_2 \end{pmatrix} \middle| \begin{array}{ll} \beta_0 & = 0 \\ \alpha_1 & = \alpha_2 + i \\ \beta_1 & = 0 \\ \alpha_2 & = j \\ \beta_2 & = 1 \\ \alpha_1 & > N \end{array} \right\}$$

$$\text{(d)}\quad \theta'''_S = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to \begin{pmatrix} \beta_0 \\ \alpha_1 \\ \beta_1 \\ \alpha_2 \\ \beta_2 \end{pmatrix} \middle| \begin{array}{ll} \beta_0 & = 0 \\ \alpha_1 & = \alpha_2 + i \\ \beta_1 & = 0 \\ \alpha_2 & = j \\ \beta_2 & = 0 \\ \alpha_1 & \leqslant N \end{array} \right\} \cup \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to \begin{pmatrix} \beta_0 \\ \alpha_1 \\ \beta_1 \\ \alpha_2 \\ \beta_2 \end{pmatrix} \middle| \begin{array}{ll} \beta_0 & = 0 \\ \alpha_1 & = \alpha_2 + i - N - 1 \\ \beta_1 & = 0 \\ \alpha_2 & = j \\ \beta_2 & = 1 \\ \alpha_1 & \geqslant 0 \end{array} \right\}$$

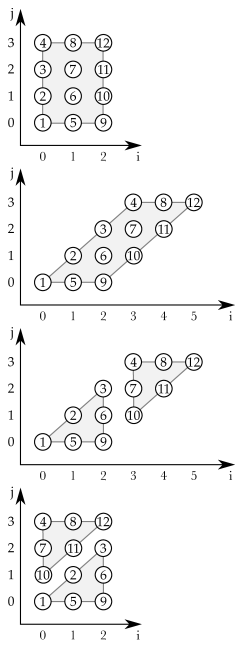


Figure 44 – Scheduling Relation Transformation of the Polynomial Multiply Kernel: (a) Original Scheduling, (b) After Skewing, (c) After Index-set Splitting, (d) After Shifting

by *index-set splitting* the outer loop iterations after N and *shifting* them by $-N$, hich is expressed in *Clay* as $\textsc{IndexSetSplit}((0)^T, \alpha_1 \geqslant N)$ followed by $\textsc{Shift}((0,0,1)^T, 1, -N)$. The former transforms the scheduling relation to a union of relations with supplementary constraints (Fig. 44(c)) while the latter transforms only the union component that has the required β-vector (Fig. 44(d)). Iterations of the outer loop are still independent and can be executed in parallel, which is allowed by the final $\textsc{Parallelize}((0)^T)$ transformation.

The final transformation sequence is presented in Figure 45. Our impementation of *Clay*[2] transformation set uses specially structured comments that precede the polyhedral code. These comments start with a keyword `/* Clay`. *Clay* interprets each line of the comment as a transformation directive. Beta-vectors and other vector-like variables are encoded as comma-separated values in brackets, e. g. `[0,0,0]`. Linear constraints are encoded as groups of comma-separated values. First group corresponds to α-dimensions, second group corresponds to input dimensions, third group corresponds to parameters and the last group with a single value is the constant. Groups are separated by pipe symbol and the entire linear constraint is enclosed in braces. As the constraints are likely to contain multiple zeros, *Clay* allows to omit trailing zeros in each group. Thus `{1,0||-1|0}` is the same as `{1||-1|}` and as `{1,0|0,0|-1,0|0}` for the IndexSetSplit in the

2. *Clay* available at https://periscop.github.io/clay

running example. This line is interpreted as $1 \cdot i + (-1) \cdot N \geqslant 0$. The comment form of the *Clay* transformation sequence is presented in the right part of the Figure 45.

$\textsc{Skew}((0,0)^T, 1, 2, -1)$
$\textsc{IndexSetSplit}((0)^T, \alpha_1 \, geq N)$
$\textsc{Shift}((0,0,1)^T, 1, -N)$
$\textsc{Parallelize}((0)^T, 1)$

```
/* Clay
   skew([0,0,0],1,2,-1);
   iss([0,0], {1,0 ||-1|0});
   shift([0,0,1],1,[-1],0);
   parallelize([0]);
 */
```

Figure 45 – Parallelizing polynomial multiply kernel with *Clay*: left - transformation sequence, right - *Clay* implementation syntax.

5.2.5 *Discussion of the Transformation Set*

SIMILAR EFFECTS Analyzing the transformation effects, one may notice that Reverse and Grain are, in fact, variations of Skew of a dimension by itself with coefficients -2 and $(k-1)/k$ where k is the grain factor. Except these cases, Skew by itself results in fractional explicit definitions violating the *logical date integrality* condition and possibly the *schedule existence* condition. At the same time, loop reversal is a well-known transformation with clear semantics that on its own makes sense as a part of the transformation set. Classical definition of Skew also does not imply skewing by itself, which, in the allowed cases, has a different effect on code than skewing by another dimension.

SKEW COMPARED TO RESHAPE A Skew transformation by an explicitly defined dimension may be expressed as a sequence of Reshape and Shift transformations using the coefficients of this explicit definition in the affine form. However, for implicitly-defined dimensions, such replacement is impossible as there is no single place with all coefficient. Therefore, Skew and Reshape should coexist in the transformation set. While we could limit the applicability of Skew to implicitly-defined dimensions, we decided otherwise for multiple reasons. First, Skew is a classical loop transformation with well-known semantics whereas Reshape is not. Second, it avoids the differentiation between explicitly-defined and implicitly-defined transformations at the level of the transformation set keeping it at a lower level. Finally, it provides a useful shortcut for "taking" the current explicit definition of dimension rather than requiring the user to analyze it and extract various coefficient.

▷ Some transformations result in equal schedulings in simple cases, but have different semantics in more intricate cases.

The difference in semantics between Skew and Reshape on the code level is that Skew primarily affects the loop bounds, e. g. the loop on i will iterate over $i + j$ instead, while Reshape primarily af-

fects the array subscripts, e. g. `A[i]` will be replaced by `A[i+j]`. In simple cases, these transformations may look identical, but they combine differently with other transformations. For example, if j is already skewed by k as $j + k$, it becomes impossible to skew i by the *original* j without resorting to RESHAPE or undoing the previous transformation.

MINIMALITY Although the presence of transformations with partially overlapping effects evidences that *Clay* transformation set is not minimal, we did not strive for minimality in the set composition. In fact, the minimal set would contain a single transformation entirely replacing the scheduling relation union with a new one, supplied as a parameter. We argue that *Clay* transformation provides a good compromise between conciseness and expressiveness: transformations have a single effect, thus easily composable, and most of them are inspired by classical loop transformations with well-known semantics.

DIMENSION INDEPENDENCE The corollary of *schedule existence* condition requires explicitly defined dimensions to be linearly independent in terms of input dimensions (otherwise, not all input dimensions are connected to the output dimensions and their iterations are lost). Thus RESHAPE and SKEW transformations only affect the dimensions involved in parameters, making these transformations easily composable in order to reach a target schedule through small modifications.

On the other hand, implicitly-defined dimensions depend on other dimensions by their nature. Defined by inequalities rather than equations, they redistribute the points from the iteration domain between different dimensions or different components of the relation union. They do not affect schedule existence if the global validity conditions are respected. At the same time, the structure of inequalities prevents decomposition of transformations affecting inequalities into smaller pieces. For example, INDEXSETSPLIT transformation may use an arbitrary complex affine splitting condition.

HIGHER-LEVEL TRANSFORMATIONS Several higher-level syntactic transformations may be obtained combining *Clay* transformations. For instance, *tiling* is a classical composition of STRIPMINE and INTERCHANGE. Iteration domain *rotation*, often used in parallelization to prevent dependences from spanning between loop iterations, corresponds to the composition of GRAIN and *Reshape* with parameters computed from the dependence vector. Although these transformations may give more freedom to the user, they do not increase the expressive power of the already complete transformation set. The

trade-off between the transformation set size and the complexity of the transformation semantics can be a subject of a separate user study.

SEMANTICS PRESERVATION AND TRANSFORMATION COMPOSITION
In order to avoid analyzing the iteration domain and performing semantics preservation verification on every step, *Clay* requires all transformations to result in globally valid scheduling relation unions by construction. This way, an intermediary state of the program during the transformation may violate the original semantics as long as it remains "verifiable" against it as ensured by validity. Therefore, transformations can be easily composed as modifications to the scheduling relation union, similarly to the linear functions. Constructing a transformation sequence where each transformation preserves semantics is not necessary with *Clay*. Existence of such sequence that transforms a globally valid scheduling into another globally valid scheduling remains an open question.

Conditionally valid scheduling relations can be incorporated in the *Clay* transformations set in two ways. On one hand, transformation directives preceding a conditionally valid transformation could be applied to the domain without composing globally and conditionally valid transformations. The transformed domain can be then checked against the validity conditions. On the other hand, *Clay* transformations may be analyzed to find their effect on validity conditions expressed as affine inequalities. For example, a SHIFT transformation would change the inequality involving the same dimension as SHIFT parameter by an inverse coefficient. In this case, all validity conditions can be propagated back to the beginning of the sequence so as to check the domain against this conditions before applying the transformation.

5.3 FROM POLYHEDRA TO TRANSFORMATION DIRECTIVES

5.3.1 *Combining Manual and Automatic Program Transformation*

Clay allows to manually request program transformations. The polyhedral engine then takes care of computing the new program schedule and implementing it by generating the source code. Even though the transformation set is largely inspired from existing practices, using *Clay* still requires analysis and decision from the user. Polyhedral model provides opportunities for automatic program optimization using optimization algorithms such as Pluto [36, 37] or iterative techniques [205, 206]. However, automatic approaches may fail due to the lack of analysis precision (caused by, e. g., over-approximation [28]) or overly simplified heuristics. Polyhedral engines operate on internal representations, such as relations or scheduling functions, that have little connection to the observable changes of the code

and their semantics. We propose to build on *Clay* transformation set to create an algorithm that, based on the original and the optimized polyhedral scheduling relation, identifies a sequence of transformation primitives that perform equivalent restructuring of the program. This sequence is understandable for and modifiable by the programmer offering control over the automatic polyhedral optimization. It also enables safe combination of manual and automatic transformation.

A brute force approach for building a transformation sequence is theoretically possible but intractable due to a combinatorial explosion: it should iterate through all possible transformation combinations of arbitrary length with all possible parameters. We leverage the following properties of the union of relations abstraction and *Clay* transformations in order to drastically reduce the search space making the search feasible:

↩ Properties enabling the search for transformation sequence.

— all modifications are contained in the scheduling relation union, statements are not created nor deleted in the polyhedral representation;
— the effects of *Clay* transformations that do not affect β-dimensions are limited to their target statements;
— the transformation set is complete thus allowing to reach any globally valid scheduling union from any other;
— effects of all transformations are undoable, e. g. by complementary transformations.

The algorithm, called *Chlore*[3] after its implementation, consists of two parts. First, it creates a sequence of FUSE, DISTRIBUTE and REORDER directives to restructure statement positions within loops by reassigning their β-vectors as described in Section 5.3.3. Second, it identifies transformations necessary to transform α-dimensions of the scheduling relation unions following the algorithm, presented in Section 5.3.4, for each statement individually as these transformations cannot affect other statements. In order to manage scheduling relation unions with different cardinality and dimensionality, it finds transformation sequences transforming both the source and the target scheduling union to a predefined *minimal* form described in Section 5.3.4. The final sequence consists of the directives transforming the source relation to the minimal form and the directives undoing the transformation of the target relation to the minimal form.

5.3.2 *Detecting Complementary Transformations*

Clay features pairs of complementary transformations that ensure undoability as shown in Figure 46. In each pair, one of the transformations has less arguments as it deduces them from its preconditions to properly undo the effect. For example, LINEARIZE transformation

3. CHunky Loop Optimization Reverse Engineering

uses a sort of pattern matching to detect inequalities implicitly defining the dimension and extract the strip-mine size from them. We leverage this property in a procedure COMPLEMENTARY that searches for transformations with smaller arity in both the original and transformed scheduling relation unions. If preconditions for the transformation with less arguments are met by the transformed relation, it means that the complementary transformation was applied to the original relation unions. The arguments of this transformation can be partially extracted from the precondition. The set of remaining arguments is small enough to allow iterating through all possible values. In order to support COLLAPSE that operates on a pair of scheduling relations, the procedure iterates over all pairs of relations instead of individual relations. As an extra precondition, only pairs of equally-dimensional relations are considered.

▷ *Some transformations can be undone by other transformations with less parameters.*

Direct	Inverse	Deducible	Others
INDEXSETSPLIT	COLLAPSE	*condition*	$\vec{\rho}$
STRIPMINE	LINEARIZE	*size*	$\vec{\rho}$, i
GRAIN	DENSIFY	*grain*	$\vec{\rho}$, i
DISTRIBUTE	FUSENEXT	i	$\vec{\rho}$

Figure 46 – Inverse transformations take only dimension index i as non-deducible parameter, if any.

As Table 46 suggests, several inverse transformation require dimension index as an non-deducible argument. We may iterate through all possible values that span from 1 to the output dimensionality of the relation. This value corresponds to the loop nest depth that rarely exceeds 10 in practical cases, making by-depth approach frequent for the polyhedral algorithms, as in ,e. g. , CLooG [16] or Candl [250].

The remaining parameter for all transformations is the β-prefix $\vec{\rho}$. In our per-statement approach, it is fixed for each scheduling relation. A β-vector of a scheduling relation can be composed from the relation itself following its definition. A β-prefix then corresponds to a prefix of the β-vector with length equal to either the current depth (dimension index) for STRIPMINE and GRAIN or to the output dimensionality for INDEXSETSPLIT and DISTRIBUTE.

5.3.3 *Aligning Relations and Matching Beta-vectors*

As discussed in Section 5.2.3, *Clay* allows to transform any β-vector to any other. However, this transformation is performed in the context of a particular SCoP since individual operations affect other statements. For example, REORDER modifies all statements and loops nested inside another loop. In order to be applicable, it requires to know the number of statements in each loop. Therefore, the al-

gorithm recovering the transformation sequence for β-vectors should operate on the entire SCoP rather than on individual statements. In fact, the algorithm relies on the β-tree structure described in Section 3.5.6.

For each pair of the source and the target β-vectors, it is necessary to identify the largest common prefix. Iterative calls to DISTRIBUTE allow to enclose the statement in question in its own loop nest. This nest can be then relocated at the new position and FUSED with other loop nests if necessary. Direct application of these manipulations could lead to infinite iteration as some REORDER transformations may undo effects of previous REORDER transformations making two mutually-undoing updates repeat infinitely. We propose a depth-by-depth recursive variant of the algorithm that avoids mutually-undoing updates. This algorithm, presented in Figure 47, operates on a given β-prefix and reassigns new values to the β-dimension immediately following the last dimension of the prefix, starting from the smallest new value. It maintains the following invariant: at a given depth, a correctly positioned statement will remain at its position. The algorithm distributes away the statements not having the same values in the original and transformed β-vector and temporarily assigns them a value larger than the current (part of the DISTRIBUTE effect). Once a statement has the correct new value of the β-dimension, it is not modified anymore because the algorithm advances to the next mismatching value that is known to be larger than the current.

▷ The algorithm proceeds depth-by-depth and keeps the statement position fixed once placed correctly.

The matching algorithm requires knowing the one-to-one mapping between original and target β-vectors for each relation. Yet scheduling unions may not have the same structure: INDEXSETSPLIT may have changed the union cardinality and STRIPMINE may have changed each relation dimensionality. These transformations should be found and applied before identifying the transformation sequence for β-vectors. We propose to analyze effects of these transformations on α-dimensions independently of the actual β-vectors that can be handled later. For each statement, we will look for COLLAPSE and LINEARIZE transformations that will minimize the cardinality and dimensionality of the scheduling relations. After all transformations have been found on both sides, the scheduling union cardinalities should become equal and one-to-one mapping between relations is established using the order of appearance of the relations in their respective unions.

5.3.4 *Generating Transformation Sequence*

The main part of the transformation sequence generation algorithm consists in transforming both the source and the target scheduling relation union to the *minimal form*. This form should have the mini-

```
Data : ρ⃗ – the current β-prefix
let d = dim ρ⃗ + 1 ;
let n = max_{T∈θ} β⃗_d where β⃗ is the β-vector of T ;
let n′ = max_{T′∈θ′} β⃗′_d where β⃗′_d is the β-vector of T′ ;
while ∃β⃗, β⃗′ : β⃗_d ≠ β⃗′_d, β⃗_d → min do
    /* split away until depth d                                    */
    for i = dim β⃗ downto d do
        Reorder(β⃗_{1...(i−1)}, put β⃗_d last) ;
        Distribute(β⃗, i) ;
    end
    if β⃗_d ⩽ n then
        Reorder(β⃗_{1...(d−1)}, put β⃗_d right after β⃗′_d) ;
        FuseNext(β⃗_{1...d}) ;
        /* split away betas that do not belong to the same
           transformed prefix                                      */
        foreach β⃗, β⃗′ : β⃗_d ≠ β⃗′_d do
            Reorder(β⃗_{1...(i−1)}, put β⃗_d last) ;
            Distribute(β⃗, i) ;
        end
    else
        Reorder(β⃗_{1...(d−1)}, put β⃗_d at β⃗′_d) ;
    end
end
/* n = n′ at this point; recurse to sub-prefixes                   */
for i = 0 to n do
    recurse with ρ⃗ ← (ρ⃗_1, ρ⃗_2, ..., ρ⃗_{dim ρ⃗}, i) ;
end
```

Figure 47 – Algorithm of the MatchBetas procedure identifying the transformation sequence for beta-vectors.

mal number of coefficients in the dimension definitions and the minimal number of dimensions. In the union of relations abstraction with global validity constraints, the only inequalities that are present in the scheduling relations correspond to IndexSetSplit or StripMine transformations. Even if the inequalities are present in both the source and the target schedulings, they can be temporarily removed and reintroduced later by means of the complementary transformations. Therefore, the *minimal form* should not feature any inequalities. For the similar reason, it may only include one relation in the union as all the others may be merged in it by *Collapse* operations. Given the absence of inequalities, all α dimensions are explicitly defined. These definitions contain the least non-zero coefficients when they

define equalities between the input and output dimensions. Thus, the scheduling relation union is said to be in *minimal form* when

↩ Minimal Form of scheduling relation.

— it contains exactly one polyhedral relation;
— this relation is defined exclusively by identity equalities between the input and the output dimensions in their respective orders;
— β-dimensions are removed.

For example the following relation is in the *minimal form*.

$$\theta_S(N, M) = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to \begin{pmatrix} \alpha_1 \\ \alpha_2 \end{pmatrix} \middle| \begin{array}{l} \alpha_1 = i \\ \alpha_2 = j \end{array} \right\}$$

The main iteration of the algorithm considers explicit dimension definitions as a matrix. With this structure, a Skew transformation correspond to the matrix row addition, Grain to the multiplication of a row by a constant, Densify to the division of the row by the largest common factor of its values and Reshape to individual value manipulation. The algorithm first tries to reduce the cardinality and the dimensionality by using Collapse and Linearize, respectively. It then uses a process similar to LU decomposition (or Gaussian elimination) in order to convert the relation to the minimal form, i. e. make the matrix diagonal. The *logical date integrality* condition, necessary for maintaining global validity, is ensured by using only integer values in the computation. In the forward pass, a matrix is transformed into a lower diagonal form by subtracting lines with coefficients. Instead of fractional coefficients, we multiply both lines by integers computed from least common multiplier of two values, subtraction of which should result in zero. For example, before subtracting the line $[4, 0, 3]$ from $[6, 7, 0]$, we multiply the first line by 3 and the second line by 2. After subtraction, we divide each line by its largest common divisor performing a Density transformation. If a subtrahend row has a zero coefficient in the target column, it is swapped with another row with a non-zero coefficient in this column, an Interchange transformation. (Note, that if there exist a column with only zeros in the matrix, it is singular and the scheduling relation most likely violates global validity conditions.) The algorithm removes the remaining non-zero coefficients in the lower triangle of the matrix by Reshape transformation. Finally, the values on the diagonal are converted to 1 by means of Densify and Reverse transformations.

▷ The main algorithm coerces both the original and the transformed scheduling into minimal by making coefficients zero with Clay directives.

Some cardinality-reducing transformations may not be applicable if two relations in the scheduling union were not changed in the same way after their creation, which is typically the reason for index-set splitting transformations. To account for this situation, the algorithm proceeds to elimination if no Collapse and Linearize transformations are currently possible, but verifies these transformations again each time another transformation has been made as shown by ↺ sym-

bol in Figure 48. This iteration restart approach allows to prioritize complex transformations by performing them as soon as possible.

When applied to the target scheduling relation union, forward transformations are performed on the relations to transform them into minimal form. Inverse transformations are recorded in the inverse order so as to apply to the original scheduling relation union once its in the minimal form in order to reach the target scheduling relation.

5.3.5 *Discussion of the Algorithm*

SEQUENCE EXISTENCE AND ALGORITHM COMPLETION The existence of a sequence of *Clay* transformations transforming any globally valid scheduling into any other globally valid scheduling is a corollary of the transformation set completeness. The completion of the algorithm from Figure 48 is based on the completion of the main loop. On every iteration of the loop, the algorithm selects one statement and either applies a cardinality/dimensionality reduction transformation or a group of transformations that set one of the coefficients to 0. The goal of the algorithm is to transform all relations into minimal form containing $2d$ non-zero coefficients (one for output and one for input dimension), where d is its dimensionality. Each iteration that does not perform a cardinality reduction transformation will set one of the coefficients to 0. It may affect other non-zero parameters with SKEW, but will not turn zero parameters non-zero: lines are subtracted from bottom to top and their last coefficients are made zero in a previous iteration. With RESHAPE, only one coefficient will be affected at it is selected to be non-zero. Thus after a finite number of iterations ($d^2 - 2d$ per relation at most), the algorithm will achieve the required number of non-zero coefficients.

▷ The algorithm makes at least one coefficient zero on each step and does not introduce non-zero coefficients.

Inequalities are not considered in elimination process and get removed by cardinality reduction transformation separately. Their coefficients are not included in the total non-zero coefficient count. When the inequalities have a specific form allowed by the global validity conditions, each cardinatlity reduction transformation removes a pair of inequalities without adding new elements to the scheduling relation union. Therefore, a finite number of cardinality reduction transformation is sufficient to remove all inequalities from the schedule. If the algorithm is unable to apply such transformation even after transforming equations to the minimal form, a special *transformation prioritization* procedure may be applied as described below.

The sequence recovery algorithm consists of two parts iterative parts, both with finite iterations. Therefore, the algorithm will eventually complete for any pair of globally valid scheduling relation unions with compatible dimensionalities.

```
transform all relations in both the source and target scheduling
  unions to the output form;
repeat
    restart iteration after each step with ↺;
    foreach statement S do
        foreach implicitly defined dimension d do
            Complementary(StripMine) ↺;
        end
        Complementary(IndexSetSplit) ↺;
        foreach dimension d do
            Complementary(Grain) ↺;
        end
        foreach α⃗_i, i → max, explicitly defined by
          {α_i = v⃗ · ī^T + w⃗ · p⃗^T + C} in 𝒯 or in 𝒯′ do
            foreach j ← 1..(i − 1) : v_j ≠ 0 do
                let x_j be the v_j value in the explicit definition of α_j ;
                use inverse transformation if ∈θ′_S ;
                Grain(β_{𝒯,1..i}, lcm(x_j, v_j)/v_j) ;
                Skew(β_{𝒯,1..i}, j, −lcm(x_j, v_j)/x_j) ;
                Densify(β_{𝒯,1..i}) ↺;
            end
            if v⃗_i < 0 then
                Reverse(β_{𝒯,1..i}) ↺;
            end
        end
        foreach α⃗_i, α⃗′_i both explicitly defined by
          {α_i = v⃗ · ī^T + w⃗ · p⃗^T + C} in 𝒯, 𝒯′ do
            foreach v⃗_j ≠ v⃗′_j /* loop A */ do
                Reshape(β_{𝒯,1..i}, v⃗′_j − v⃗_j) ↺;
            end
            repeat loop A for w⃗ and C with Shift ;
        end
        foreach α⃗_i, α⃗′_i both implicitly defined by
          {u⃗ · α⃗^T + v⃗ · ī^T + w⃗ · p⃗^T + C ⩾ 0} in 𝒯, 𝒯′ do
            foreach u⃗_j ≠ u⃗′_j /* loop B */ do
                Skew(β_{𝒯,1..i}, j, u⃗′_j − u⃗_j) ↺;
            end
            repeat loop B for v⃗ with Reshape ;
            repeat loop B for w⃗ and C with Shift ;
        end
    end
until no transformation happened in the loop;
MatchBetas (ρ⃗ ← ());
```

Figure 48 – Main iteration of the transformation sequence identification algorithm.

TRANSFORMATION PRIORITIZATION If a cardinality reduction transformation is blocked by a coefficient mismatch caused by another transformation, it is possible to coerce the inequalities involved in this transformation to the required form. For example, a LINEARIZE transformation requires a pair of inequalities with specific form $Cd_1 \leqslant d_2 \leqslant Cd_1 + C - 1$ with $C \in \mathbb{Z}, C = const$, but may be blocked if d_2 dimension was involved in a SKEW resulting in $Cd_1 \leqslant d_2 + d_3 \leqslant Cd_1 + C - 1$. In this case, we expect the algorithm to first perform the complementary SKEW transformation and a LINEARIZE. In case when all other transformations were performed and did not enable a LINARIZE, we may modify the algorithm so that it first applies SKEW,SHIFT and RESHAPE transformations modifying the inequality to the required form. Then reduces the relation dimensionality and continues as normal. It may potentially increase the number of non-zero coefficients, but will never increase relation dimensionality. Therefore, the algorithm still completes. Contrary to the base algorithm proposed in Figure 48, such modification *prioritizes* α-modifying transformations to cardinality reduction transformations. In all practical cases, the base algorithm was able to detect the transformation sequence correctly.

The base algorithm performs the cardinality reduction transformations as soon as possible, potentially resulting in, e. g. , more complex conditions of the INDEXSETSPLIT transformation. However, it allows to group together transformations applicable to multiple relations in the statement union thus reducing the total sequence length. In fact, INDEXSETSPLIT appears in the inverted sequence obtained from converting the target scheduling union to the minimal form. As this sequence is applied in the inverse order, INDEXSETSPLIT gets applied as late as possible after all common SKEW and RESHAPE transformation have taken place.

SEQUENCE PROPERTIES Given that each transformation in the *Clay* set is undoable, there exist an infinite amount of transformation sequences to convert one scheduling relation union into another. To demonstrate that, it is sufficient to introduce into any existing sequence an arbitrarily long subsequence of mutually undoing transformations, which will not modify the outcome. Our algorithm identifies *one* transformation sequence leading form the source to the target scheduling relation union without any guarantee on its length or composition. In many cases, the sequence will contain mutually undoing transformation pairs that result from converting to minimal form and back the relations that do not differ in the source and target scheduling unions.

▷ *Chlore provides one out of an infinite number of equivalent directive sequences.*

Rather than restraining the algorithm output, we suggest to apply a post-processing step to modify the transformation sequence once it has been computed. Having a separate step allows to develop a

more systematic view on equivalent transformation sequences and study the properties thereof. Indeed, the transformation sequences should be considered in a context of a particular task and empirically evaluated with the users. For example, the shortest sequence in terms of transformation count may not be the most understandable or the easiest to reproduce manually.

5.4 DIRECTIVE RECOVERY IN PRACTICE

5.4.1 *Recovering beta-vectors*

The $2d+1$ structure of the polyhedral scheduling is not mandatory. In practice, automatic optimization tools may diverge from this structure as long as the lexical order of statements is maintained. This divergence often results in a mixed structure between a pure $2d+1$ and a $d+1$. When a lexical order of statements is not explicitly stated by β-dimensions, it is inferred from the values of α-dimensions. In particular, if all instances of the statement `S2` are scheduled after all instances of the statement `S1`, then `S2` lexically follows `S1` and belongs to a different loop. However, *Chlore* expects schedulings to have $2d+1$ structure.

In order to reintroduce missing β-dimensions, we rely on their semantics — they represent statement order and loop nesting. Given a polyhedral domain and a scheduling relation union, we have to reconstruct the placement of the statements inside loop nests. This task corresponds to the abstract syntax tree construction that happens during code generation from the polyhedral model. Therefore, we leverage a code generation algorithm, CLooG [16], to obtain the AST. Inside CLooG, the syntax tree consists only of loop and statement nodes. It corresponds to the structure of β-tree as described in Section 3.5.6. We then apply a variant of β-normalization algorithm (see Figure 26) assuming the tree nodes are already sorted according to the actual statement order. Given the newly found β-vectors, we introduce constant dimensions with corresponding values between each pair of relation dimensions in the scheduling union.

When generating loop structure, CLooG performs loop *separation*: when a statement is executed only during some loop iterations, these iterations are separated out from the host loop into a new one. Loop separation allows to remove branching control flow from loop nests and thus improve performance. However, it may duplicate statements. For example, for a loop nest with two statements shifted with respect to each other, CLooG will generate three loops and two *occurrences* of each statement. Each statement will occur twice in the AST and thus have to distinct β-vectors while in the polyhedral scheduling it appears only once. We remove the loop separation procedure from CLooG algorithm in order to maintain the number of statement

```
for (0..2)
  S1;
for (3..6)
  {S1; S2;}
for (6..7)
  S2;
```

occurrences constant by generating guard `if` statements within loops instead of separate loops.

CLooG uses its own verification to detect scalar dimension, including β-dimension, separate from the tree construction and hardly accessible to the external user. Therefore, a preexisting β-dimension gets surrounded by a pair of new β-dimensions in our recovery procedure. After recovering the β-vectors and reintroducing them to the scheduling relations, we replace multiple adjacent β-dimensions by a single one following their common lexicographical order. This ensures that the relations with recovered β-dimension have a proper $(2d + 1)$ structure.

5.4.2 *Benchmarks*

Polybench is a set of programs suitable for polyhedral analysis, commonly used for evaluating polyhedral techniques.

We implemented the transformation recovery algorithm in *Chlore* project[4] based on the *Clay* library and transformation set. We evaluated this reference implementation on the widely used polyhedral optimization benchmark suite, Polybench[5]. For each benchmark, the initial polyhedral representation was obtained using *Clan* [20], version 0.7.1. This tool produces OpenScop descriptions of the programs that respect global validity conditions. We modified the Pluto optimizer [38], version 0.11.1, so that it not only accepts OpenScop descriptions, but outputs the transformed program in the same format without generating the final code. Pluto was used to generated transformed versions of the benchmarks. We requested *Chlore* to recover a transformation sequence that transforms the original program to the Pluto-transformed one (*forward* direction), and another sequence that performs the inverse transformation (*backward* direction).

Our implementation successfully recovered transformation sequences for all 30 benchmarks in both directions. The resulting transformation sequence contains 24 directives on average (SD = 29), ranging from 1 to 143 directives per benchmark as shown on Figure 49. The lengths of the forward and backward transformation sequences are close to each other for all benchmarks. They feature equal means and SDs. No post-processing was applied to the transformation sequence.

Normalizing the number of transformations to the number of statements in the benchmark, the resulting sequence features a mean of 4.8 transformations per statement (SD = 5.4), ranging from 0.5 for `gemm` and `syr2k` to 22.5 for `heat-3d`.

In order to estimate the performance of *Chlore* implementation, we measured the duration of the transformation sequence recovery on Polybench. We compiled *Chlore* using `clang` version 3.8 with `-O3` optimization enabled. The test platform was a MacBook Pro laptop computer with Intel i7-4850HQ CPU operating at 2.30GHz and

4. https://periscop.github.io/chlore/
5. Polybench/C 4.1, http://sourceforge.net/projects/polybench/

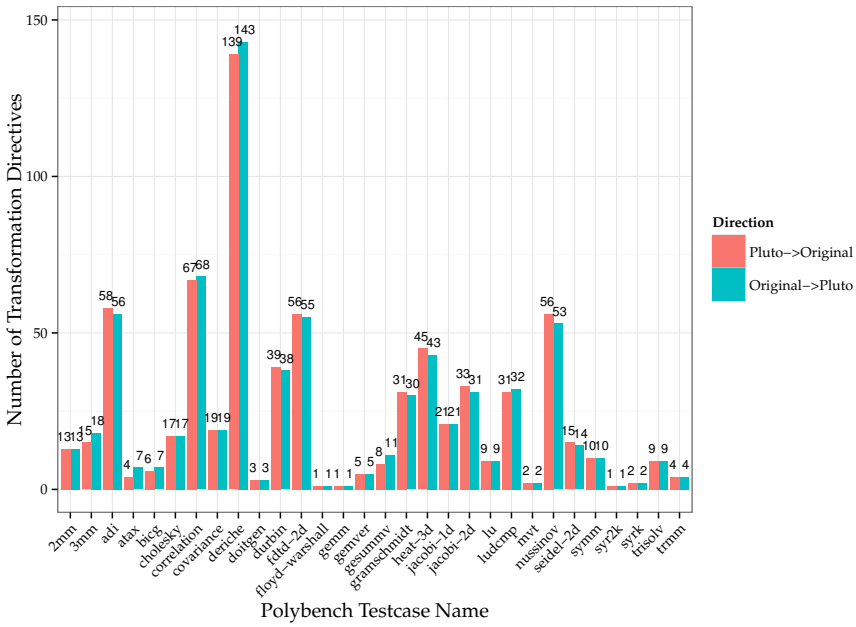


Figure 49 – Length of *Clay* transformation sequence transforming *PolyBench* tests into their *Pluto*-optimized counterparts and back.

16GB DDR3 memory, running MacOS X version 10.10.2. We measured completion time using a custom code based on `C++` standard high-resolution timer. The measurement was performed on 24 subsequent runs of *Chlore* with the same input data.

▷ *Chlore implementation is fast enough to be ran interactively in the user interface.*

The means and confidence intervals of duration for each benchmark are presented in Figure 50. Sequence recovery took on average 49 ms (SD = 63) per benchmark, ranging from 8 ms for `mvt` to 418 ms for `deriche`. Recovering the transformation sequence in forward direction was slightly longer on average (56 ms, SD = 79) than in backwards direction (41 ms, SD = 40). We also observed a significant positive correlation between the number of statements in the SCoP and the duration (Pearson's $r = 0.73$, $CI = [0.58, 0.83]$, $p < 0.001$).

These results suggest that, despite the apparent computational complexity of the proposed transformation recovery algorithm, it may be used in a dialog interaction with the developer without substantial delays. However, more scrutiny is required to evaluate the developers' understanding of long transformation sequences or transformations of individual statements in large code blocks.

5.4.3 *Example*

Let us illustrate the operation of the *Chlore* algorithm by an example. Consider the polynomial multiplication kernel in the Figure 43. Four *Clay* transformations are applied to it before paralleliza-

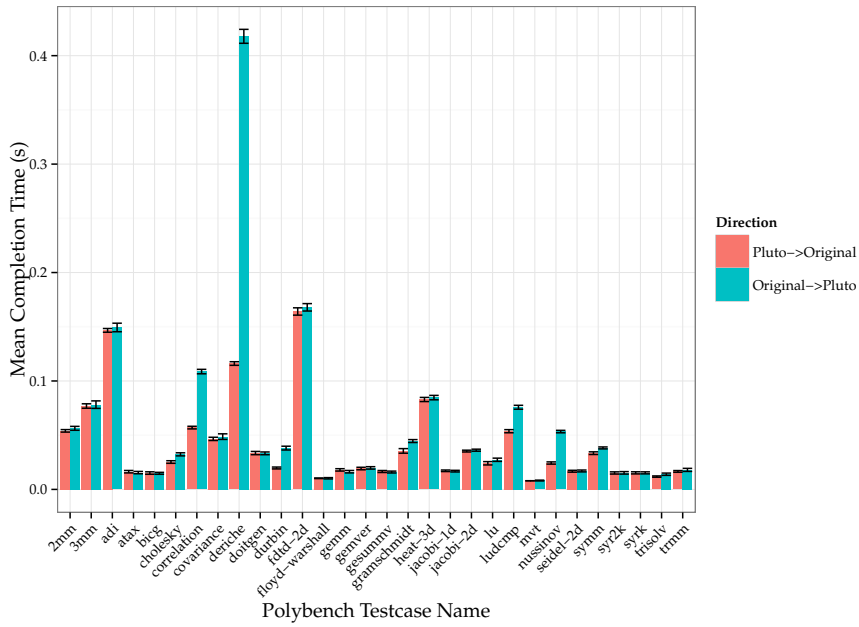


Figure 50 – Duration of transformation sequence recovery for *Pluto*-optimized *Polybench* tests using *Chlore* reference implementation.

tion. This kernel features only one loop nest with a single statement, its iteration domain is encoded in Eq. (20).

$$\mathcal{D}(N, M) = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \middle| \begin{array}{l} 0 \leqslant i \leqslant N \\ 0 \leqslant j \leqslant M \end{array} \right\} \tag{20}$$

The scheduling of the original program, extracted by *Clan* has the form of Eq. (21). After all transformations were performed, the scheduling becomes a union of relations. When converted to the *output form*, this union takes form of Eq. (22).

$$\mathcal{T}(N, M) = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to \begin{pmatrix} \beta_1 \\ \alpha_1 \\ \beta_2 \\ \alpha_2 \\ \beta_3 \end{pmatrix} \middle| \begin{array}{ll} \beta_1 & = 0 \\ \alpha_1 & = i \\ \beta_2 & = 0 \\ \alpha_2 & = j \\ \beta_3 & = 0 \end{array} \right\} \tag{21}$$

$$\mathcal{T}'(N, M) = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to \begin{pmatrix} \beta_1 \\ \alpha_1 \\ \beta_2 \\ \alpha_2 \\ \beta_3 \end{pmatrix} \middle| \begin{array}{l} \beta_1 = 0 \\ \alpha_1 = i + \alpha_2 \\ \beta_2 = 0 \\ \alpha_2 = j \\ \beta_3 = 1 \\ \alpha_1 - N < 0 \end{array} \right\} \cup \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to \begin{pmatrix} \beta_1 \\ \alpha_1 \\ \beta_2 \\ \alpha_2 \\ \beta_3 \end{pmatrix} \middle| \begin{array}{l} \beta_1 = 0 \\ \alpha_1 = i + \alpha_2 - N \\ \beta_2 = 0 \\ \alpha_2 = j \\ \beta_3 = 0 \\ \alpha_1 - N \geqslant 0 \end{array} \right\}$$

(22)

In the absence of transformations permuting statement order, the sequence recovery boils down to converting the scheduling relation unions to *minimal form* using the algorithm from Figure 48. The original scheduling, extracted from the source code, is identity. Therefore, it is already in the minimal form (see Fig. 51d). The algorithm then converts the transformed scheduling to the minimal form. Although complementary inequalities are present in two relations of θ', the COLLAPSE transformation is blocked by difference in equations of explicit definitions, namely $-N$ summand in the definition of α_1. Representing only equations in matrix form as in Figure 51a, one can easily see a mismatching -1 coefficient. In order to remove it, the algorithm executes a SHIFT$([0,0,1],[1,0],0)$ transformation, found by its *loop A*, before restarting iteration. This transformation results in identical equations in both relations as shown in Figure 51b. COLLAPSE$([0,0])$ is now possible. *Chlore* performs it on the next iteration step squashing together two relations as in Figure 51c. When iteration is restarted once again, the algorithm does not find the SKEW transformation in *loop A* as it only looks to turn the lower triangular coefficients on input dimension to zero. It rather detects a RESHAPE$([0,0,0],1,2,1)$ directive that transforms the remaining off-diagonal coefficient into zero. This transformation results in the minimal form as shown in Figure 51d. In the next iteration, no transformation happens and the algorithm falls out of the loop and terminates.

$$
\begin{array}{l}
\text{(a)} \begin{array}{c} \begin{array}{ccccccc} \alpha_1 & \alpha_2 & i & j & N & M & 1 \end{array} \\ \left[\begin{array}{cc:cc:cc:c} -1 & 0 & 1 & 1 & 0 & 0 & \boxed{-1} \\ 0 & -1 & 0 & 1 & 0 & 0 & 0 \end{array}\right] \end{array} \begin{array}{l} =0 \\ =0 \end{array} \cup \begin{array}{c} \begin{array}{ccccccc} \alpha_1 & \alpha_2 & i & j & N & M & 1 \end{array} \\ \left[\begin{array}{cc:cc:cc:c} -1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 & 0 & 0 \end{array}\right] \end{array} \begin{array}{l} =0 \\ =0 \end{array} \\
\text{(b)} \begin{array}{c} \begin{array}{ccccccc} \alpha_1 & \alpha_2 & i & j & N & M & 1 \end{array} \\ \left[\begin{array}{cc:cc:cc:c} -1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 & 0 & 0 \end{array}\right] \end{array} \begin{array}{l} =0 \\ =0 \end{array} \boxed{\cup} \begin{array}{c} \begin{array}{ccccccc} \alpha_1 & \alpha_2 & i & j & N & M & 1 \end{array} \\ \left[\begin{array}{cc:cc:cc:c} -1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 & 0 & 0 \end{array}\right] \end{array} \begin{array}{l} =0 \\ =0 \end{array} \\
\text{(c)} \begin{array}{c} \begin{array}{ccccccc} \alpha_1 & \alpha_2 & i & j & N & M & 1 \end{array} \\ \left[\begin{array}{cc:cc:cc:c} -1 & 0 & 1 & \boxed{1} & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 & 0 & 0 \end{array}\right] \end{array} \begin{array}{l} =0 \\ =0 \end{array} \\
\text{(d)} \begin{array}{c} \begin{array}{ccccccc} \alpha_1 & \alpha_2 & i & j & N & M & 1 \end{array} \\ \left[\begin{array}{cc:cc:cc:c} -1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 & 0 & 0 \end{array}\right] \end{array} \begin{array}{l} =0 \\ =0 \end{array}
\end{array}
$$

The rectangle highlights the scheduling element that gets removed or nullified on the current step.

Figure 51 – Converting transformed scheduling union into minimal form: (a) transformed scheduling; (b) SHIFT applied; (c) COLLAPSE applied; (d) RESHAPE applied, minimal form same as original scheduling.

The final sequence to turn the original schedule into the transformed one consists of the sequence converting the original scheduling to the minimal form, empty in this example, and the sequence inverse to the one converting the transformed scheduling into the minimal form. This latter sequence consists of RESHAPE and SHIFT

with negated parameters and an IndexSetSplit with inequality removed by Collapse as a parameter. The final sequence is presented in Figure 52.

```
// Original -> Transformed
reshape([0,0,0], 1, 2, -1);
iss([0,0], {1,0 | 0,0 |-1,0 |
    0});
shift([0,0,1], 1, [-1,0], 0);
```

(a) forward transformation

```
// Transformed -> Original
shift([0,0,1], 1, [1,0], 0);
collapse([0,0]);
reshape([0,0,0], 1, 2, 1);
```

(b) backward transformation

Figure 52 – Transformation sequences identified by *Chlore*.

The *Chlore*-recovered sequence converting the original scheduling to the transformed one (see Figure 52a) only differs from the manually applied *Clay* sequence (see Figure 43) by choice of Reshape over Skew. In fact, *Clay* version of Skew supports skewing a loop by the iteration of the *following* loop, which is not allowed in the classical loop skewing. Therefore, we opt for Reshape directive to represent such skewing. In addition, performing Skew with a triangular-shaped coefficient matrix often leads to linearly dependent explicit definitions violating global validity. Reshape allows to avoid this problem. In this particular example, Skew by α_2 and Reshape by j are interchangeable since the explicit definition of α_2 is $\alpha_2 = j$. Choosing the most understandable transformation may be user-subjective and is left and requires a separate study in future.

5.5 INTERACTING WITH A POLYHEDRAL COMPILER

COMPILER-ASSISTED PROGRAM RESTRUCTURING The combination of the directive-based program restructuring engine and the algorithm recovering the transformation sequence enables interaction between the optimizing polyhedral compiler and the developer. A typical standalone polyhedral compiler, such as Pluto or PPCG, usually operates on the source-to-source level: it takes the input code and eventually a limited set of parameters, e. g. whether to transform for parallelism, and returns the transformed code. Such source-to-source compiler raises the program to the polyhedral model, transforms it and generates the final code without any feedback to the user (see black and dashed red parts of the Fig. 53). From the user's perspective, the entire process happens as a single step with no interaction possibility.

By adding the transformation directive recovery algorithm and the transformation engine to the polyhedral framework (light blue parts on the Fig. 53), we enable the user-compiler interaction:

1. the framework suggests an automatically generated transformation sequence for the user to review;

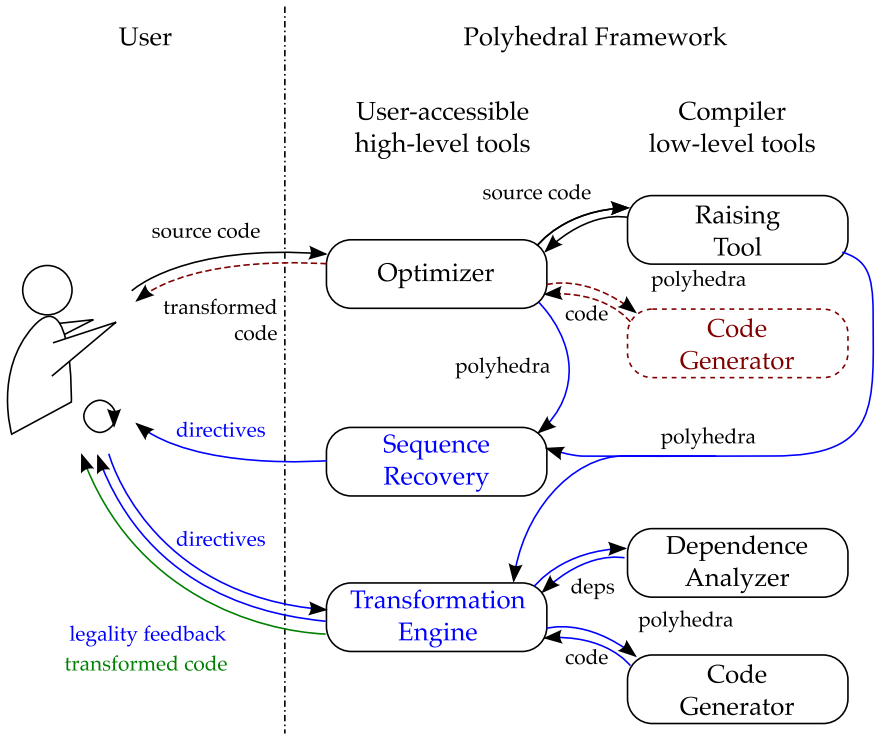


Figure 53 – Points of Interaction with the Polyhedral Framework: dashed parts were removed and lighter solid parts are added with *Clay*.

2. the user either accepts the sequence as is or modifies it before submitting to the transformation engine;
3. the latter reports on the semantics preservation (producing a list of violated dependences, if any) and generates the final code if required;
4. repeat transformation sequence modification at will.

The user is free to keep modifying the sequence, repeating the last steps and forming the interaction loop with the compiler where transformation directives and dependency graphs are used as a means for communication: the framework and the user exchange sequence of directives that transform the program. The user can also invoke the transformation engine directly, without relying on the automatic optimizer or completely ignoring the suggested transformation sequence. We thus open a way for interaction between the developer and the polyhedral compiler in order to leverage human-machine partnership in compiler-assisted program optimization.

COMPLEMENTARY APPROACHES Multiple frameworks expose a high-level directive approach on top of a polyhedral engine, including UTF [138], URUK [104], CHiLL [58] and AlphaZ [271]. However, they do not claim transformation set completeness and have more stringent restrictions on the polyhedral schedulings such as using only unimodular or invertible matrices. The completeness of *Clay* al-

lows to always recover a transformation sequence. Given a mapping between *Clay* transformation set and other directives, a transformation sequence may be expressed within another framework.

Sequence recovery algorithm may be also combined with iterative polyhedral optimization approach [205, 206] allowing to observe modifications to the transformation sequence introduced by each iteration in order to guide the space exploration.

Other directive-based tools were proposed to implement high-level loop transformations. Xlanguage [81] uses compiler pragmas for syntactic loop transformations in the C source code. A scripting language POET [269] stores and parameterizes AST-based transformations. Goofi [192] features visual interface for transforming loops in FORTRAN code. All of them operate on syntactic level and cannot benefit from precise data dependence analysis provided by the polyhedral model. With *Clay*, we are able to extract syntactic transformations from the polyhedral optimization engine and introduce them to the existing syntax-based or mixed polyhedral/syntax approaches [225].

FEEDBACK AND INTERACTION Many other tools provide the developer with the feedback about the compiler optimization stages. The Paralax infrastructure [248] allows for completing compiler analyses by manual code annotations and uses a specially modified optimizing compiler to suggest them. Larsen et.al. [158] instrument a production compiler to provide feedback as to why an automatic loop parallelization was not possible, for example highlight a statement having a loop-carried dependence. Jensen et.al. [129] propose an IDE plug-in incorporating feedback from compiler optimization passes. They enrich the IDE warning mechanism with hints like "this function can be inlined" or "interchanging these loops will increase locality and performance". A tool by Göhringer and Tepelmann [117] gives feedback on the fitness of compute-intensive loops to the polyhedral representation. Prospector [140] uses dynamic profiling to discover loop-level parallelism and provide feedback about the loop execution with particular input data. Pareon[6] provides multithreaded profiling and parallelism problems feedback along with automated OpenMP pragma generation, yet does not perform legality checks.

These tools are limited to providing *feedback* keeping the burden of time-consuming and error-prone code modification on the developer. On the contrary, we propose to manage both the feedback and the user *input*. The user analyzes and expresses a program restructuring using concise syntax while the semantics preservation is verified and the final code is generated by the framework.

6. https://www.vectorfabrics.com/en/pareon/profile

5.6 CONCLUSION

Automatic polyhedral optimization tools operate as a opaque black box providing no control or details about the process. In this section, we proposed a set of high-level program transformation directives that allow the user to leverage the power of the polyhedral framework without using its intricate mathematical structure. The *Clay* transformation set is a result of a common work with L. Bagnères, published in [12]. It is complete given global validity, i. e. allows to transform any globally valid scheduling into any other globally valid scheduling. Furthermore, we designed an algorithm allowing to express an arbitrary polyhedral relation union modification with *Clay* directives. This algorithm could identify transformation sequences for all PolyBench test cases.

Combining *Clay* transformations and *Chlore* algorithm, we virtually open the polyhedral black box enabling the developer to interact with an optimizing compiler. Directive-based approach simplifies program transformation, and transformation recovery leverages heuristic-based algorithms to suggest transformation recipes. However, the directive-based approach alone has three major challenges:

— target loop identification throughout transformation sequence;
— discovery of optimization problem or opportunity;
— estimation of the effect made by a single transformation.

Clay transformation directives are targeted to specific statements or loops using beta-vectors. Not only these vectors are hardly recoverable from the code by a developer, but they change throughout the transformations sequence. The transformation sequence recovery algorithm provides the caller with the list of transformations that were applied but not the *reasons* why they were applied. These reasons may become apparent from detailed program analysis, difficult with the source code without special tools. Program visualization techniques may help overcome these challenges by replacing target identification with direct manipulation and leveraging visual pattern identification skills for transformation discovery and effect estimation.

6 POLYHEDRAL PROGRAM RESTRUCTURING

6.1 AUGMENTING DIRECTIVE-BASED RESTRUCTURING WITH DIRECT MANIPULATION

The visualization approach for polyhedral programs, proposed in the Chapter 4, allows to address the challenges of the directive-based program restructuring. Manipulating a program represented visually requires the possibility to interact with this representation. This interaction can be then mapped to the restructuring directives in order to modify the program. Finally, the visual representation is updated to reflect the changes.

We opt for the direct manipulation approach [227] for interacting with *Clint* visualization. In fact, this visualization features program statements and their instances as on-screen objects (polygons and points) that allow for direct manipulation in a similar way to graphic editors. Furthermore, many classical loop transformations featured in *Clay* set are inspired by graphical actions, for example loop skewing or shifting. Finally, direct manipulation allows to maintain the visualization consistent throughout the program transformation: given a full mapping between visual transformations and transformation directives, we can generate a directive sequence transforming the program so that it matches the visualization updated by the user.

Direct manipulation greatly simplifies selection of the transformation targets. The user selects the transformation target directly, e. g. by clicking on the polygon corresponding to the program statement, instead of specifying the statement β-vector. Similar interaction can be enabled in the code view by coordinating selections across views [217], the statement is selected as the target for the following transformation by clicking on it in the code. If the developer analyses the program visually, he may manipulate the same representation directly rather than moving to the code. In addition, visual representation allows to manipulate statement instances or groups thereof, which are not accessible in the code.

Program restructuring for loop-level parallelism exposure is a matter of reshaping loop iteration spaces to remove loop-carried dependences. Traditionally, data dependence analysis results in a set of dependences between statements. Polyhedral frameworks provide considerably more information thanks to instance-wise analysis. However, this instance-wise information is difficult to represent without statement instances themselves and is often compacted to, e. g., dependence distance vectors for the code view. In the visual represen-

tation, instance-wise dependences are immediately visible and manipulable. They can be represented entirely or as a pattern allowing the developer to reason about performance problems and optimization opportunities using their visual pattern identification capabilities [259]. For example, scalar dependences, such as those caused by reductions, can be easily identified visually and distributed in a separate loop nest. The visualization allows to justify most reshaping transformations (skewing, shifting) by providing immediate feedback on how the data dependences are affected.

In case of scalar dependence, all loop iterations depend on each other. Therefore the loop is not directly parallelizable. Reductions are special cases of scalar dependences caused by associative operations, e.g. summing up values, that can be executed in parallel with special modifications.

Interactive visualization allows to smoothly perform step-by-step program restructuring and observe the effect of each individual transformation. The visual transformation made by the user may to perform each transformation may be recorded and replayed later. More generally, the mapping between visual transformations and *Clay* directives may be used not only to recognize user input, but also to illustrate transformations using animated transitions. These transitions can be used for training and tutorials as well as a be a part of undo/redo mechanism. They can be also turned into a static visualization as a part of interactive history view.

Once all transformation directives are mapped to animated transitions, we can integrate *Chlore* algorithm in *Clint* to visually replay the automatically computed transformation sequence entirely or step by step. This replay will give the developer better understanding of the reason for each transformation and its effect as well as the possibility to correct it when needed.

6.2 DIRECT MANIPULATION OF STATEMENTS AND INSTANCES

6.2.1 *Mapping Transformation Directives to Graphical Actions*

The scatterplot-like visualization in *Clint* maps program execution properties, such as loop bounds and traversal order, to spatial positions of polygons. This geometrical representation of program statements and statement instances affords direct manipulation as a way to apply program transformations. The user may manipulate the position and the shape of the polygon and thus change the schedule of the statement. Individual points are also manipulable in order to reschedule individual instances (executions) of a statement. Given the set of *Clay* transformations, we were able to map direct manipulation operations to high-level program transformations encoded in the polyhedral model. One operation corresponds to a sequence of transformation directives that, if applied to the program, would change its structure so that its *Clint* visualization matches the displayed one. Since the *Clay* transformation set is complete, providing a consistent mapping from graphical actions to directives allows the user to perform any polyhedral program transformation from within *Clint*.

▷ Graphical manipulation is mapped to a transformation that, if applied, would change the program so that its visualization matches the displayed one.

The mapping leverages the intuition behind classical transformations. A statement polygon can be dragged inside the coordinate system and thus *shifted* in space as shown in Figure 54(a). If the user drags the corner of the statement polygon, it is either *skewed* or *reshaped* depending on which transformation can express the transformation. SKEW is preferred to RESHAPE if both are possible as it is a more common, classical transformation. An example of SKEW transformation is depicted in Figure 56(a).

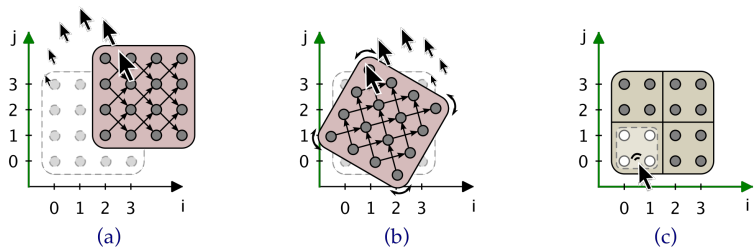


Figure 54 – Direct program manipulation in *Clint*: (a) drag to SHIFT; (b) rotate using corner handles to INTERCHANGE; (c) select and double click to StripMine, TILE or LINEARIZE if everything is selected.

Complementary transformation pairs are mapped to similar graphical actions. Expanding a polygon using side handles corresponds to GRAIN transformation and shrinking it corresponds to DENSIFY as shown in Figure 55(a). While shrinking a polygon, the user may continue dragging the handle in the same direction to trigger "negative" grain. Semantically, this corresponds to REVERSE transformation followed by a GRAIN, see Figure 55(b).

Connecting visualization to direct manipulation, we re-encounter similarities between some transformation directives. Both GRAIN and REVERSE are similar in behavior to SKEW, and all the manipulations change the polygon shape. Contrary to SKEW that changes the angles between polygon edges since the transformation involves two dimensions, GRAIN maintains the angle and only changes the distance between points. In order to underline the similarity, *Clint* uses similar graphical operations for these transformations. They all require dragging handles placed around the polygon.

Other transformations may be seamlessly combined using *Clint* direct manipulation approach. Statement may be *shifted* along two dimensions in the same action, see Figure 54(a). Rotating the statement around its center using corner handles corresponds to a combination of *loop interchange* and *reversal*. INTERCHANGE alone would require mirroring the statement along the coordinate system diagonal.

Transformation directives that modify statement execution order are especially easily combined. Dragging the statement outside the coordinate system corresponds to *loop distribution* as in Figure 56(c). If the statement being distributed is released on an empty space, a

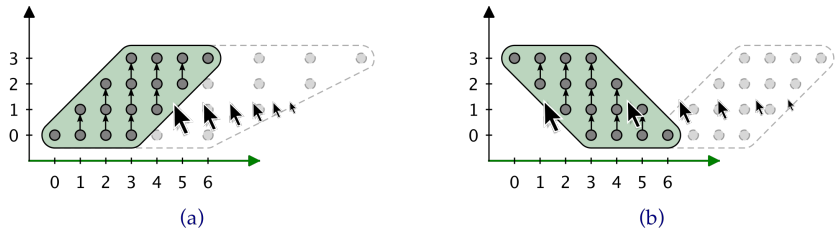


Figure 55 – Direct program manipulation in *Cilnt*: (a) expand or shrink using mid-border handles to GRAIN, DENSIFY, or (b) Reverse.

new coordinate system is created to match to the newly created loop in the program. If, however, it is released over a coordinate system, it is *fused* with the corresponding loop as depicted in Figure 56(b). Since FUSENEXT directive in *Clay* allows to fuse only with immediately following loop, *Clint* inserts a REORDER directive to appropriately place the new loop nest before fusion.

Multiple *Clay* transformations are defined only for loops and are not applicable to individual statements in order to preserve global validity. *Clint* allows to apply such transformations to individual statements in the loop by creating a transformation sequence that, first, distributes the statement away from the loop nest, then performs the requested transformation and, finally, fuses it back to the nest.

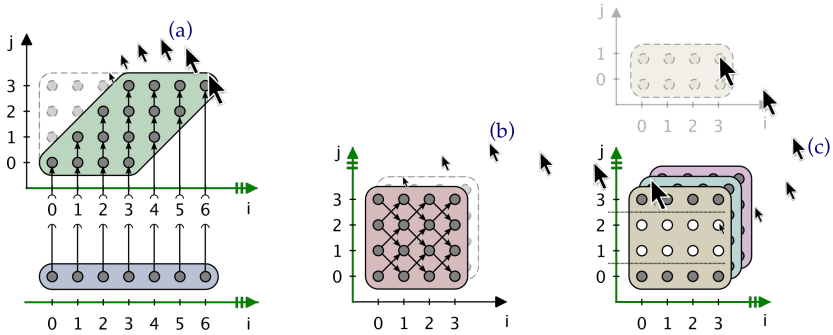


Figure 56 – Direct program manipulation in *Clint*: (a) drag polygon corners SKEW and RESHAPE; (b) move polygons between coordinate systems to change lexical order of statements with FUSE, SPLIT and REORDER; (c) double click the rectangular point selection for TILE.

▷ *Individual instances, statements or groups thereof are manipulable.*

The user may use rubber-band or click selection to specify a group of statement instances as a target of transformation. If the selection is a convex polygon, i. e. can be expressed in the polyhedral model, *Clint* will insert INDEXSETSPLIT directives before preforming the actual transformation. For example, a subset of statement instances may be separated into its own loop nest by dragging selected points outside the coordinate system as shown in Figure 56(c). Using index-set splitting allows to schedule parts of the iteration domain differently.

If more than one condition is necessary to perform the separation, *Clint* will create as many parts as splitting conditions. However, the parts that were not explicitly separated will be still manipulable as a single entity. The affine splitting inequalities are obtained by computing a convex hull of the selected points and using each pair of consecutive points to define a line. Lines that correspond to loop bounds are ignored.

Double-clicking a set of selected points that visually has a rectangular shape will result in *loop tiling*, see Figure 54(c). *Clint* only supports rectangular tiling as the most common variant. Other types of tiling may be introduced by defining a visual pattern recognizable in *Clint* representation.

6.2.2 *Discussion of the Mapping*

PARAMETRIC TRANSFORMATIONS As described in Chapter 4, *Clint* replaces constant parameters with user-defined values to provide a finite visualization. However, transformation directives often accept these parameters as arguments. Whenever it is the case, *Clint* tries to recover the possible parametric shape of the transformation. For example, if a statement nested in loop with $N = 6$ iterations is shifted by 6, *Clint* will issue a SHIFT by N directive. When multiple parameters are involved, it prefers the parameter involved in most inequalities with the loop iterator in question, and the alphabetically first in case of ties. The user can discard parametric argument computation by holding the ALT modifier key during the manipulation. More advanced control of parametric arguments is available in the textual representation of the transformation directives.

▷ *Parametric transformations are preferred when inferring the parameter is trivial.*

MULTI-DIMENSIONAL TRANSFORMATIONS Most transformations in *Clay* operate on one or two dimensions. Therefore, they can be expressed as graphical actions on the 2D canvas. INDEXSETSPLIT is the only transformation that may operate on more than two dimensions: its argument is an arbitrary inequality. Due to the nature of multidimensional inequalities, they cannot be specified by a combination of smaller, two-dimensional ones. In a general case, defining INDEXSETSPLIT condition graphically would require representing all dimensions simultaneously and enabling selection. It is not feasible with *Clint* visualization, nor is the multidimensional selection of numerous points convenient for the user. Therefore, *Clint* allows at most two dimensions to be involved in the INDEXSETSPLIT inequality, inferring its shape from the selected points. If a more complex condition is required, the user can enter it textually using *Clay* comments.

FORBIDDEN TRANSFORMATIONS AND AUTOMATIC COMPLETION Some graphical actions may correspond to transformations, but do

not respect the preconditions resulting in an *invalid* scheduling. For example, deforming the polygon into a linear shape along its diagonal, possible when moving handles for loop skewing, violates preconditions for both SKEW and RESHAPE as the transformed schedule will not be globally valid. Some other graphical actions may be correspond to incomplete transformations. For example, polygon expansion that only increases it size by a half, rather than by its entire width, corresponds to the GRAIN transformation with factor 0.5. We address both issues in *Clint* by providing *feed forward* about the transformation. When the user starts the manipulation, *Clint* identifies the closest valid transformation and performs it immediately. Instead of showing the result, it adds a visual *shadow* of the result, a light gray polygonal shape similar to those in Figures 55, 56. This feed forward allows to communicate the potential result to the user even before he completed the transformation. If the user stops the manipulation, the active polygon will transform into its shadow. Instead of an invalid transformation directive arguments, *Clint* chooses the closest valid ones. For partial arguments, it completes them by rounding to closest available value. As long as the user continues manipulation, the shadow reflects the transformation that will be made if the user stops manipulation at this point. We expect feed forward to increase the manipulation speed and favor exploration.

Violated dependences make the program state illegal, but remain valid.

VIOLATED DEPENDENCES As the transformation is already performed during the manipulation, *Clint* may verify its *legality* and provide more feed forward to the user. If current transformation violates certain dependences, they are highlighted in red during the manipulation. Similarly, axes that correspond to loops not carrying dependences are highlighted in green to expose potential parallelism. With this manipulation, program parallelization in *Clint* is a matter of deforming polygons so that data dependence lines remain black and become perpendicular to an axis.

MULTIPLE VIEW COORDINATION During the manipulation, all of *Clint* visualization, history and code views are kept in sync. When the polygon shadow appears, *Clint* uses CLooG to generate the corresponding code and displays it immediately, before the user completes the transformation. The history view contains the current Clay directive in its textual representation in addition to those already applied. This real-time information allows the user to observe the effect of his current manipulation on the code and analyze its details, e. g. numerical values, if needed.

EDITABLE HISTORY The history view is now filled with *Clay* directives in textual format. *Clint* allows to edit the history as a list of text items, one item per transformation. Transformations can be

undone by removing the corresponding lines or prefixing them with a # symbol to mark as a script comment. Transformations can also be reordered. Finally, the user may modify the arguments of any directive or add new directives in any point of the transformation sequence. If the directive syntax is correct, the entire sequence gets reapplied to the current program. The visual representation and the code are updated to reflect the new transformation. *Clint* also allows to export textual form of the transformation sequence as a standalone *Clay* script for reuse or to embed it in the source program in OpenScop format.

6.3 TRANSFORMATION REPLAY AND CORRECTION

6.3.1 *Undoing and Replaying Transformations*

REPLAY as the mapping between graphical actions and program transformations is bidirectional, *Clint* features transformation *replay* capabilities. A polyhedral program transformation, expressed in *Clay* framework, is mapped to the corresponding graphical action in the *Clint* visualization. For example, loop shifting corresponds to moving the polygon in the coordinate system, the same graphical action that triggers the transformation.

UNDO/REDO the mapping is preserved for user actions, transformation replays and undo/redo operations. In fact, in order to provide animated replay while undoing a transformation, *Clint* computes a complementary transformation and applies it to the program. It then removes the last transformation from the history instead of adding the newly computed one. For example, undoing a SPLIT transformation that corresponds to putting a polygon outside of its current coordinate system, results in a complementary FUSE transformation, putting the polygon back to the original coordinate system — exactly the opposite graphical manipulation.

6.3.2 *Interacting with a Polyhedral Compiler Graphically*

DEVELOPER-COMPILER PARTNERSHIP We integrated *Chlore* algorithm in *Clint* as a program comparison option. The user can compare the original version of the program and the version optimized by any external tool, e. g. Pluto or `isl` optimizer. The result of such comparison is a sequence of directives that transforms the original version into the optimized one. The transformation sequence is then included into the transformation history of *Clint*. Each transformation in this sequence is mapped to a graphical action to provide a complete visual representation of the transformation process. On one hand, the user can replay the program transformation to better under-

stand it in terms of code-level operations even when the polyhedral optimizer did not reason in terms of code statements or instances. On another hand, he can undo or modify certain transformation as well as complete the sequence by supplementary transformations of his choice.

▷ *Clint is likely to reduce both program analysis and manipulation time.*

Thus, *Clint* enables *developer-compiler partnership* using the visual representation as a medium. Essentially similar to *Clay*/*Clore* combination for high-level program manipulation, *Clint* visualizes information about iteration-wise dependences, crucial for polyhedral optimization and not available immediately in the code. Therefore, *Clint* reduces both program analysis time thanks to visualization and program transformation refactoring thanks to replay and direct manipulation. The developers now can visually analyze the effect of each individual transformation on the program and modify it if necessary.

The transformation refactoring workflow with *Clint* starts with an automatically computed optimizing transformation. This transformation is expressed as a sequence of directives and included in *Clint*. The user analyzes the result of the transformation visually and eventually finds an optimization opportunity that the over-conservative optimizer have missed, or transformation that gives a negative effect due to architectural details unaccounted for in the optimizer. He then completes the program transformation by adding extra or modifying existing transformations using direct manipulation or *Clay* scripts. Finally, *Clint* generates the transformed code on request. It can also store the transformation directives along with the original code if needed for better readability.

6.3.3 *Correcting Automatically Computed Optimization*

We illustrate the utility of compiler-assisted program transformation compared to fully automatic approach by an example. Consider the code snippet in Listing 9 taken from the conventional beamforming radar application. The loop nest apparently features some degree of parallelism. For example, instances of statements with β-vectors `[2,0]` and `[2,1]` can be executed in a parallel loop. A quick glance at the visual representation, Figure 57 shows by horizontal arrows connecting all points that the two last statements of the outer loop carry dependence and prevent parallelization.

This code [1] executes in 2.37 seconds on the test platform 6-core Intel Xeon X5650 2.67GHz architecture, GNU GCC 4.8.2 compiler with options `-O3 -fopenmp`. Given the loop-level parallelism, the user expects a speedup if the code is parallelized. At this point, instead of performing the transformation manually with directive scripts or *Clint* interface, the user decided to rely on the automatic transformation

1. With constant parameters N and M set to 1000.

```
#pragma scop
t = 0;                                                  // [0]
t_val = DBL_MIN;                                        // [1]
for (i = 0; i < N; i++) {                               // [2]*
  a_i[i] = 0;                                             // [2,0]
  a_r[i] = 0;                                             // [2,1]
  for (j = 0; j < M; j++) {                               // [2,2]*
    a_r[i] += s_r[j]*m_r[i][j] - s_i[j]*m_i[i][j];   // [2,2,0]
    a_i[i] += s_i[j]*m_r[i][j] + s_r[j]*m_i[i][j];   // [2,2,1]
  }
  val = a_r[i]*a_r[i] + a_i[i]*a_i[i];                    // [2,3]
  t = (val >= t_val)? (t_val = val, i) : t;               // [2,4]
}
#pragma endscop
```

Listing 9 – C code of the conventional beamforming radar application.

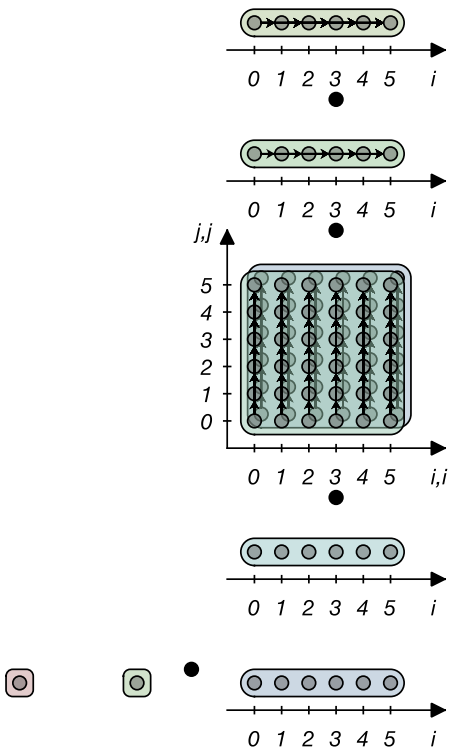


Figure 57 – *Clint* visualization of the conventional beamforming radar kernel.

engine, Pluto[2] to exploit both the parallelism and the data locality as opportunities for program performance increase. Pluto generated the optimized code listed in Listing 10. The optimized code indeed features parallel loops. Pluto has also reordered statements so that accesses to the same array were adjacent in the code. However, de-

2. Pluto 0.11.1 taken from http://pluto-compiler.sf.net

```
#pragma omp parallel for
for (i = 0; i <= N - 1; i++)
  a_r[i] = 0;
#pragma omp parallel for
for (i = 0; i <= N - 1; i++)
  for (j = 0; j <= M-1; j++)
    a_r[i] += s_r[j]*m_r[i][j] - s_i[j]*m_i[i][j];
#pragma omp parallel for
for (i = 0; i <= N - 1; i++)
  a_i[i] = 0;
#pragma omp parallel for
for (i = 0; i <= N - 1; i++)
  for (j = 0; j <= M - 1; j++)
    a_i[i] += s_i[j]*m_r[i][j] + s_r[j]*m_i[i][j];
t = 0;
t_val = DBL_MIN;
for (i = 0; i <= N - 1; i++) {
  val = a_r[i]*a_r[i] + a_i[i]*a_i[i];
  t = (val >= t_val)? (t_val = val, i) : t;
}
```

Listing 10 – Pluto-optimized C code of the conventional beamforming radar application.

spite the seemingly improved program behavior, the optimized code takes 3.32 seconds to execute on the test platform, a *slowdown*.

Without *Clint*, the user may either stick with the original sequential code or perform modifications manually trying to expose parallelism properly. With *Clint*, the user is able to analyze the code-level transformations applied by Pluto to the code and replay them visually if needed. From the Figure 58, it appears that Pluto was overly aggressive in loop distribution placing each statement into individual loop. Not only each parallel loop implicitly introduces a synchronization barrier potentially creating overhead, but it increases the data reuse distance between the accesses to arrays `a_r` and `a_i`. The analysis of the transformation replay (blue dashed lines on Figure 58) and the transformation sequence recovered by *Chlore* confirms the presence of multiple DISTRIBUTE directives, see Figure 60a.

▷ *Automatic optimization may result in slowdowns due to imprecise heuristics. Interactive program restructuring allows identifying and solving the problems.*

The user may undo the loop distributions that are potentially causing the slowdown using direct manipulation of *Clint* following green arrows in Figure 58. The transformation sequence is then completed by a series of FUSENEXT transformations as shown in Figure 60b. The new transformation sequence keeps only the distribution of the two last statements preventing parallel execution of the outer loop in the original code. Its visualization is shown in Figure 59. Thanks to the parallelism of the outer loop, this refined version executes in 0.81 seconds, with an almost 3× speedup.

Alternatively, the user may perform the manual optimization using *Clint* after having observed the optimization decisions taken by

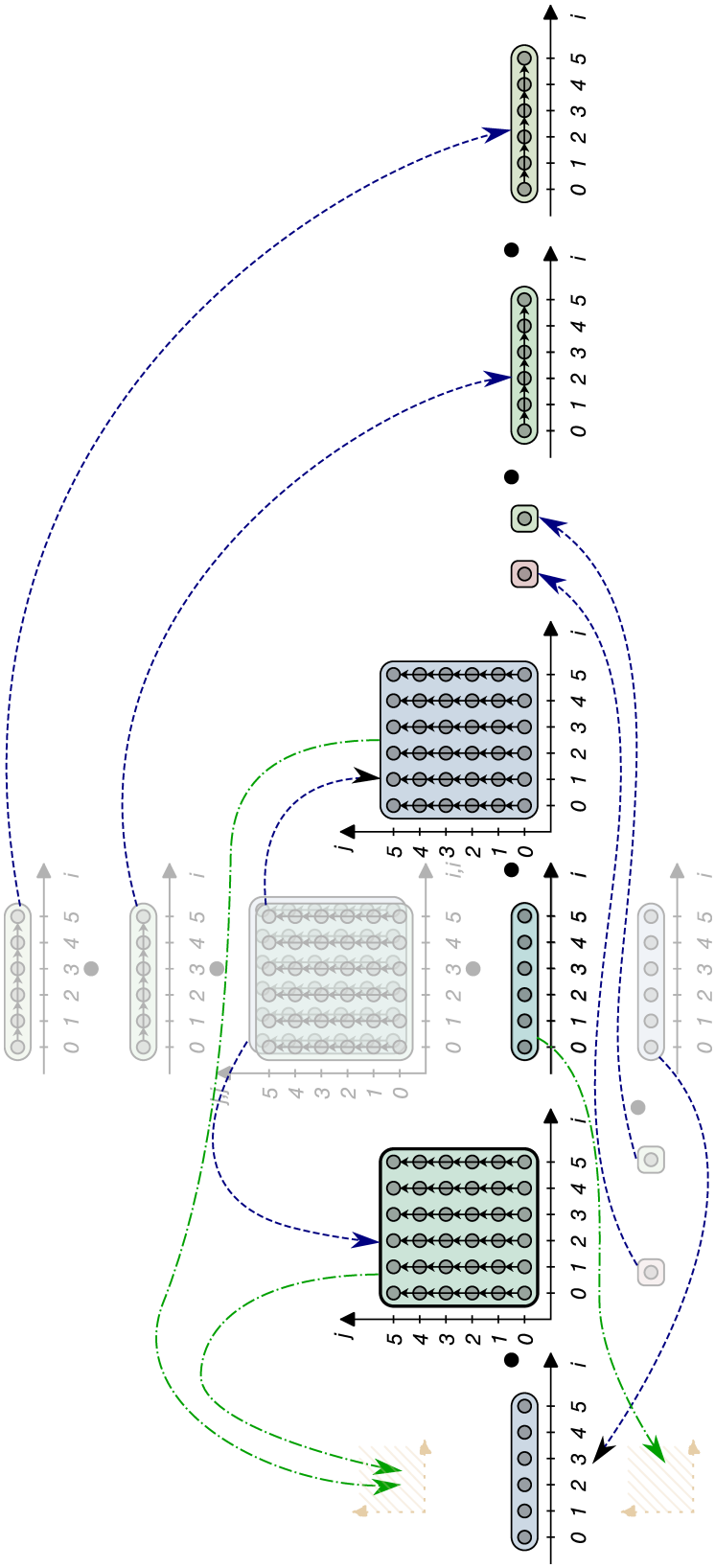


Figure 58 – Pluto-computed optimization for conventional beamforming expressed as graphical transformation primitives (blue dashed lines). Manual correction of the optimization to remove excessive loop distribution (green dash-dotted lines).

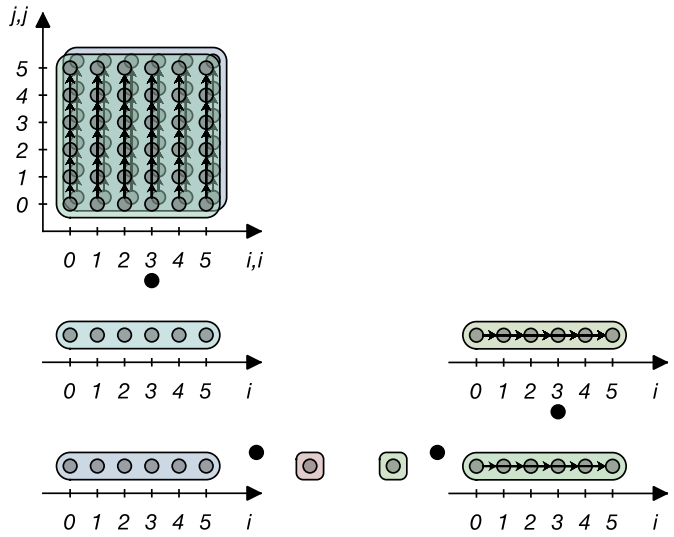


Figure 59 – The refined optimized version of the conventional beamforming radar kernel visualized with *Clint*

```
distribute([2], 3)
distribute([2], 2)
distribute([2], 1)
distribute([4, 0], 1)
distribute([4], 1)
reorder([], [4,5,2,0,1,3,6)
```

(a)

```
reorder([], [0,2,1,3,4,5,6])
fuse_next([0])
fuse_next([0])
fuse_next([0])
fuse_next([0, 2])
```

(b)

Figure 60 – Transformation sequences: (a) recovered by *Chlore* for Pluto-optimized conventional beamforming radar kernel; (b) constructed from direct manipulation in *Clint* to undo problematic loop distribution.

Pluto. To enable parallelism, it is necessary to separate the two last statements from the loop. To improve locality, the two first statements in the outer loop should be reordered. Finally, the individual statements can be placed between the two new outer loops since they are not used in the first of them. The resulting transformation sequence is shown in Figure 61.

```
distribute([2], 3)
reorder([2], [1,0,2])
reorder([], [1,2,0,3])
```

Figure 61 – Transformation sequence constructed from direct manipulation in *Clint* without Pluto input.

In both cases, *Clint* allows to achieve better program performance than automatic optimization alone. At the same time, it reduces the time necessary for program analysis and transformation application by using visualization and combining an automatically computed

program transformation with the user's input, enabling human-machine partnership.

6.4 EVALUATION OF DIRECT MANIPULATION BENEFITS

6.4.1 *Expected Benefits*

Clint features both textual and visual representations, enabling the user to transform the program from either perspective without any extra cost since the conversion between the representations is done automatically and in real time.

▷ *Clint may favor reasoning about instances and dependences, rather than about loops.*

Interactive manipulation of program statements with *Clint* is likely to ease the exploration of possible transformations, which is often tedious in the existing setup since it requires to manually transform source code and to perform semantics preservation check in order to determine the applicability of a transformation. *Clint*'s real-time feedback on parallelization and dependences may favor a trial and error strategy for exploring alternatives in order to choose the best parallelizing transformation to apply. Furthermore, the navigable and editable history view allows to easily revert or reapply any transformation.

Decoupling the manipulation from the actual program transformation also allows to postpone final source code generation in order to analyze the results of a transformation that violates dependences and to correct it without undoing, which is not possible even with semi-automatic tools. The transformed source code is then generated on-demand by the user. Contrary to existing semi-automatic tools that transform the program loops, the original program structure is kept throughout the session. This interaction model favors finer reasoning in terms of statements and instances rather than in terms of loops.

6.4.2 *Experimental Protocol*

We conducted a preliminary study to evaluate benefits of direct program manipulation with *Clint*, where participants performed program transformation visually or using the source code. With this study, we assess the usability of *Clint*. In order to clearly separate the effect of the direct manipulation and compensate for substantial individual and expertise differences, participants performed an abstracted program manipulation task without any external program analysis and transformation tools. Therefore, we chose manual code modification as a baseline for evaluating direct manipulation. The preliminary nature of this study is caused by using an initial prototype of *Clint* during the trials and feeding back comments of the participants into the tool design cycle.

Beyond the assessment of visualization efficiency, we are also interested in its acceptability by expert programmers who are more used to text-based interfaces. Participants who already took part in the visualization study presented in Chapter 4 were asked to perform parallelization tasks at several levels of difficulty and in three conditions: source code (the baseline), *Clint* without source code, and *Clint*, the latter assessing participants' preference between direct manipulation and source code editing. Our hypotheses are that *Clint* can improve programmers accuracy and efficiency when parallelizing code, but also that the direct manipulation approach is likely to change their strategy when they address a parallelization problem.

PARTICIPANTS We recruited 8 participants (5 male, aged 23-47) *via* a direct email to the participants of the polyhedral visualization study (see Section 4.5). Since all of them participated in the previous study, they were all familiar with the polyhedral model and *Clint* visualization. Therefore, they were not divided into groups by expertise.

APPARATUS The study was conducted with a prototype of *Clint*, implemented in C++, on a 15" MacBook Pro. Participants were interacting with the laptop keyboard and a standard Apple mouse. The language used was a subset of an imperative language with C-like syntax that included loops, branches and affine expressions suitable for the polyhedral modeling.

PROCEDURE The task consists in transforming a loop-based program so that the maximum number of loops becomes parallelizable, i. e. without any dependences that prevent parallel execution. Participants were asked to transform the program code, but do not include specific parallelism primitives, e. g. OpenMP pragmas, in order to avoid bias from individual expertise in various parallel frameworks. The experiment is a $[3 \times 3]$ within-subject design with two factors:

— TECHNIQUE:
  Code — code editing;
  Viz — direct manipulation without code;
  Choice — full interface, with direct manipulation and source code editing.
— DIFFICULTY:
  Easy — two-dimensional case with at most two transformations;
  Meidum — two- or three-dimensional case with rectangular boundaries and at most three transformations;
  Hard — two- or three-dimensional case with non-trivial boundaries and at least two transformations.

Trials were grouped in three blocks by TECHNIQUE. The *Code* and *Viz* blocks were presented first. Their order was counterbalanced across participants. *Choice* was always presented last in order to assess par-

ticipants' preference in using code editing or direct manipulation after having used both. In each block, participants were presented with one task of each difficulty level in random order. Tasks were different from one block to another. They were randomly picked into different blocks across participants. The tasks were drawn from real-world program examples and polyhedral benchmarks and simplified. Trials were not limited in time and participants were asked to explicitly end the trial when they thought to be done, whether they succeed or not. Prior to the experiment, participants were instructed about source code transformations and the corresponding direct manipulation techniques. They also practiced 4 trials of medium difficulty for each technique before the experiment and were allowed to perform two "recall" practice trials before each TECHNIQUE block. Each session lasted about 60 minutes. The participants filled in a short demographics questionnaire at the end.

DATA COLLECTION For each trial, we measured:

— the overall trial *Completion Time*;
— *Transform Time*, the amount of time from the start to the first change in the program structure (code edited or visualization manipulated);
— *Success Rate*, the ratio between the number of loops made parallel by transformation and the total number of possibly parallel loops.

We recorded both the final state and all intermediary transformations to the program.

During the analysis, we performed a log-transform of the *Completion Time* and *Transform Time* to compensate for positive skew in distribution typical for non-negative time interval measurements.

6.4.3 *Results and Discussion*

Because this experiment was conducted with a small sample, we decided not to conduct any statistical analysis. We report on mean values of the measures and report results graphically in order to illustrate general trends.

We did not observe any ordering effect of TECHNIQUE or DIFFICULTY on *Completion Time* and *Success Rate*.

ACCURACY AND EFFICIENCY Fig. 62(left) shows the *Success Rate* for each TECHNIQUE and in each DIFFICULTY condition. Despite large variability, it suggests that participants were in general more successful to find the expected transformations with direct manipulation than with code editing for *Easy* and *Medium* tasks: *Success Rate* is 89% for *Easy* and 100% for *Medium* tasks in *Viz* condition, but only 38% for both *Easy* and *Medium* tasks with *Code*. Thus the introduc-

tion of direct manipulation results in 40%[3] to 44% increase in *Success Rate* for these conditions. For the *Hard* condition however, *Success Rates* are identical (25%). Such low *Success Rate* for *Viz* condition suggests that, given guaranteed transformation correctness, users struggle with identification of the transformation to apply.

▷ *Visualization increased Success Rate for Easy and Medium tasks as well as Completion Time.*

We discuss *Completion Time* according the success of the trial. Figure 62(right) suggests that, for successful trials, participants performed the transformation consistently faster in *Viz* condition. The effect sizes of TECHNIQUE are 48%, 72% and 57% for *Easy*, *Medium* and *Hard* tasks, respectively. The substantial decrease in variability between *Code* and *Viz* suggests that *Completion Times* are likely to be more consistent over participants with the direct manipulation interface, compensating for individual differences in code manipulation. For the failed trials, the *Completion Times* are close for all conditions and feature large variabilities. The lower *Completion Times* for *Difficult* tasks with *Code* representation may be caused by participants quickly abandoning the most complex cases.

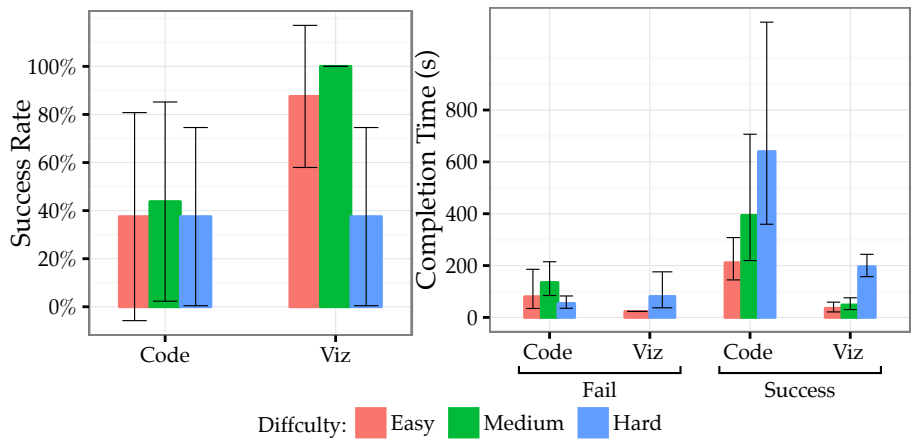


Figure 62 – Left: SUCCESS RATE is higher with *Viz* for *Easy* and *Medium* tasks, but similar to *Code* for *Hard* tasks. Right: average COMPLETION TIME is often lower with the *Viz* technique, especially when tasks were successfully performed. Error bars are 95% CIs.

STRATEGY AND EXPLORATION The ratio of tasks where participants at least tried to perform a transformation is of 76% with *Code*, against 94% for *Viz*. Additionally, we observe that the time it took to participants to start modifying the program is 135 s on average with *Code* against 13 s with direct manipulation. Combined with quick abandons of *Hard* tasks, this suggests that *Viz* condition is more likely to engage participants in solving the harder task. We computed the ratio *Transform Time/Completion Time* as a measure of "engagement" of the participants (a lower value meaning that the participant started

3. Symmetric effect sizes, computed as $\text{abs}(m_1 - m_2)/(m_1 + m_2)$ where m_1, m_2 are mean values for different groups.

to transform the program faster). As shown in Figure 63, this ratio increases with difficulty for *Code*, but drastically decreases for *Viz*. It suggests that participants were more likely to adopt an exploratory strategy for hard transformation problems with the interactive visualization than with code editing. However, when both representations are available in *Choice* condition, the ratio remains almost constant for all DIFFICULTIES, suggesting that they spend time choosing the representation or analyzing the program in a representation they will not use for manipulation.

▷ *Visual representation changed the strategy from analysis-first to exploration-first unless the code was also present.*

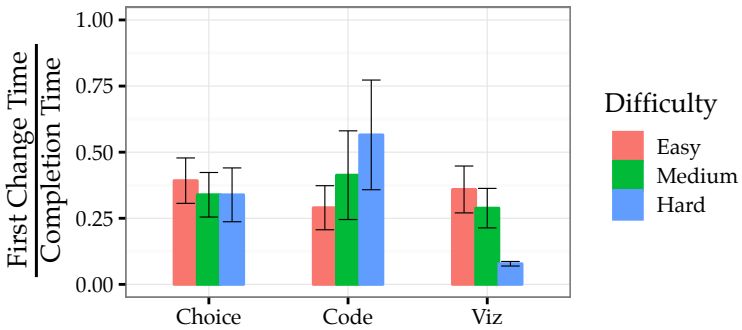


Figure 63 – Ratio of first change time to completion time. The change in trend between different techniques can be caused by an improved problem understanding and favorisation of exploring different transformations.

DIFFICULTY The choice of task DIFFICULTY was based on the visualization experiment and is essentially tailored to the *code*. In fact, the difficulty of the task is a subjective measure for the users rather than an objective property of the program. The presence of the visualization may have changed the perceived difficulty. Figure 62(left) may suggest that the *Hard* task remain complex independently of the representation. As a general observation, an efficient interactive program restructuring tool should *decrease the perceived difficulty* of program manipulation.

CHOICE BETWEEN CODE EDITING AND DIRECT MANIPULATION For the *Choice* condition, we observed that the participants used the interactive visualization and that only three of them edited the code during the first 30s of two trials on average before switching to the visual interface (12% of all the trials). In the post-experiment interview, these participants explained that they were trying out the code-visualization mapping or changing the code for the sake of analysis. We found *Success Rate* and *Completion Time* to be very similar to those with only the visualization.

We observed that most participants were examining the original and transformed source code, but not editing or selecting it. These results suggest that most users would prefer using the visual inter-

face to perform transformations, but still need the source code view to have a link with conventional program editing approach despite its evident misfit for the parallelization task at hand. While these results support the observation that expert programmers are reluctant to non-code representations, they may also be caused by the insufficient expertise in manipulating visual program representations compared to the code.

6.5 THE NEED FOR CODE

Following up on this observation, we decided to conduct a controlled study in order to observe and quantify the use of textual and visual representations in program analysis-related tasks. We are particularly interested in how programmers distribute their attention between multiple representations in cases where only one representation is specifically designed for a particular task and how the presence of other representations affects their performance. We used eye tracking in order to precisely measure the attention distribution between different representations. We expect that, given sufficient training, programmers will prefer direct manipulation of the graphical representation to code editing if there is a meaningful mapping between visual properties and program structures in the context of the task.

Our experiment software was based on *Clint*'s structure and tasks. It comprised two views of equal size: code and *Clint* visualization – that represented the same program part. We created a pair of small tasks for program analysis. Each task was designed so that either the code or the *Clint* visualization support it better, but never both. The first task consisted in verifying whether the loop nest was executed with certain iteration variable values. We expect the visual representation to be better adapted for this task. The second task consisted in observing whether the loop bounds in the nest had a specific form. We expect the code to be better adapted for this task.

6.5.1 *Protocol*

PARTICIPANTS We recruited 12 participants (9 male, aged 21-34, mean=27) *via* mailing lists. Their self-reported experience in programming was 5 to 15 years. Two of the participants were familiar with the polyhedral model and *Clint* visualizations. None of them participated in the previous studies of *Clint*. Due to the operation of the eye tracking device, we required all participants to have normal vision without correction.

APPARATUS The experimental setup consisted of a 15" MacBook Pro with Retina display (2880×1800 resolution at 86.6 pixels per centimeter) laptop running the program representation software and

SMI-ETG v1 eye-tracking system [4]. The eye-tracking system itself consists of a pair of glasses with embedded HD camera worn by the participant and connected to a laptop computer with recording software. The eye tracking is based on binocular dark pupil tracking through two front-facing cameras with infra-red highlight. The tracking device provides the angular resolution of $0.1°$ and temporal resolution of 30 Hz. The gaze tracking range is $80°$ horizontally and $60°$ vertically. The tracking system outputs a 30 FPS video recorded by the frontal camera of the glasses and a gaze position for each frame of the video.

The participant was seated 70 centimeters away from the screen. This distance was sufficient to put the entire screen in the eye tracking range. Given this distance, we can compute the linear accuracy of the tracking. We only consider the accuracy at the edge of the tracking range, at $80°$ as the upper bound. $D = 70\text{cm} \cdot \text{atan}(80°/2) - 70\text{cm} \cdot \text{atan}(80°/2 + 0.1°) = 83.42 - 83.1 = 0.32$ cm. Given the screen resolution, this corresponds to $D_p = 0.32\text{cm} \cdot 86.6\text{px/cm} = 27.7\text{px}$.

We placed bright-colored tokens on the screen edges and corners, as shown in Figure 64, to be able to track its position in the output video. Using four corners of the screen, we were able to compensate for perspective distortion of the screen in the video and recover its rectangular shape.

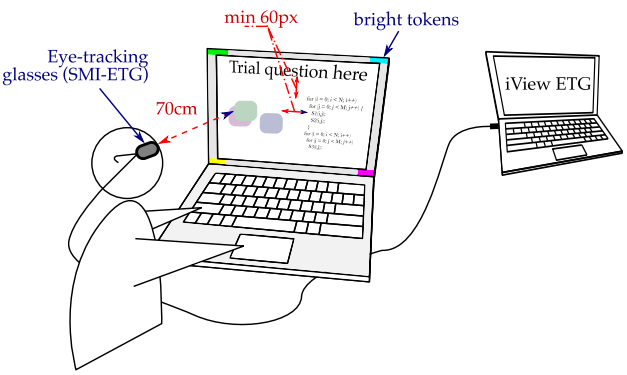


Figure 64 – Scheme of the Experiment

SOFTWARE We used a specially adapted version of *Clint* with all manipulations disabled. In the first part of the study, it featured either visual representation or code. In the second part, it featured both representations simultaneously. The sizes of all representations were identical throughout the study to ensure identical eye movement distances and equal amounts of visible information. In the first part of the study, the screen real estate not occupied by the representation was filled with default background color (gray) to avoid distraction. Visual objects and code were vertically and horizontally centered in the respective widgets. We ensured at least 60 px (0.7 cm) distances

4. http://www.eyetracking-glasses.com/

between representations, i. e. the double of eye-tracking device resolution, so as to clearly identify the gaze at one of the representations.

TASK The participants were asked to answer a binary question about a loop nest represented either visually or as code. Questions concerned either loop bounds, namely "is `i <= 5` a correct and exact upper bound for this loop nest?", or statement instances, namely "is the statement `S1(i,j)` executed with `i = 4` and `j = 7` in this loop nest?". To avoid positive answer bias, we also included negative versions of these questions, "is `i <= 5` a wrong or inexact upper bound for this loop nest?", and "is the point `i = 4` and `j = 7` outside the iteration domain of `S1(i,j)`". All questions in the study had these wordings, only the values could change.

We created an ad-hoc program to generate tasks and verify answers using the polyhedral framework. In addition to difficulty requirements, we imposed a constraint that a loop nest iterates at least 5 times and each loop in the nest iterates at least 2 times. This lower bounds allowed to avoid edge cases where a single-iteration loop could be removed from the code or a visualization would have a seemingly low dimensionality. We also imposed the upper limit of at most 100 iterations in the loop nest and at most 10 iterations for each individual loop. The upper bounds allowed participants to perform small computations, e. g. computing minima, maxima and integer divisions, without resorting to calculators.

PROCEDURE The study is a $[3 \times 3 \times 2]$ within-participants experiment with 4 repetitions per participant and the following factors:

- REPRESENTATION used in the trial, one of visual representation (*Viz*), source code (*Code*) or both simultaneously (*Both*);
- DIFFICULTY of the question, one of *Easy*, a loop nest with constant conditions, *Medium*, a loop nest with at least 3 non-constant conditions, or *Hard*, a loop nest with a branch inside and at least 5 non-constant conditions;
- QUESTION asked to the user, either focused on loop bounds (*Bounds*) or on the execution of a particular statement iteration (*Execution*).

Conditions for task DIFFICULTIES are drawn from the previous study. Namely, the presence of non-constant conditions or conditions with `min` and `max` function involved seems to impact the duration of the trial. We refined the numerical criteria for difficulty during the pilot testing. *Bounds* questions were targeted at *Code* representation, where the answer is immediately visible, while *Execution* questions were targeted at *Viz* representations. We refer to conditions with such factor combinations as *matching questions*, while we refer to other conditions as *mismatching questions*. In total, we collected data for $12 \cdot 3 \cdot 3 \cdot 2 \cdot 4 = 864$ trials.

↩ *Matching and mismatching questions.*

Trials were first blocked by REPRESENTATION and then by repetition. First two REPRESENTATIONS are *Viz* and *Code* in a counterbalanced order. *Both* condition is always the last block in order to observe the participants' choice of representation after sufficient exposure. Each REPRESENTATION block comprises 4 repetition blocks, each of which has 6 trials with different QUESTIONS and DIFFICULTIES in a randomized order. REPRESENTATION blocks were preceded by a practice session with 4 trials of *Medium* difficulty and varying QUESTIONS. After each trial in the last block, participants were also asked about their preferred representation for this trial.

First two REPRESENTATION blocks were conducted without eye tracking since they only featured one representation. Before the third block, the participants were asked to wear the eye-tracking glasses. We performed a 3-point calibration procedure using a special image with 30 px-sized elements. Before proceeding, participants were required to read aloud a short text presented on screen with the same font size as the trial questions as the code. This procedure allowed us to ensure that participant's vision was not affected by the tracking glasses and to verify the calibration.

Participants started the trial by clicking the "start" button and ended it by clicking the answer button. In each trial, participants were given a possibility to *abandon* after at least 15 seconds. The delay was imposed to avoid immediate abandons for *Hard* tasks with *mismatching* questions that we observed in the pilot test. No information about the upcoming trial was shown before its start. After the end of each trial, the software provided feedback on the answer correctness, but not the explanation of the correct answer. One session lasted 50 minutes on average. After the session, the participants filled in a short demographics questionnaire.

DATA COLLECTION We collected the following data:

Completion Time of the trial;
Correctness of the answer;
Preference between REPRESENTATIONS for the last block;
Gaze from the eye-tracking glasses for the last block.

An experimenter present in the session observed participant's behavior and their gaze reported in real time by the eye tracking system.

DATA PROCESSING The eye tracking system outputs a video recorded from the glasses frontal camera and the spatial gaze position in this video. We used a custom OpenCV-based script to track the bright tokens placed on the corners of the screen in the video. Using the positions of the tokens, we computed a quadrilateral that corresponds to the screen. We then mapped this quadrilateral to the screen rectangle and linearly interpolated the gaze position in screen coordinates. Finally, we identified the widget in the attention focus as one out of

three: *Code Widget*, *Viz Widget* or *Question Widget*. Outside any of the widget areas, the gaze was considered *Off Screen*. We randomly sampled 10 frames from each participant's video and manually checked which widget was in focus. All 120 frames were an exact match with the results of our automatic classification script.

When the participant was looking at the neutral space between two widgets, we assumed they were looking at one of the widgets first half of the time, and at another widget another half of the time. This behavior was observed for all participants. It never lasted more than 300 ms in a row and represented less than 2% of the trial duration. We did not perform any global filtering by the duration of fixation, but rather defined specific criteria for each derived measure.

We performed a log-transformation of the *Completion Time* to compensate for the positive skew of its distribution.

DATA ANALYSIS Due to growing concerns in various research fields over the limits of null hypothesis significance testing for reporting and interpreting experimental results [73, 82], we base all our analyses and discussions on estimation, i.e., effect sizes with confidence intervals [74]. Furthermore, we argue that estimation results and accompanying graphs are better suitable for communicating results in a work mixing methodologies from different disciplines. We describe the computation and interpretation of effect sizes and confidence intervals in Appendix B.

6.5.2 *Duration and Correctness*

ORDERING EFFECTS We did not find any significant ordering effect on *Completion Time* or *Correctness* between repetition blocks within larger REPRESENTATION blocks. Figure 65 shows the differences between means and their 95% confidence intervals.[5]

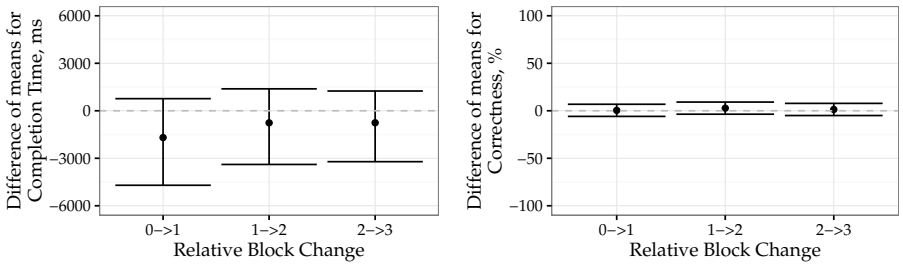
Assymertic confidence intervals are due to log-transformation of the Completion Time during the analysis.

Completion Time decreases throughout the experiment with the effect sizes[6] of -13.6% (95%CI $= [-37.7, 6.1]$) between blocks 0 and 1, -6.8% (95%CI $= [-30.1, 12.3]$) between blocks 1 and 2, and -7.2% (95%CI $= [-30.6, 11.9]$) between the last pair of blocks. Despite relative large effect sizes, their variability does not allow to observe an order effect.

Correctness slightly increases with the effect sizes of 0.5% (95%CI $= [-6.5, 7.5]$) between blocks 0 and 1, 3.0% (95%CI $= [-3.9, 9.9]$) be-

5. Confidence intervals for differences of means are computed using Tukey HSD procedure as $d \pm (q_{0.05,df,N-df}/\sqrt{2})\hat{\sigma}\sqrt{2 \cdot df/N}$, where d is the mean of differences, q is the quantile function of the studentized range distribution, $\hat{\sigma} = \sqrt{MSE}$ and MSE is the mean standard error, N is the number of observations, df is the number of degrees of freedom for a factor.

6. Symmetric effect sizes computed as $2 \cdot abs(m_a - m_b)/(m_a + m_b) \cdot 100\%$, conidence intervals scaled from those computed for means differences.



(a) (b)

Figure 65 – Differences of means of (a) *Completion Time* and (b) *Correctness* between different blocks are small.

tween blocks 1 and 2, and 1.5% (95%CI $= [-5.3, 8.2]$) between blocks 2 and 3.

COMPLETION TIME *Mismatching question* conditions, i. e. using *Code* for *Execution* questions and using *Viz* for *Bounds* questions, required substantially more time to complete the trial than *matching question* conditions, except for *Easy* tasks. With *Code*, participants spent 14% (95%CI $= [-22, 40]$) more time on *Easy Execution* questions, and respectively 132% (95%CI $= [108, 146]$) and 134% (95%CI $= [111, 147]$) more time on *Medium* and *Hard Execution* questions than on the *Bounds* questions of the same difficulty. Similarly, with *Viz* representation, participants spent 9% (95%CI $= [-20, 49]$) more time on *Easy*, 40% (95%CI $= [1, 102]$) more time on *Medium*, and 57% (95%CI $= [10, 129]$) more time on *Hard Bounds* questions than on *Execution* questions of the same difficulty. This result supports the definition of *mismatching* question suggesting that the participants had problems answering the question using the program representation not adapted to this question. The smaller increase of *Completion Time* with *Viz* compared to *Code* suggests that *Viz* representations allows to reason about mismatching question easier than *Code*.

Figure 66 shows means and 95% confidence intervals of the *Completion Time* for all conditions. We also show the density of the underlying distribution using a bipartite violin plot.

▷ *Mismatching questions required substantially more time to answer.*

Both representations show *Completion Times* close to those for *matching* representation. For *Bounds* questions, it took on average 6% (95%CI $= [-27, 56]$), −3% (95%CI $= [-40, 56]$), and 7% (95%CI $= [-33, 70]$) more time compared to *Code* for *Easy*, *Medium* and *Hard* difficulties, respectively. For *Execution* questions, it took 5% (95%CI $= [-27, 53]$), −14% (95%CI $= [-54, 54]$) and −21% (95%CI $= [-63, 58]$) more time compared to *Viz* for increasing DIFFICULTY. These results suggest that, given two representations, participants are likely to chose the *matching* one. The fact that they don't spend substantially more time than with one representation may indicate that only one of two representations is

Note the logarithmic scale of the y axis. Confidence intervals are symmertic given logarithmic scale due to log-transform.

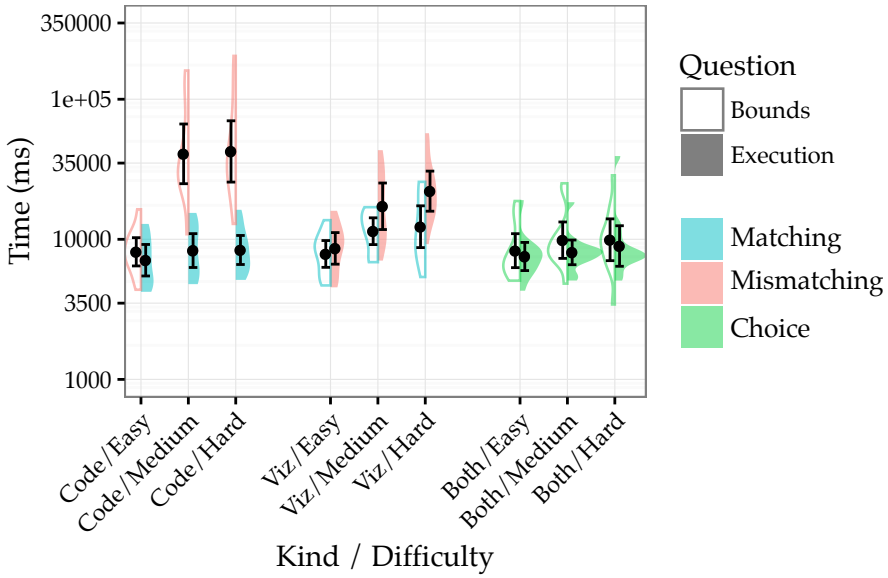


Figure 66 – Means, 95% confidence intervals and distributon densities of the *Completion Time* for all conditions. Mismatching questions require up to 4 times more time. *Completion Times* for *Both* representations are comparable to those for matching questions.

effectively used. We analyze this supposition later using eye trakcing data.

CORRECTNESS The participants succeeded to answer the majority of the questions with 93% (95%CI = [90, 95]) of correct results on average. Abandoned trials were considered incorrect answers. Figure 67 shows the means, CIs and distribution densities of the correct answers ratio for all conditions.

With *Code*, participants gave more correct answers to *matching*, *Bounds* questions (98%, 96%, 100% for increasing DIFFICULTY) than to the *mismatching Execution* questions (85%, 71%, 82%, respectively). These values result in effect sizes of 13.6% (95%CI = [1.7, 25.5]) for *Easy* tasks, 30.0% (95%CI = [11.8, 48.2]) for *Medium* tasks, and 20.7% (95%CI = [10.4, 31.0]) for *Hard* tasks. Similarly, with *Viz* representation, participants gave more correct answer to its *matching* question, *Execution* (96%, 98%, 100% for increasing DIFFICULTY). However, for *Easy Bounds* question the *Correctness* ratio was also as high as 96%. It dropped to 81% and 88% for *Medium* and *Hard* questions, respectively. These values result in effect sizes of 0% (95%CI = [−8.6, 8.6]) for *Easy* tasks, 18.6% (95%CI = [5.1, 32.1]) for *Medium* tasks, and 13.3% (95%CI = [2.6, 24.0]) for *Hard* tasks.

The 100% success rates for *Hard matching* questions suggests that participants correctly estimated the high difficulty of the task and were answering *Hard* tasks more carefully.

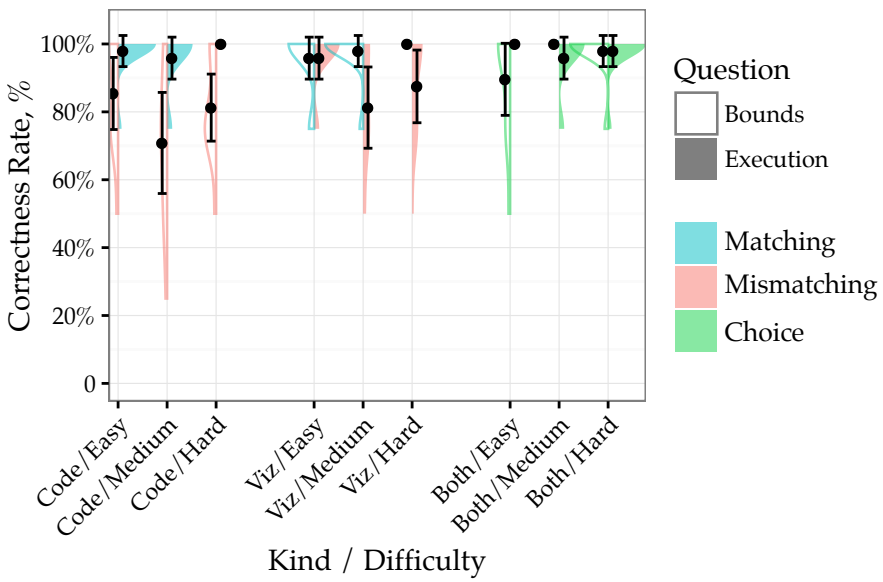


Figure 67 – Means, 95% confidence intervals and distribution densities of the *Correctness* for all conditions. *Medium* and *Hard* questions with *mismatching* representaiton result in lower correct answers ratios.

With *Both* representations available, participants generally had a high success rate for all QUESTIONS and DIFFICULTIES. The only exception is *Easy Execution* questions with the average 89.5% ($95\%\text{CI} = [79, 100]$) correctness rate, which is 11% ($95\%\text{CI} = [0.4, 21.5]$) less than the 100% *Correctness* ratio for the *Bounds* question of the same difficulty. This divergence may be due to the *mismatching* representation selection for this condition.

ABANDONS Throughout the experiment, the participants only abandoned 4 trials. All abandoned trials featured *mismatching* questions: 3 *Execution* questions with *Code* and 1 *Bounds* question with *Viz*. Two of the trials had *Medium* difficulty and two other had *Hard* difficulty. Participants abandoned tasks after long time, 91 s on average, whereas the average duration of these kinds of trials was 13.7 s. The participants felt engaged with the task and insisted on finishing the trial even when the experimenter reminded them the possibility to abandon it. They gave different reasons to continue depending on the REPRESENTATION. For *Code*, participant 2 said that "*[he was] a skilled programmer and [could] understand any code . . . given enough time*". Participants liked using *Viz* representation and found it engaging. Participant 6 stated that "*it [was] more entertaining to answer these questions with graphics*".

Given such low abandon rates, we did not perform any further analysis.

INTERACTION FOR MATCHING/MISMATCHING QUESTIONS We observed important differences in *Completion Time* between *matching* and *mismatching* questions. These conditions are defined as an interaction of REPRESENTATION and QUESTION.

Figure 68 shows the effect of REPRESENTATION-QUESTION interaction on the mean *Completion Time*. For *Easy* tasks, all REPRESENTATIONS feature close *Completion Times*. This similarity suggests that representations may be used interchangeably for easy tasks if the *Correctness* levels are the same. *Matching* question conditions shows substantially lower completion times than *mismatching* question conditions. When *Both* representations are available, mean *Completion Times* are comparable for all DIFFICULTIES.

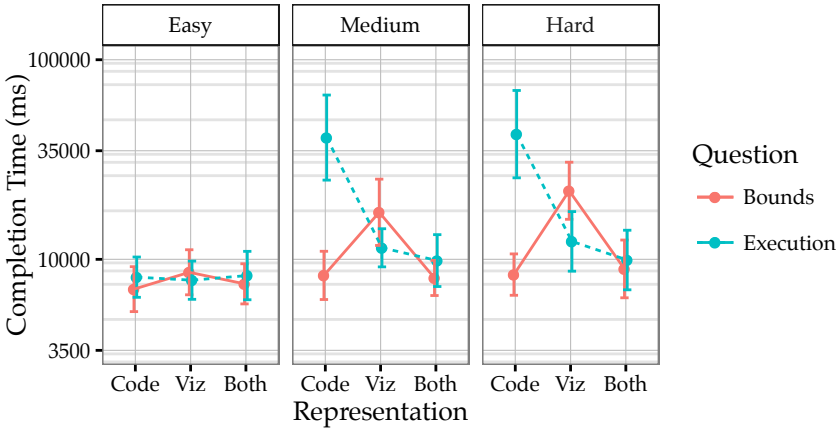


Figure 68 – Effect of interaction between REPRESENTATION and QUESTION on *Completion Time* for different DIFFICULTIES. For *Easy* tasks, *Completion Times* are similar across all conditions. For *Medium* and *Hard* tasks, participants answered faster to *matching* questions.

▷ *Completion time increases with difficulty for mismatching questions, but remains stable for matching questions.*

We also observed an effect of DIFFICULTY on the *Completion Time* with *mismatching* question conditions. For *Execution* questions with *Code*, the mean *Completion Time* increases from 8.1 s (95%CI = [6.5 s, 10.3 s]) to 40.7 s (95%CI = [24.9 s, 66.6 s]) between *Easy* and *Medium* tasks, resulting in the effect size of 133.4% (95%CI = [101.3, 150.0]). This effect is negligible, 4% (95%CI = [−90, 52]), between *Medium* and *Hard* tasks in the same condition, suggesting that the presence of complex loop boundaries makes the task hard, rather than the complexity of the conditions. For *Bounds* questions with *Viz* representation, the mean *Completion Time* increases from 8.6 s (95%CI = [6.6 s, 11.2 s]) to 17.2 s (95%CI = [11.7 s, 25.2 s]). The effect size reaches 66.6% (95%CI = [21.4, 93.5]). Between *Medium* and *Hard* tasks, the effect is larger than with *Code*, but still negligible due to large variability, 24.7% (95%CI = [−34.6, 60.1]).

For *matching* question conditions as well as with *Both* representations available, DIFFICULTY has negligible effect on the mean *Completion Time*, with effect sizes not exceeding 20%. These results suggest

that a representation properly adapted to the task is effectively alleviating the complexity of the task.

Figure 69 shows the effect of REPRESENTATION-QUESTION interaction on the mean *Correctness* rate. For *mismatching* question conditions, correctness rates are smaller for all conditions except *Easy* questions with *Viz* representation. Surprisingly, *Easy Execution* questions result in lower correctness rates with *Both* representations, 90% (95%CI $= [79, 100]$). However, the *Correctness* rates of *Both* are similar to *Code* that may be explained by the participants choosing code representation even when it is mismatching the question.

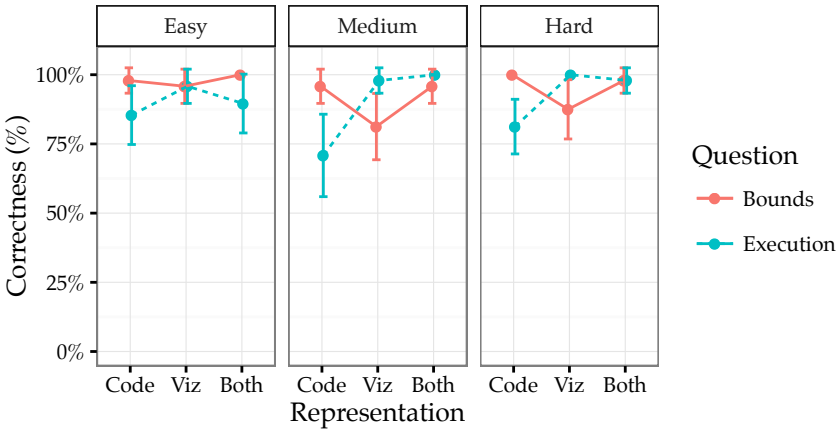


Figure 69 – Effect of interaction between REPRESENTATION and QUESTION on *Correctness* rate for different DIFFICULTIES. *Mismatching* question conditions result in lower *Correctness* rates. The effect is stronger for larger DIFFICULTIES.

▷ *Correctness rate decreases with difficulty for mismatching questions, but not for matching questions.*

The DIFFICULTY of the task has an impact on the *Correctness* rates in *mismatching* question conditions. The correctness rate drops from 85% (95%CI $= [79\%, 91\%]$) to 71% (95%CI $= [65\%, 76\%]$) when moving from *Easy* to *Medium Execution* questions with *Code*, resulting in the effect size of 18.6% (95%CI $= [5.5, 31.7]$). It also drops from 96% (95%CI $= [90\%, 101\%]$) to 81% (95%CI $= [75\%, 87\%]$) for *Bounds* questions with *Viz*, giving an effect size of 16.5% (95%CI $= [1.2, 31.8]$). Interestingly, *Correctness* rates increase by 13.7% (95%CI $= [-11.2, 38.6]$) when moving from *Medium* to *Hard Execution* tasks with *Code*.

6.5.3 *Representation Choice*

METRICS We derived the following metrics using the gaze information. They were defined prior to the study in order to observe the effects of representation choice on *Preference* and on the trial *Completion Time*. We hypothesize that the trial completion time increases if the participant uses both representations to answer this question since only one is sufficient. It may increase even more when he uses the mismatching representation.

Visual Preference, VP — duration of gaze on the *Viz* representation normalized to the duration of gaze on either *Viz* or *Code* representation. Visual preference 1 means the participant did not look at the code at all, while visual preference 0 means he only looked at the code.

Representation Uncertainty — the measure of attention distribution computed as $2 \cdot \text{abs}(\text{VP} - 0.5)$. The closer is this value to 1, the more evenly distributed is the total gaze duration on *Viz* and on *Code*. 0 representation uncertainty means that the participant only looked at one representation.

Using the eye tracking data, we quantified the distribution of participants' attention between two representations. Figure 70 shows whether their eyes were dwelling more on the visual representation (higher) or code (lower). The center line corresponds to the equal distribution of dwell duration between representations. For *Medium* and *Hard* tasks, participants looked mostly to the *matching* representation. Effect sizes reach 66.6% ($95\%\text{CI} = [4.8, 128.5]$) for *Medium* and 81.2% ($95\%\text{CI} = [16.7, 145.7]$) for *Hard* tasks. However, for *Easy* tasks they fixated more on the *Code* representation independent of QUESTION. For *Execution* questions, they only spent 39.6% ($95\%\text{CI} = [25.1\%, 54.0\%]$) of time on average on the *Viz* representation, while this ratio reaches 59.5% ($95\%\text{CI} = [36.7\%, 82.2\%]$) and 64.5% ($95\%\text{CI} = [40.8\%, 88.1\%]$) for *Medium* and *Hard* difficulty, respectively. This observation supports the earlier suggestion that participants estimate both *Viz* and *Code* appropriate for *Easy* tasks. They tend to prefer *Code* even for mismatching questions as it is a more familiar representation. This suboptimal choice results in lower *Correctness* ratio for *Easy Execution* questions as seen before.

▷ *Participants looked mostly at the matching representation for harder tasks, but preferred Code for easier tasks even if it was the mismatching representation.*

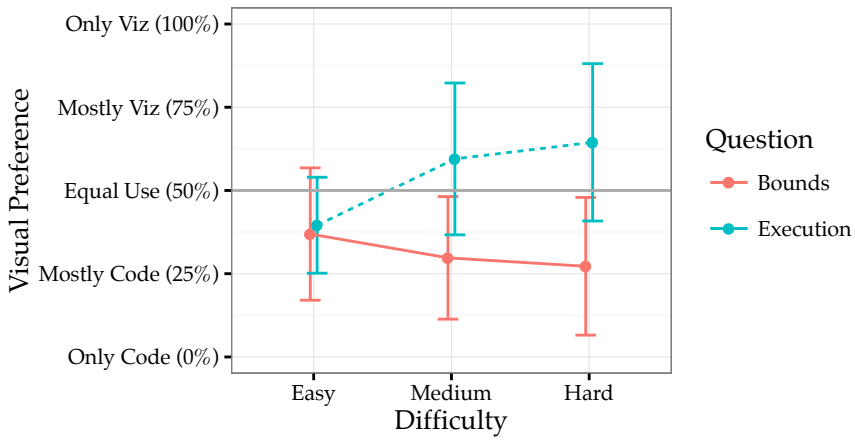


Figure 70 – Effect of interaction between REPRESENTATION and QUESTION on *Visual Preference*. *Matching* representations are more used for *Medium* and *Hard* tasks, but *Code* is more used for *Easy* tasks with both QUESTIONS

The reported representation *Preference*, depicted on Figure 71, shows the same tendency. For *Easy Execution* tasks, participants preferred *Code* in 56% cases while for *Medium* and *Hard Execution* tasks, they preferred it only in 6% cases. Given that we asked for "most useful" representation, the difference between reported *Preference* and measured *Visual Preference* suggests that participants tend to look at both representations even though they don't find one of them useful. Nevertheless, we observed positive correlation between reported *Preference* and *Visual Preference* ($r = 0.41$ ($95\%CI = [0.20, 0.57]$), $p < 0.001$), which may suggest that participants tend to use more the representation that they find most useful.

▷ *Participants preferred matching representations for harder questions.*

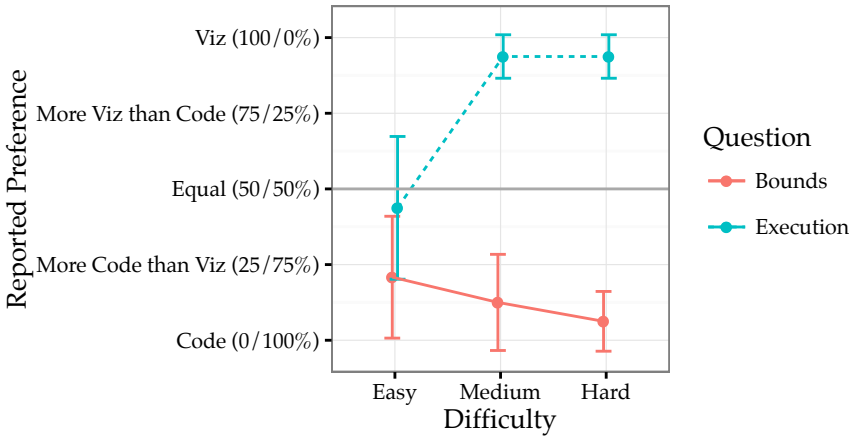


Figure 71 – Mean PREFERENCE of representation. Participants strongly preferred *matching* representations for *Medium* and *Hard* tasks. For *Easy Execution* questions, less than half preferred *matching Viz* representation.

Finally, we observed a correlation between *Representation Uncertainty* rate and the total trial *Duration* ($r = 0.41$ ($95\%CI = [0.19, 0.58]$), $p < 0.001$). We also found a negative correlation between *Representation Uncertainty* rate and the *Correctness* ($r = -0.27$ ($95\%CI = [-0.47, -0.04]$), $p = 0.023$): the more participant's attention was distributed between representations, the less correct answers they gave. Although the correlation does not allow to link these values causally, the connection between the simultaneous use of different representations and the total trial duration may suggest that one *matching* representation should be preferred to two.

6.5.4 *Visual Behavior*

During the sessions, we observed several recurring aspects of participants' visual behavior. Using the eye tracking information presented as a timestamped sequence of gaze areas, we constructed a normalized timeline of the areas using linear interpolation. We ana-

lyzed the most noticeable cases manually and defined the metrics on the timeline to quantify these aspects of behavior.

REREAD QUESTION in multiple trials, the participants reread the question before giving definitive answer. They would place the cursor over the answer they thought was correct, eventually press it without releasing, reread the question and either confirm the answer or change it to the opposite. In most cases, participants did not spend substantial time re-analyzing the representations and just changed the answer. This behavior suggests they reached understanding of the program overall, but they were uncertain about the question details. We quantify this behavior as follows. The last gaze on the question area is at least as long the first gaze on the same area. The first gaze on the question area is at least 2 seconds long, the time we observed sufficient for all participants to read the questions entirely. The last gaze should also *end* in the last 20% of the trial duration. In many trials, reading the question takes a large part of the trial duration. If reading a question once takes more than 20% of trial duration, the last gaze on question will not fully fit in the last 20%. Therefore, we require it only to end in this interval. Following our observations, we assume that most participants find the answer to the trial question during the middle interval of the trial. We also allow short (less than 20% of duration) gazes on representations after a rereading a question.

The button triggers the action if mouse is released while the cursor is still over the button.

All participants consistently demonstrated this aspect of behavior, in 17%–58% of trials. They re-read the task independent of the QUESTION, in 36.8% ($95\%\text{CI} = [28.8\%, 44.8\%]$) of trials for *Bounds* and 36.1% ($95\%\text{CI} = [28.2\%, 44.1\%]$) for *Execution*. The DIFFICULTY of the question also had a negligible effect on the number of trials with re-read.

CONFIRMATION / COMPARISON QUICK LOOK the participants took a quick look at a representation before giving the definitive answer, often after rereading the question. In the first 80% of the trial duration, representation R1 is used more than 60% of the total duration of gaze on representations. In the last 20% of the trial duration, a gaze on question area is followed by a gaze on representation R2 and back on the question again. The gaze on the representation should be at least 0.5 seconds long, the duration we observed long enough for participants to scan the visual representation. If R1 and R2 are the same representation, we refer to this behavior as *Confirmation Quick Look*. This behavior is most often combined with *Reread Question* suggesting that participants were aiming at high accuracy. Otherwise, we refer to it as *Comparison Quick Look*. This behavior can be explained by the need to verify the choice of representation prior to answering the question.

▷ *Participants used the second representation for confirmation.*

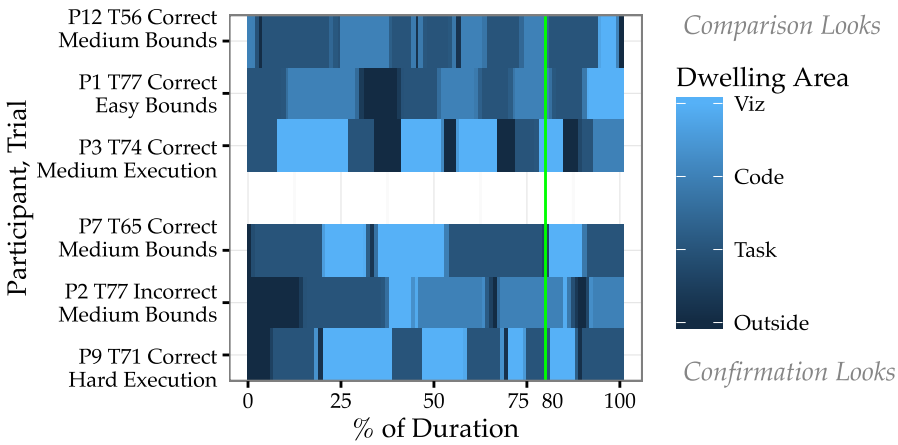


Figure 72 – Gaze timelines for selected trials featuring *Comparison Quick Look* (top) and *Confirmation Quick Look* (bottom).

All participants performed *Confirmation Quick Looks* in the experiment, in 21% to 79% of their trials, mean 48% ($95\%\text{CI} = [42\%, 54\%]$). Figure 72 shows participant's gaze target throughout the selected trials. The type of QUESTION did not seem to have an effect on the number of confirmation looks, 47% ($95\%\text{CI} = [38\%, 55\%]$) and 49% ($95\%\text{CI} = [41\%, 58\%]$) of trials for *Bounds* and *Execution*, respectively. However, *Medium* questions resulted in more confirmation looks, 39% ($95\%\text{CI} = [27\%, 48\%]$), than *Easy* questions, 56% ($95\%\text{CI} = [46\%, 66\%]$), with the effect size of 36% ($95\%\text{CI} = [2, 73]$). *Hard* questions resulted in less confirmation looks, 49% ($95\%\text{CI} = [39\%, 59\%]$), but not considerably.

START IN READING ORDER the participants may have started analyzing the representations in their usual reading order, whether it was matching the question or not. As all our participants have a cultural background of reading left-to-right, we define this metric as the first gaze on the representation that is on the left and lasts at least 0.5 seconds.

Participants started the trial by looking on the left representation in 58% ($95\%\text{CI} = [49\%, 67\%]$) cases on average. One may expect a 50% ratio since in a half of trials, the *matching* representation is indeed on the left. While the expected mean is within the confidence interval, we observed large inter-participant variability. Two of the participants started with the left representation in 83% cases, and another one in 67% cases. Nevertheless, we did not find a correlation between the ratio of trials started with left visualization and the *Completion Time* of the trial or its *Correctness*.

NUMBER OF REPRESENTATION CHANGES while *Representation Uncertainty* gives a measure of attention distribution between representations, we need a more detailed description of this distribution. For

example, whether the participants use both representations equally throughout the trial, perform one switch, or converge fast to the matching one. We compute the number gaze switches between representations, possibly passing through the question area or off-screen area. Each gaze should be at least 0.5 seconds long. This metric gives us a better understanding of how participants use both representations than *Representation Uncertainty* that just signifies they do use both representations.

▷ *Participants changed representations more often when Visual representation was matching the question.*

We observed an interesting change in representation selection. The number of representation changes for *Execution* question is large for *Easy* tasks, 1.7 (95%CI = $[0.9, 2.6]$) per trial on average, and decreases to 0.8 (95%CI = $[0.3, 1.4]$) for *Hard* tasks, see Figure 73. On the contrary, the number of representation changes for *Bounds* question is small for *Easy* tasks, 0.6 (95%CI = $[0.1, 1.0]$), and increases to 1.0 (95%CI = $[0.3, 1.8]$) for *Hard* tasks.

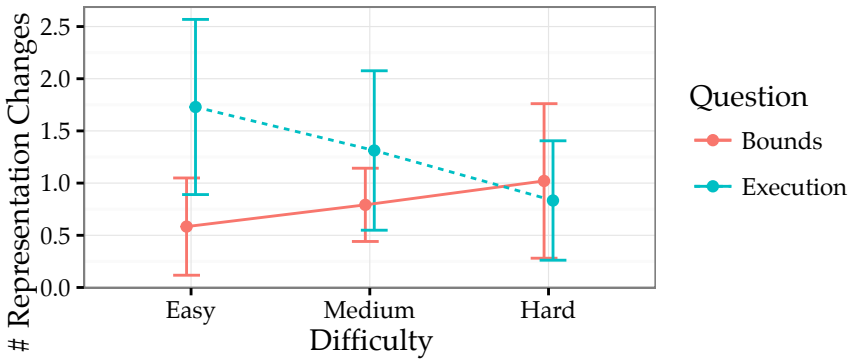


Figure 73 – The number of times participants changed representations during the trial increases with DIFFICULTY for *Bounds* questions and decreases for *Execution* questions.

Combined, these results suggest that for *Easy Execution* tasks, participants actually use both *Code* and *Viz* representation interchangeably. For *Easy Bounds* tasks they mostly use one representation, *Code*. On the other hand, for *Hard Code* questions, they tend to look for extra information in the *Viz* representation even if *Code* is more appropriate. At the same time, for *Hard Execution* questions where *Code* is clearly the *mismatching* representation, participants most often rely on the *Viz* from the start.

SWITCH REPRESENTATION we observed several cases where participants chose the mismatching representation in the beginning, then noticed it and switched to another, matching representation, possibly after rereading the question. This behavior can be defined as follows. In the first part of the trial, less than 80% of total duration, the participant only looks at the representation R1 and the question. In the second part, he only looks at another representation, R2, and the question. We compute the relative duration of the first part in percents of

the total trial duration and the number of gazes of at least 0.5 seconds on the representation. We also differentiate between switches from mismatching to matching representation and backwards.

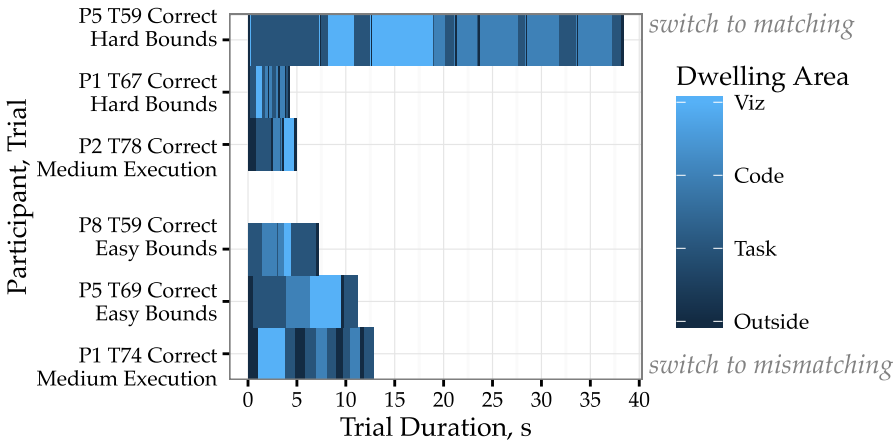


Figure 74 – Gaze timelines for selected trials featuring *Representation Switches* to the matching (top) and mismatching (bottom) representations.

Participants switched the representation in 17% (95%CI = [13%, 21%]) of the trials. In 29% (95%CI = [15%, 42%]) of these trials, they switched to the *matching* representation. Considerably more switches happened for *Bounds* questions, 24% (95%CI = [17%, 31%]) of the respective trials, while only 10% (95%CI = [5%, 15%]) of the *Execution* questions resulted in a representation switch, giving an effect size of 78% (95%CI = [27, 128]). The switch happened after 66.7% (95%CI = [60.3%, 72.9%]) of trial duration on average, with no significant difference between *Bounds* and *Execution* questions. Relating this data with numerous switches to *mismatching* representation, we theorize that participants resorted to *Viz* representation, possibly due to its novelty or visual appeal, after having already found the answer using the *Code*. They may have expected *Viz* to have extra clues for the answer or be better matching for both questions. Figure 74 shows participant's gaze target throughout the selected trials featuring representation switches.

6.5.5 *Discussion*

MATCHING REPRESENTATION TO THE TASK Overall, we observed that participants identified the suggested matching between proposed tasks and representations. When faced with both representations in the second part of the experiment, they are more likely to select the matching representation. Considering the first part of the experiment as an extensive training in representation use, we can argue that the developers learned the question-representation mapping successfully thanks to the practical demonstration of the each representation's benefits. The choice of representation without extensive hands-on

▷ *Users were more open to visual representations once code limits for the task clearly demonstrated.*

assessment of these benefits remains an open question. Participants were generally more open to the visual representation even though its drawbacks for the mismatching tasks were immediately demonstrated.

REPRESENTATION CHOICE When both representations are available, a majority of the participants used both even in the case where one representation would suffice. This behavior can be caused by different reasons: visual appeal and novelty of the graphical representation, familiarity and trustworthiness of the code. Several participants stated that they were more confident in the answer using the code. They also tried to gain in confidence by using the second representation to confirm the answer, whenever it was possible to achieve without considerable effort. In multiple cases, they only had a quick glance on the second representation to self-justify the initial representation choice. On the downside, using both representations led to longer task completion times in several cases. The presence of both representations tricked participants into choosing the mismatching representation, which, in turn, increased the error rate.

▷ *Participants used both representations even when one would suffice, sometimes resulting in longer completion times.*

The current study structure did not allow us to evaluate the effect of removing the code from the program manipulation tool, but we may expect the negative effect on the tool perception will be lower if the potential users have a practical example of code limitations for a given task. This hypothesis can be operationalized and transformed into a separate study.

STRATEGIES FOR MISMATCHING REPRESENTATIONS The users quickly developed strategies to apply given mismatching representations and stuck with them throughout the experiment. Most successful strategies relied on pruning: for *Bounds* questions, checking whether a point obviously outside bounds was present in the visualization; for *Execution* questions, checking whether a sub-condition did not hold for the given point. Participants resorted to this strategy independent of the question being negative. This strategy did not have a strong effect on the results as we asked both positive and negative questions and ensured that participants had to verify a minimal number of conditions for harder trials before getting the definitive answer.

Several participants were very confident about their strategies and were tempted about using them given both representations. For example, P7 used the *Code* to answer a *Hard Execution* question when both representations were available. When asked about this choice, he stated "*[he] did it out of a challenge for [his] strategy*". He than checked the answer with the matching *Viz* representation.

ENGAGEMENT We observed a very low number of abandoned tasks. Participants abandoned tasks with mismatching questions when only one representation was present independently of the type of this representation. This observation may give a clue for improving acceptance of graphics-based approaches for program manipulation: it is necessary to demonstrate on practice the limitations of the code for a particular manipulation task. In most cases, the participants continued the trial even after the experimenter reminded them the possibility to abandon. For *Code*, they may have been guided by necessity to demonstrate their expertise in programming. Several participants found the visualization appealing and engaging, encouraging them to continue.

MULTIPLE COORDINATED VIEWS While our study demonstrates a considerable decrease in completion time and error rate with the representation designed specifically for the particular task, creating an ad-hoc representation for a each possible program manipulation task seems problematic. On one hand, it would require a tremendous effort from the designer. On the other hand, the user will be faced with numerous possibly disparate representations and will have to mentally keep them in sync. We suggest two complementary approaches for the general case. First, analyze the set of program manipulation tasks together, select all the aspects of the program relevant for these tasks and design a set of of coordinated representations for these aspects. Second, provide the user — who is a professional developer — with an ability to map these aspects to the visual and textual cues at will, letting him to appropriate and personalize his working utilities. Both approaches, should they involve multiple views, are to coordinate manipulation to smoother the transition between them.

POTENTIAL FOLLOW-UPS This study is primarily focused on the static visualization where users are not instructed to interact with the representation. For evaluating program manipulation through both representations, one should consider interaction affordances of each of them. A study with a similar design featuring different tasks and success criteria should be conducted to assess the representation choice for program manipulation, after ensuring that both representations technically allow for identical program transformations.

A separate study with eye tracking may be conducted to observe the effect of simultaneously changing multiple representations. On one hand, keeping the changes immediately synchronized may improve understanding of the program transformation effect [213]. On the other hand, multiple moving objects are likely to distract the user from the manipulation and cause more errors. Given *Clint*'s transformation replay functionality, we can differentiate between coordinated

updates triggered by the system or performed in response to the user interaction.

6.6 INTERACTIVE PROGRAM RESTRUCTURING IN THE POLYHEDRAL MODEL

In order to enable *interactive program restructuring* in *Clint*, we mapped direct manipulations of the program visualization to the polyhedral program transformation directives. The visualization provides information that is not easily accessible in the source code and the direct manipulation provides way to interact with the program through the visualization.

TRANSFORMATIONS TO TRANSITIONS The manipulation-directive mapping is bidirectional: graphical action on the visualization are translated into the transformation directives, and transformation directives are translated into the animated transitions. This essentially represents program *transformations* graphically as *transitions*. Transitions can be triggered by user input or by an automatic program optimizer. Provided consistency between user actions and transitions, we can store, replay, adjust and reuse sequences of program transformations in a visual way. On one hand, this allows the developer to better understand the functioning of the polyhedral optimizer, which does not operate in code-level terms by itself. On another hand, it gives the developer fine-grained control over the optimization process enabling him to tweak and hack the program transformation in cases where optimization heuristic fails.

▷ Interactive program restructuring removes the gap between visual analysis and hitherto textual modification.

DIRECT MANIPULATION REPLACES DIRECTIVES Compared to *Clay* and *Chlore* combination that enable developer-compiler partnership by themselves, interactive visual approach in *Clint* replaces transformation directives expressed as text with direct manipulation. Not only this allows to address the issues of directive-based approach, such as transformation target selection, but it removes the need for the user to transition between the visual representation used for program analysis and the directive-annotated code used for program transformation.

CHANGE OF STRATEGY An empirical evaluation demonstrated the utility of the interactive program restructuring tool and its benefits compared to conventional code editing. Our results suggest that the interactive approach reduces both the program analysis and the program transformation time. Furthermore, the presence of interactive visualization is likely to change they strategy developers use when faced with a complex optimization problem. Instead of long and cumbersome program analysis, developers started to explore the space

of available program transformations in order to find the most suitable one. While this trial-and-error strategy may not be preferred for production-level program optimization, it may allow expert developers to alleviate automatic optimization shortcomings as well as may allow the optimizer designers to improve their heuristics. Finally, the combination of an automatically computed optimization and the user input allows to prevent the developers from performing boilerplate transformations manually (even with the help of *Clint*). At the same time, they remain in control of the final program transformation and may avoid cases where an heuristics mishap jeopardizes the program optimization by diverging from the original intention.

REPRESENTATION CHOICE During the preliminary evaluation, we observed that developers were reluctant to abandon the code and use exclusively the visual representation, even after its benefits were demonstrated. We conducted a controlled study of how the developers choose between code and visualization faced with different tasks. After a sufficient practice, they tend to chose the representation that is more appropriate for the task at hand if it is difficult. For easy tasks, they prefer the most familiar representation, the code, even though it could make them less efficient and accurate. When both representations are visible, the developers are susceptible to look at both representations in the cases where one would suffice. This behavior increases the task completion time, but may also increase developers' confidence in the result. An additional study is necessary to assess the perception of switch between multiple and single-representation interfaces for program visualization.

▷ *Users are more likely to rely on novel yet more appropriate representations for hard tasks.*

6.7 CONCLUSION

In this section, we introduced direct manipulation into *Clint* visualization to implement *interactive program restructuring*. It allows developers to directly manipulate with the program visualization in order to restructure it for parallelization and memory access locality. We evaluated the usability of interactive program restructuring with programmers and observed that it effectively changes the strategy they use when faced with complex loop parallelization tasks. Furthermore, we conducted an eye-tracking study with combined textual-visual representation to assess the effects of multiple representation on program manipulation. We observed that, given sufficient training, developers chose a single representation that better matches the task at hand and rarely change it. When they decide to change representations, they take more time to complete the task are are more likely to complete it with errors.

DESIGNING TASK-TAILORED REPRESENTATIONS A program is a complex object described by numerous informational aspects. While the source code is a primarily used representation for the program, we argue that it is not the only possible representation since it does not include all of the pertinent aspects, for example the statement instance-wise dependences. The source code is rather *one of multiple* representations the developer may use to interact with the program, which is not accessible directly, but *through* representations. Software visualizations are another representations that abstract away certain aspects of the program and highlight other aspects. For a visualization to be useful, the aspects it highlights should be different from those already available in other representations (or, at least, more easily available) and appropriate for a specific task in program development or manipulation. Multiple representations may be necessary when none of them taken individually provide all the necessary aspects of the program in an easy to manipulate way.

INACCESSIBLE OBJECT OF INTEREST While one very specific representation may be the best option for a specific program manipulation task, the generalization of this approach would imply creating tailored representations for each program restructuring-related task and making the developer switch representations along with tasks. Aside from considerable effort in designing these representations, it may reduce the productivity of the developers by requiring them to learn all the different representations and constantly maintain the link between all of them. A more balanced approach is to first perform the analysis of the tasks in program manipulation and then design a set of representations, textual or visual, that address all of these tasks. This set of representations should capture all the program aspects required by all tasks in a way that supports interactive manipulation.

PRIMARY AND SECONDARY TASKS The proposed program visualization and manipulation system is tailored to exploiting parallelism. Yet parallelism is not the only potential source of performance increase in the polyhedral model. Improving memory access locality, i. e. scheduling accesses to the same or adjacent addresses to leverage the cache system, allows to drastically increase performance. However, program transformations for locality do not necessarily corresponds to transformation for parallelism. Worse, they are often contradictory. For example, distributing away a reduction statement enables parallelism, but reduces locality. In order to properly integrate access locality feedback in an interface oriented towards parallelization, we need to provide this information discreetly, without distracting the user from the primary task.

7 CONSTRAINED MANIPULATION

7.1 CONSTRAINED PROGRAM MANIPULATION AND BACKGROUND INFORMATION

Program restructuring using the advanced formalism of the polyhedral model allows to verify semantics preservation throughout program manipulation. In *Clay* transformation set, we avoid transformations that do not respect *global validity* conditions by construction. Direct manipulation in *Clint* does not allow the user to perform transformations leading to invalid schedulings. However, some *valid* program transformations may have undesirable effects, in particular a transformation may violate a data dependence and break the semantics of the program. Unless compensated by a further transformation, this manipulation makes the scheduling *illegal*. Another example is a transformation that decreases the locality of the data accesses and thus the potential performance of the program. While these transformations are jeopardous, they should remain allowed as the user may intentionally override the conservative semantics preservation checks. The program transformation tool can notify the user about the danger of such transformations, but not prevent them.

In order to integrate information about potential negative effects of transformation, let us consider the ways to extract it from the model before selecting the appropriate channel to communicate it. *Dependence violation* can be used as a binary property of the transformation or can be quantified further. For example, the number of different dependences violated by the transformation or the total number of dependent statement instance pairs, the execution order of which was interchanged. This information is already present in *Clint* visualization as color-based *feedback*: violated dependences turn red. However, as it is always the case with feedback, the user has to complete the manipulation before receiving the information on its effect. The *feedforward* approach allows the user to evaluate the effect of the manipulation before completing it. Feedforward reduces the interaction time and allows the user to correct their manipulation as they perform it [21, 150].

To introduce feedforward in *Clint*, we need to analyze the effect of transformation before it is requested by the user. We can either pre-analyze all possible transformations from the current schedule, or focus on particular transformations when the user starts the manipulation. Although iterative approaches to polyhedral scheduling space exploration exist [205, 206], they are not adapted to directive-

based transformations and lead to exponentially large search spaces. Therefore, we have to perform the program transformation and analyze its effects during the direct manipulation. Given the simplicity of program transformation in *Clay*, performing one manipulation and analysis step is feasible. In addition to dependence violation feedforward, it allows to show the user the outcome of the transformation during the manipulation.

Effects of transformation on memory *access locality* can also be computed during the manipulation. However, they are hard to quantify [19, 48, 181] and would require a specific visualization technique. Attempts to integrate this information into the existing visualization are likely to result in cluttered, harder to understand interface as *Clint* already uses numerous visual cues for representing program execution and dependences. Given that transformations for locality improvement are often orthogonal, if not opposed, to transformations for program exposure, it is desirable to use different interaction channels to communicate locality and parallelism information. In a general task of increasing performance of a program using a variety of methods, one of the methods can be considered *primary* and others *secondary*. When all methods are related to each other, it is important to inform the user about the effects of such secondary methods while they are using the primary task.

A transformation is considered "dangerous" when it has an undesirable effect, i. e. leads to a potentially problematic schedule of the program. This schedule is represented as a specific state of the visualization characterized by polygon shapes and positions. When the user is directly manipulating visual objects, we can provide him with the feedforward on the danger of certain positions and shapes. We propose a force metaphor: *potentially dangerous positions repulse user's manipulation making him physically overpower the system's warning*. We rely on modifications of the cursor behavior during the direct manipulation to invoke the feeling of resistance. The simulation of haptic perception (texture, weight) using only visual cues is known as *pseudo-haptic* illusion [163, 169]. With conventional pointing devices, such as mice or trackpads, visual modifications to invoke haptic effect boil down to altering the *pointing transfer function* — the mapping between the position of the pointing device in physical space and the position of the cursor on screen.

↩ *Pointing transfer function.*

Both dependence violation and access locality information in the polyhedral model offer different degrees of precision, e. g. binary or multilevel (number of violated dependences), and granularity, e. g. at loop, statement or statement instance level. Prior to integrating this information in *Clint* using pseudo-haptic channel, we need to select the information structure and resolution according to the perceptual limitations of the channel.

While pointing transfer functions have seen a major research interest, most advancements are focused on improving target selection with *semantic pointing* [34] or simulating haptic perception through vision with *pseudo-haptic feedback* [163]. On the contrary, we are interested in mapping the dynamically changing pointing transfer function, i. e. the cursor behavior, to the *abstract information* that is not anyhow related to the cursor motion itself. While potential selection targets can "attract" the cursor in semantic pointing, and virtual object can have "weight" thanks to pseudo-haptics, we are interested in human ability to interpret and quantify the cursor behavior beyond senso-motoric illusion.

We study the morphology of the pointing transfer function and create a design space of its properties that can be mapped to the task-related data. We then select compelling point designs and evaluate them empirically in order to discover whether and how well people can perceive them. Finally, we demonstrate how information related to polyhedral program restructuring can be mapped to cursor behavior in *Clint* interface.

7.2 POINTING TRANSFER FUNCTIONS AND PSEUDO-HAPTIC FEEDBACK

The growing size of displays used for direct manipulation interfaces created a mismatch between the active range of the pointing device, limited by human hand reach, and the range of cursor positions on screen. This issue was resolved by introducing a non-identity mapping between the input device and the cursor displacement: small movements of the device resulted in larger movements of the cursor [171]. This mapping effectively separated the coordinate space of the device, the *control space*, from the coordinate space of the pointing cursor, the *display space*. At first, the mapping was implemented as constant scaling. Hence its name *Control-Display gain*, or C-D gain for short.

↩ *C-D gain*

Unexpectedly large cursor movements led to difficulties in selection through pointing especially for small targets. A group of techniques, commonly referred to as *mouse acceleration*, was implemented to ensure precise pointing and fast cursor movement. Mouse acceleration boils down to making the C-D gain vary as a function of input device speed. It builds on the observation that people move the device slowly when seeking precision and fast when seeking speed. Generalization of the varying C-D gain led to the notion of *pointing transfer function* that maps input device movements in the control space to the pointing cursor movements in the display space [15].

Pointing transfer functions are present in all modern graphical systems and take a variety of non-linear, often piecewise shapes [50]. Most systems do not offer precise control of the transfer function

shape parameters, but instead provide users with simple configuration panels to disable it or adjust its base C-D gain. Casiez *et. al* studied the utility of dynamic pointing transfer functions techniques and identified the usable range of C-D gains [51]. Mandryk and Gutwin evaluated the utility and perceptibility of a "sticky target" effect created by a dynamic piecewise transfer function [178]. They observed that large changes of C-D gain are noticed by the users but are likely to interfere with the pointing task.

Dynamic pointing transfer functions are the cornerstone of two broad groups of interaction techniques. On one hand, *semantic pointing* distorts the relation between movements in control and display spaces to improve target acquisition [34]. On the other hand, *pseudo-haptic* illusion modifies the cursor speed and shape to induce a haptic feeling without force feedback device [163]. Despite different objectives, both pseudo-haptic feedback and semantic pointing rely on subtle changes in pointing transfer function that are not interpreted consciously by the user. Pseudo-haptic feedback was demonstrated to improve small target acquisition in a Fitts' law task, similarly to semantic pointing [185].

Both semantic pointing and pseudo-haptic illusion techniques have been extensively studied in order to make C-D gain variations as smooth and invisible as possible for the user. However, communicating supplementary information through this channel – in, e. g., visually cluttered interfaces – requires to make C-D gain changes explicitly perceivable by the user, which in turn requires a more systematic view and a better understanding of pointing transfer functions for dynamic C-D gain changes and of their effects on interaction.

7.3 MORPHOLOGY OF THE TRANSFER FUNCTION

A pointing transfer function maps the position of the input device in the control space to the position of its visual representation, e.g. a cursor, in the display space. Not only it takes a variety of different shapes, but may also depend on multiple arguments, such as current velocity, acceleration, cursor position, and the context of interaction including modes, hardware and software configuration and settings.

7.3.1 *Transfer function arguments*

In most graphic systems, default pointing transfer functions either maintain a static C-D gain or increase it depending on the *input device velocity* in order to ease and quicken large cursor moves. On the contrary, semantically-loaded techniques are often based on pointing transfer functions with more arguments, including

— input device *displacement* and derived values: *velocity*, *acceleration* and higher-order derivatives;

— time;
— cursor position; and
— interaction context;

A combination of these arguments allows to build advanced pointing and pointer-based interaction techniques.

DISPLACEMENT All pointing transfer functions take input device displacement as an argument, even if it is kept unmodified. For example, a constant C-D gain can be represented as a linear function of input device displacement where the slope corresponds to the gain and the intercept is zero. However, more advanced uses include non-linear dependences, where the C-D gain itself may change depending on the displacement, for example most system pointing transfer functions increase the C-D gain non-linearly for large displacements [50]. Any non-linear function may be used as long as its value is zero for zero input device displacement to ensure no cursor movement happens without moving the input device. Given that the displacement is an N-dimensional vector, where N is the dimensionality of pointing ($N = 2$ for mouse pointing), a function can transform different components of the displacement vector differently. For example, it may change the *direction* of cursor displacement along with its distance by changing the ratio its between horizontal and vertical components. This technique decreases the movement time in steering tasks thanks to subtle readjustments of movement along a narrow path [2, 5]. We prefer pointing device *displacement* to *position* because most widespread pointing devices (mice and touchpads) do not communicate their absolute position, but rather the displacement between two positions.

VELOCITY Average velocity is computed as a *displacement* distance divided by its duration. In the case of constant C-D gain, there is no difference between applying the linear pointing transfer function to the *displacement* vector, or to the *velocity* vector, and then using the update velocity to compute the new position by adding the distance traveled by the cursor during the given time interval. Depending on the intended movement semantics, pointing transfer functions may be specified using displacement or velocity arguments. For example, techniques that add acceleration to the cursor movement, such as force feedback simulation in pseudo-haptic techniques [7, 170], express pointing transfer functions for velocity.

ACCELERATION AND HIGHER-ORDER DERIVATIVES Acceleration, jerk and other high-order derivatives may also be used as arguments of pointing transfer function. However, they require multiple successive displacement measurements to be properly computed and tend to become inaccurate if these measurements are made over a

large time interval. Therefore, they are mostly used to build predictive models for pointing rather than actual pointing transfer functions [153].

TIME The pointing transfer function may as well change depending on time. We distinguish two different cases:

— rare *instantaneous changes*, such as when the user changes settings of the mouse acceleration in their graphic system; and
— steady dependence on time, in which case the C-D gain constantly evolves with time.

Steady dependence on time may be used, for example, to gradually introduce a dynamic pointing transfer function, which combined with other arguments adapts to user's behavior over time [124]. In any case, even when it is not explicitly used in the the pointing transfer function, *time* is used to compute movement velocity, acceleration and other derived values described above.

CURSOR POSITION Techniques that dynamically change the C-D gain to produce effects related to visual on-screen information introduce pointing transfer functions that depends on the on-screen cursor position rather than on the raw input device movement. While the actual cursor position may be computed from its initial position, pointing transfer function and all input device displacements, it also makes the pointing transfer function recursive and increasingly complex to compute over time. The cursor position provides a robust alternative to such intricate definition, and is closer to the underlying model: all visual elements have a certain shape and a position in the same coordinate system as the cursor. For example, object pointing [116], target expansion [66, 115], cursor expansion [54] or snapping [22] techniques heavily rely on cursor position, whether they affect the cursor position, its speed or its shape. On the other hand, pseudo-haptic illusions may either depend on the on-screen position of a specific object [170] or be effective over the whole interaction space to simulate tactile images [7].

INTERACTION CONTEXT In some cases, the mapping between the control and display spaces may depend on the *interaction context*, which is not directly related to a position in any space or of any visible object. For example, a CAD direct manipulation interface may change the velocity of the cursor depending on the *weight* of the virtual object(s) being manipulated [193]. It *carries* contextual information about the interaction that is happening, e.g. the weight in our example, and its effect remains steady wherever the object is placed, contrary to using pointing transfer function defined over *cursor positions*.

7.3.2 *Transfer function shape structure*

While the various shapes of the system pointing transfer functions were extensively studied in the literature [50], interaction techniques relying on C-D gain modification feature a broad range of shapes depending on their intended use and expected effects. For example, pseudo-haptic simulation of physical properties uses transfer functions that reproduce the effects of the corresponding physical laws [169, 212]. We propose two properties for classifying shapes:

— *continuity* that is directly related to the transfer function shape; and
— *output structure* that describes whether the function affects only the absolute value of cursor's velocity, its direction, or both.

CONTINUITY A pointing transfer function should be defined for the entire possible range of its arguments. However, for a particular argument, this definition may be either:

— *Continuous*, featuring a single way to map control space displacement to visual space displacement albeit not necessarily linearly; or
— *Piecewise*, featuring different mappings for different intervals of the input arguments.

A special case of a *piecewise* pointing transfer functions, often found in practice, is the one with different values on different sides of the interval boundary resulting in a "jump". We refer to such functions as *Piecewise with Jumps*.

Piecewise functions may include definition intervals with similar properties. In practice, one may design a function with different constant levels of C-D gain over specific areas and define a *transition* interval that describes how these different levels are connected together without creating *jump artifacts* that may hamper perception of C-D gain change.

OUTPUT STRUCTURE System pointing transfer functions mostly affect the absolute value of the cursor displacement by changing all its directional components proportionally. Such direction-agnostic transfer functions have *Scalar* output. On the contrary, transfer functions that modify the cursor motion direction have *Vector* output. Even though one can express *Scalar* output as a special case of *Vector* output, their application remain different: pointing improvement techniques, e. g. DynaSpot [54], use *Scalar* output while those for steering rely on *Vector* output [2].

Note that the output structure may not be connected to the structure of the input arguments: the displacement may change differently depending on the input device movement direction (vector input), but keep its direction consistent (scalar output). For instance, move-

ment toward an attractive target may be faster than moving away from it, but without any direction changes.

7.4 A DESIGN SPACE FOR C-D GAIN CHANGE-BASED INTERFACES

Pointing transfer function arguments are often combined with visual feedback either to improve or to reinforce the perceptual effect associated with the cursor behavior. For instance, decreasing cursor size while increasing its speed could suggest it is going deeper in a virtual third dimension [170]. In addition, C-D gain changes due to the function's shape and arguments can be designed either to be seamlessly integrated in the interaction or to be explicitly noticeable by the user. In order to better describe these possible associations, we propose a design space that classifies pointing transfer functions according to the following principles.

— Arguments of the pointing transfer function.
— Shape and Structure of the pointing transfer function.
— Associated Visual Channel.
— Interpretation of C-D gain change.

7.4.1 *Design Space Dimensions*

POINTING TRANSFER FUNCTION ARGUMENTS

This dimension aggregates the possible arguments of a pointing transfer function, as discussed before. It is made of multiple binary subdimensions: *input device displacement/velocity*, *cursor position*, *interaction context*, *time*. For better accuracy, specific aspects of the arguments can be specified: direction and velocity of the displacement, particular values of the interaction context such as the state or the number of dragged objects, etc.

POINTER TRANSFER FUNCTION SHAPE AND STRUCTURE

This dimension describes both the specific form of the pointing transfer function, such as linear or exponential, and its structure as described above. The shape defines how the arguments are connected to each other and how they affect the output within its structure. We do not restrict the shape classification to, e. g., linear and non-linear, but rather leave the interpretation open for any definition.

ASSOCIATED VISUAL CHANNEL *None*, *Cursor*, *Background*.

The change in cursor motion is often combined with visual modality, which reinforces or eases its perception, most often with the modifications of *Cursor* shape or size. For example, bump and hole effects were successfully simulated by the simultaneous change of the cursor speed and size [170]. If we consider the C-D gain change as a chan-

nel for communicating information, it becomes redundant with the visual channel as they both convey they same information.

Contrary to the pointing cursor, which is displayed on top of any other graphics, i. e. in the *foreground*, all the other graphical elements can be considered in background. Visual *Background* information may complement different arguments of the pointing transfer function. For example, on-screen objects with non-standard C-D gain may be highlighted in color as in Sticky Widgets [3]. Background may either remain static when C-D gain changes or change as well, such as in elastic image deformation [7]. *Background* and *Cursor* modifications are not mutually exclusive and are often used together, for example, the elastic image deformation technique features both.

INTERPRETATION OF C-D GAIN CHANGE *Illusion*, *Association*. Most of existing interaction techniques featuring dynamic C-D gain use subtle changes that are barely perceivable by the user. Semantic pointing techniques leverage this changes to adjust pointing on a target without necessarily making the user aware of such adjustments, thus creating a sort of *illusion* of precise selection. Pseudo-haptic techniques rely even more on *illusion* rather than conscious interpretation to induce a haptic feeling [212]. Although recent work on elastic images [7] features more explicit connection between the mouse action and visual feedback suggesting a deformation, it still exploits illusion. The key difference between *illusion* and *association* is in users noticing the change consciously and interpreting it with respect to the task at hand, such as associating a particular cursor behavior to a value or a category, in the same way they would do for a color in a bar chart.

7.4.2 *Using the Design Space: Combining Dimensions*

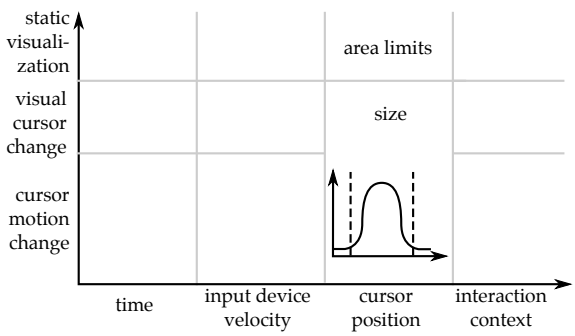


Figure 75 – "Bumps and Holes" experiment [170] in the transfer function arguments design space.

Our design space unifies the representation of interaction techniques based on dynamic pointing transfer functions, including semantic pointing and pseudo-haptic effect. It covers approaches that either

involve dynamic C-D gain modification or rely on visual modification of the cursor, or both. To represent an interaction technique in the design space, we:

1. list the *arguments* of the transfer function and analyze how each of them or their combination impact on the C-D gain, capturing the *shape* properties of the transfer function;
2. specify which attributes of the *Cursor* and *Background* visual representations are affected by the transfer function, e. g. cursor size or background color, and which are statically present, e. g. pointing targets;
3. *associate* the channels with similar dependencies on a specific argument, building the association diagram following the model of Figure 75;
4. put the technique into a high-level view aggregating the changes it makes to the *Cursor* visualization as one dimension and the expected *Interpretation* as another (Figure 76).

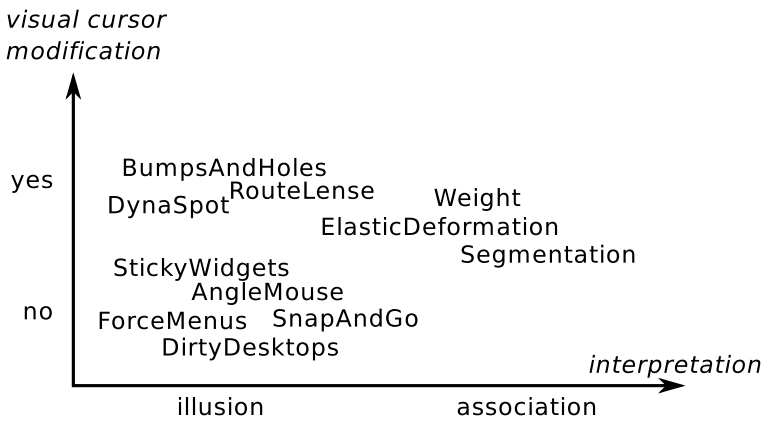


Figure 76 – Most existing techniques use C-D gain modifications to create *illusions* often involving cursor shape or size modifications.

The literature analysis shows that multiple successful pointing facilitation or effect simulation techniques *leverage the redundancy between multiple channels*, propagating the effect of pointing transfer function to cursor shape and background in a multimodal approach [65]. Our design space captures this property with its channel *association* dimension that allows to explore C-D gain perception combined with other visual information, in a more information-centric perspective.

As an example of argument association, Lécuyer *et. al* 's Bumps and Holes experiment [170] features a piecewise pointing transfer function without jumps, that depends polynomially on *cursor position*. The *cursor position* argument is connected to the dynamic *Cursor* size visual feedback and a static *Background* delimits variable C-D gain zones. In Figure 75, we present the shape of this function with solid

line and piece boundaries with dashed lines. We also remove horizontal grid lines to denote that the C-D gain change is reflected by *Visual Cursor Change* (size) and connected to static visual cues (bump area boundaries).

The Dirty desktops [124] technique feature a *time*, *position* and *velocity*-dependent transfer function that is related to visible objects (Figure 77), where the direction of *velocity* and the *position* are combined to define the shape of the function. More specifically, C-D gain changes increase with *time*, and the direction of the input device displacement affects the sign of positional C-D gain change: going toward frequent targets results in higher C-D gains, moving away from them results in lower C-D gains.

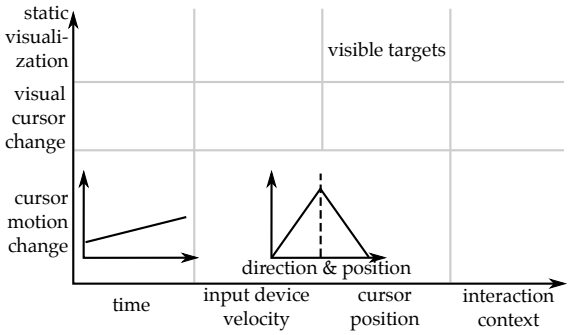


Figure 77 – "Dirty Desktops" technique [124] in the transfer function arguments design space.

Studies of pseudo-haptic techniques suggest that combining *position* and *velocity*-dependent C-D gain changes with *Cursor* size and shape changes is suitable for creating perceptual *illusions*. For instance, dragging elements with weight [193] uses a pointing transfer function that depends on the *interaction context* – the number and total "weight" of the objects being dragged –, that is coupled to the *Cursor* modifications by having the dragged objects follow the cursor (Figure 78).

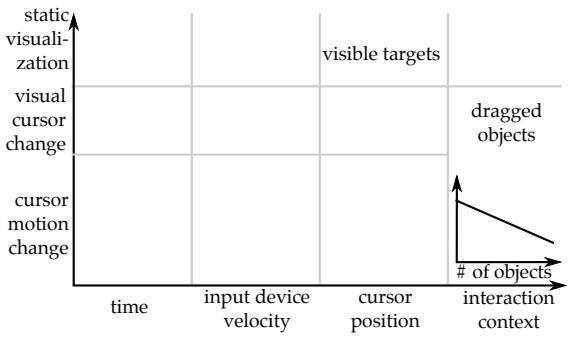


Figure 78 – Dragging Weighted Objects technique [193] in the transfer function arguments design space.

7.4.3 *Design Space Limitations*

Our design space is targeted at visual changes: cursor movement, shape, size or scene background are sharing the same modality. Therefore it does not allow to represent techniques spanning beyond visual modifications, e. g. with specific input devices, for example using inconsistent haptic and visual feedback [164], or using pressure sensing for weight perception [137]. It also assume that transfer function arguments are independent and remain loosely coupled in the function definition, i. e. the contribution of each argument is clearly visible, or the arguments are trivially combined. Trivial combination includes, for example, the case where the cursor *position* defines areas or "pieces", in which the function is *velocity*-dependent; but does not include, e. g. , transfer functions multiplying their arguments. In practice, the majority of transfer functions respect this limitation.

Function shapes are represented separately for each argument to simplify 2D representations.

7.5 COMMUNICATING INFORMATION THROUGH C-D GAIN

High-level view of dynamic pointing transfer function techniques in Figure 76 shows that most techniques focus on creating *illusions* and that only a few rely on *association*, although without completely removing the physicality of interpretation such as weight or elasticity. On the contrary, communicating abstract information such as program execution-related data in software visualization, totally disconnected from physical interpretation of movement, requires to rely only on *association*.

Our objective is to go beyond *illusion*-based transfer functions. Similarly to sticky target techniques, we propose to associate abstract information to on-screen areas with *position*-dependent *piecewise* transfer function. By making *Background* visualization unrelated to the C-D gain changes, we can observe whether people can interpret C-D gain modifications independently from the visual information. For example, a map-based visualization may use color to encode population density and C-D gain to encode unemployment rate of the areas on the map.

DIRECT ASSOCIATION WITH A VALUE Changes in C-D gain may convey information through *direct association*, i. e. the value of C-D gain or cursor speed is proportional to a task-related numerical value. For the map example, higher unemployment density will correspond to lower C-D gain (to attract attention of the user to this particular area when hovering). Direct association is likely to communicate order or other relative information as the user is unlikely to estimate the absolute value of the C-D gain but rather compare the values.

INDIRECT ASSOCIATION WITH A VALUE C-D gain changes may also be used through *indirect association*, i. e. presenting an explicit mapping between different pointing transfer functions and their meaning for the task at hand. For the same map example, three successive slowdowns may delimit crossing the city limits. Indirect association is more suited for *categorical* data, typically represented as explicit mapping. For the C-D gain, the user should compare cursor behavior over the area of interest to the "legend" area. They are likely to learn cursor behavior patterns and recognize them given sufficient training.

However, the information related to program execution in the polyhedral model, *access locality* and *dependence violation* is numerical, rather than categorical. Therefore, we concentrate the effort on studying *direct association* techniques for communicating abstract information.

7.5.1 *Transfer function shape for communicating information*

Due to its dynamic nature – changes in C-D gain are perceivable only when moving the cursor – this channel requires specific structure of information to communicate. First, C-D gain change should be consistent over an area that is big enough for the user to hover it and react to cursor movement. It may be achieved by discretizing the display space. Second, pointing transfer functions over these areas should be distinguishable between each other, preferably allowing to detect multiple different values in the same interface. It may be achieved by discretizing levels of C-D gain.

DISCRETIZING TRANSFER FUNCTION OVER SPACE When the user is moving the cursor, for the C-D gain change to be perceived, it should be consistent during an amount of time longer than user's reaction time to such change. As the change is only observable during movement, the lower bound of the reaction time translates to the minimum control or display space over which the C-D gain should remain stable (but not necessarily constant) given the speed of movement.

While it is hard to establish the minimum speed-dependent space size in the general case, one may use an over-approximated unified size that is sufficient for users to react when moving within typical range of velocities. According to our design space, multiple existing techniques feature *position*-dependent *piecewise* functions and empirically establish transfer function properties for certain piece sizes. For instance, Lécuyer *et. al* used targets of 50 to 150 pixels in their original experiment [170] while Sticky Widgets were tested with 40mm targets on 20" screen with 1024×768 resolution which corresponds to 101 pixel [179]. Although these areas may take arbitrary shapes, they should respect the minimum size in any possible direction of

movement, allowing the user to discover different areas and their boundaries.

DISCRETIZING THE OUTPUT OF TRANSFER FUNCTION The ability to recognize the pointing transfer function shape within certain limits, as it was demonstrated by Lécuyer [170], is more suitable for *indirect association* since it does not allow to compare particular function types for order. On the contrary, *Direct association* requires the user to quantify the C-D gain, resulting in a "just noticeable difference" problem. The absolute value of the C-D gain should thus be constrained to discrete distinguishable levels. As for other physical stimuli, such as weight or friction, these levels are subject to *Weber-Fechner law* stating that the just noticeable difference is proportional to the magnitude of the stimuli [164]. Thus, the C-D gain values should form a geometric progression with base equal to Weber fraction of the effect. For example, Weber fraction in stiffness perception induced through pseudo-haptic feedback was empirically estimated at 0.14, while it was observed as 0.23 on average in physical stiffness perception studies [7].

Discretized levels lead to abrupt changes in C-D gain on the edge of two areas, which may be noticed on its own. On the other hand, sudden changes may hinder the usability when, for example, pointing is required along with hovering. Smooth *transition* between levels, shorter than minimum size of the area, may alleviate these effects. Discretization also allows to include multiple distinct levels of the C-D gain to the same function, providing a clear association between discrete task-related value and C-D gains.

7.5.2 *Evaluating information communication*

Rather than directing our study towards specific usecase of polyhedral program optimization in *Clint*, we propose an abstract task on C-D gain perception with possibly conflicting visualization. By varying the discrete levels, the transition functions and the area sizes, we want to observe whether and how people can perceive C-D gain changes in visual interfaces. We observe their use strategies in order to suggest general implications for building interfaces with supplementary information communicated through cursor behavior.

In order to simplify the tool design and rule out confounding factors, we restrict the structure of the information one can communicate so that it can be mapped to a grid of equally-sized on-screen areas using discreet ordered levels of C-D gain. First, we explore the perception of C-D gain changes without any visual feedback. We analyze the effect of transfer function shape and the number of discernible levels. Then, we study the relation between C-D gain changes and possibly contradicting visual *Background*. Finally, we evaluate how

well people can separate cursor motion patterns from other visual elements.

7.6 USING C-D GAIN CHANGES ALONE

With this first experiment, we investigate how many C-D gain areas a casual user is able to discriminate with different piecewise transfer function shapes and which strategies he applies to perform this task. We study the perception of multiple distinct C-D gain levels in a single interface rather than explore the just noticeable difference since this approach is ecologically closer to the intended use.

7.6.1 *Experimental Protocol*

HYPOTHESES According to our design space, we use a piecewise position-dependent pointing transfer function. We expect that the discretization step, the function shape within the discretized areas and the transition between them will influence the number of discovered zones, resulting in the following hypotheses:

— Hypotheses on transfer function shape (**H1**)
  - **H1a** Transitions that reach lower C-D gain levels than the following level improve the perception of difference.
  - **H1b** Lasting changes in C-D gain are likely to be perceived better than short changes, even if the amount of C-D gain change is big.
  - **H1c** Sharp C-D gain transitions are likely to distract users and to be misinterpreted.

— Hypotheses on transfer function discretization (**H2**)
  - **H2a** Users are more likely to better discriminate discretized levels of C-D gain than a continuous change.
  - **H2b** C-D gain changes are more likely to be perceived if they remain stable over a larger area.

PARTICIPANTS Fourteen unpaid volunteers recruited through a public mailing list in authors' organizations, 8 male and 6 female aged 27.6 on average (SD = 4.1), participated in our study. Two of the participants were left-handed but used the mouse with their right hand as they are used to do everyday. All participants had normal or corrected to normal vision. Each participant performed the full experiment.

TASK Participants were instructed to hover a horizontal rectangular strip on the screen with the mouse cursor. The strip was separated into 8 *zones* with C-D gain decreasing along the horizontal dimension following Weber-Fecnher law conforming to the resent JND result with pseudo-haptic feedback [7] (Figure 79a). The participants only

see the contour of the whole area but not the separation between the zones (Figure 79b). They were asked to press any key on the keyboard each time they have detected a new C-D gain zone — i.e. they thought the speed of the cursor was slower than before —, which we refer to as a *hit*. A visual indication confirming each hit appeared for 500 ms, but disappeared if another hit was made during this time so that only one indication was visible at a time. Participants controlled the mouse with the right hand and pressed the key with the left hand. It allowed them to keep moving the mouse cursor when marking a zone while preventing for unintended moves that could occur when clicking with a mouse button. A trial started when the participant clicked with the left mouse button on the start area at the left of the strip, and ended when they clicked anywhere outside of the strip. Participants were allowed to hover the area multiple times during the same trial, as they would be able to do in normal use of a variable C-D gain interface.

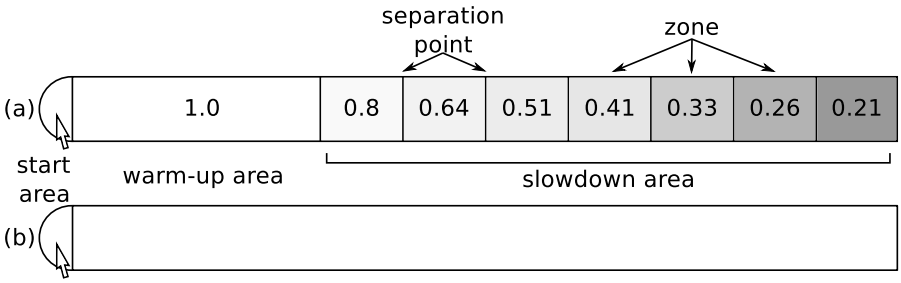


Figure 79 – Participants hovered a horizontal area divided into zones (a) with decreasing C-D gain while seeing only its contour (b). Presented C-D gain values are normalized to the default system value.

The first zone of the rectangular strip, referred to as *warm-up area*, was larger than others zones and had a constant C-D gain equal to the default value of 1 [1]. This zone allowed the participants to establish a baseline for the cursor behavior at a constant C-D gain. The *warm-up area* was lying within the visible contour of the strip, without any visual difference from the other zones. Participants were instructed that the C-D gain would eventually decrease when they moved to the right but were not given any detail about the size of the warm-up area. All the other zones were featuring decreasing C-D gain and are thus referred to as *slowdown area*.

PROCEDURE The experiment is a full factorial [5 × 3] within-subject design with SHAPE of the C-D gain function and base SIZE of the zone as factors. Six blocks with randomly permuted combinations of all factors were performed by each participant after a training of 5

1. C-D gain values are relative to the default system gain. A value of 1 means unaltered C-D gain, lower values correspond to slowing down the cursor, higher values — to speeding it up.

trials. One session took 45 minutes on average. Participants filled in a demographics questionnaire upon completion.

SHAPE OF THE C-D GAIN FUNCTION Each zone has a *base C-D gain* which follows a geometric progression with a scale of 0.8 so as to follow Weber-Fechner law: If g_n is the base C-D gain of $zone_n$, the base C-D gain of $zone_{n+1}$ is $g_{n+1} = 0.8 \times g_n$ (Figure 79). Then, the SHAPE defines how the C-D gain varies between the zones of the rectangular strip. As shown in Figure 80, we considered five types of SHAPE drawn from our design space, according to the results of a pilot study suggesting that people detect either continuous C-D gain levels or changes between them:

— *Jumps*, which assigns base C-D gain for each zone without transition (Fig. 80a);
— *Smooth* uses a growing side of the Gaussian function described in [170] to smooth the boundary between zones in ±10% of width around zone *separation points* (Figure 80b).
— *Bumps*, which decreases following the same Gaussian function that peaks at 15% lower C-D gain than the next level. This shape serves to analyze the effect of a local "bump" similar to those used by Lécuyer *et. al* in [170].
— *BumpsOnly*, that features the same C-D gain transitions as *Bumps* but maintains the base C-D gain equal to 1 in each zone (Figure 80c). This is a control condition for *Bumps* in order to determine if the participants will interpret bumps as a change even though the C-D gain in all zones is identical.
— *Continuous*, that varies the C-D gain following a continuous exponential function with power 0.8, normalized so as to fit in C-D gain range $[0.2, 1.0]$ on the given length for each size (Figure 80c).

Comparing *Continuous* with *Jumps*, *Smooth* and *Bumps* will allow to investigate our hypothesis **H2a** about the quality of the perception of discretized C-D gain levels against continuous change. Comparing *Jumps* against *Bumps*, and against *Smooth* will allow to study hypotheses **H1a** and **H1c**, respectively, about the effect of the transition between areas with stable C-D gain. *BumpsOnly* compared to *Bumps* will assess the effect of the transition alone, according to **H1b**.

SIZE OF THE ZONES The base SIZE of the zones defines its length in pixels. We considered three base SIZE conditions: *Small* (40px), *Medium* (70px) and *Large* (100px). In order to prevent participants from learning the relative size of the zone and relying on distance difference rather than on cursor speed, we multiplied the size of the warm-up zone by a random number within the range $[3, 6]$ and randomized the length of each other zone in the same trial within ±10% of its base length. The position of the slowdown area on the screen

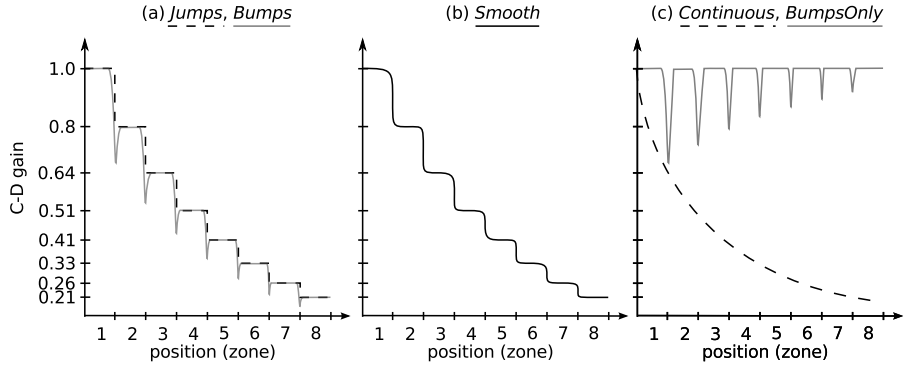


Figure 80 – *Shapes* of the C-D gain functions used in our experiment.

was randomized as well. Variation of the zone size will allow to investigate the hypothesis **H2b** by comparing short and long changes of C-D gain.

In total, we collected data for 5 SHAPE $\times$ 3 SIZE $\times$ 6 replications $\times$ 14 participants $=$ 1260 trials.

7.6.2 *Data Collection*

For each *hit*, we recorded the cursor location (x and y) and the *timestamp*. We also recorded an event-based cinematic log of the cursor position with and without the correction imposed by the C-D gain change. In order to measure participants behavior in the motor space, we videotaped their hands and recorded mouse position and rotation 30 times a second. Since the camera was controlled by our software, we were able to synchronize the cinematic logs in both the display and motor space. We used linear interpolation to compute instantaneous linear and angular velocities at the same moment in both spaces by taking numeric derivatives of position and angle respectively. Finally, an experimenter observed the participants and took notes summarizing their strategies for detecting C-D gain changes.

7.6.3 *Apparatus and Implementation*

The experiment was conducted on a MacBook Pro 15′ Retina with a screen resolution of 1920 $\times$ 1200 (57.7 ppcm) with a standard Apple Mouse. In order to track participant movements in the motor space, we attached a smoothly-shaped arc made of hard foam to the top of the mouse with two bright green and orange tokens on top of it as shown in Figure 81. The centroids of these tokens were tracked by a web-cam mounted on a curved support at 400 mm height over the table. The resolution of the camera was of 800 $\times$ 600 pixels and it covered an area of 500 $\times$ 315 mm, allowing to track tokens' positions with 1 mm precision and to compute the rotation angle between them with 1° precision.

Figure 81 – Mouse tracking and experimental setup.

We implemented a Qt widget restricting the variable C-D gain to one particular application. On top of this widget, the system cursor is replaced with a circular black and white cursor in order to prevent any directional bias in the movement. When computing the cursor displacement, we took into account all C-D gain values of the points hovered during this movement so as to ensure small changes, e. g. *Bumps*, are still perceivable.

We disabled the system transfer function (mouse acceleration) during the experiment, since it is non-linear and may interfere with our C-D gain changes [50]. In the post-experimental questionnaire we also asked participants about their knowledge and preference on system transfer function.

7.6.4 *Metrics and data post-processing*

In order to quantify participants' performance in perceiving C-D gain changes, we define the following metrics on the hits that allow to analyze the number of simultaneously distinguishable zones and reveal some information about participants' strategies to find them. We also establish metrics to describe participants behavior elements such as repeatedly hovering a small area or changing the speed in the motor space.

We report most data graphically with violin plot-like charts, like Figure 83a, which present a vertical histogram of the measure distribution for each value of a factor. Dark part of each histogram represents the 95% confidence interval[2] of the mean and the darkest line the mean itself.

2. Confidence intervals are computed using single-sided t-test for means, and using Tukey HSD procedure for differences of means.

Due to growing concerns in various research fields over the limits of null hypothesis significance testing for reporting and interpreting experimental results [73, 82], we base all our analyses and discussions on estimation, i.e., effect sizes with confidence intervals [74]. Furthermore, we argue that estimation results and accompanying graphs are better suitable for communicating results in a work mixing methodologies from different disciplines. We describe the computation and interpretation of effect sizes and confidence intervals in Appendix B.

HITS CLUSTERING AND FILTERING Since the task allowed multiple passes over the area with variable C-D gain and thus allowed multiple hits in the same zone, we had to post-process the initial list of *raw hits*. For each trial, we aggregated all the *raw hits* lying in the range of ±5% C-D gain of one of the predefined levels as a single *zoned hit* for the corresponding *zone* (excluding the baseline value of 1.0). The number of these *zoned hits* correspond to the number of different C-D gains found in all passes during one trial. All the other hits are considered *inter-zone hits* and are analyzed separately from zoned hits. If multiple *raw hits* were made in the same zone throughout the trial, all the hits after the first are treated as *repetition hits*, which may suggest better detection of a particular C-D gain. Raw hits around the bump peak in the *Bumps* shape condition do not count into any zone and are treated as *inter-zone hits*.

NUMBER OF PASSES The *number of passes* in a trial is computed as the ratio of the distance traveled by the cursor in the slowdown direction (left to right) by the size of the whole area. This ratio may thus be fractional and less than 1 if the participant did not travel across the whole area. We chose it over the number of full passes from the start to the end of the area since participants were not required to start over at the left of the area each time they decided to re-pass for strategy exploration purpose.

AVERAGE SPEED The *average speed* in motor space was computed as a mean of mouse speeds registered over a specific period of time: the entire trial, one particular pass over the slowdown area or a fixed ±150ms interval around the time of a *raw hit*. The speed was calculated from the logged mouse positions, i.e. in the motor space.

7.6.5 *Ordering effects*

We first analyzed the number of *zoned hits* per block and observed an ordering effect as shown in Figure 82. The substantial change of 24% between the first block and others may be due to participants being less used to the interface and cursor parameters, and thus paying more attention to subtle changes in cursor behavior. We also observed

that more time was spent per trial in the first block, suggesting that spending more time or moving slower results in higher hit rate. Consequently, data from the block 0 is discarded from future analyses unless explicitly mentioned.

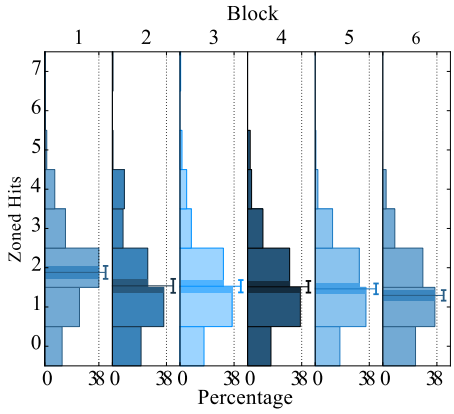


Figure 82 – Mean of *zoned hits*. Each column depicts a histogram of distribution. Darker parts and error bars are 95% CIs. Number of zoned hits decreases by 24% between blocks 0 and 1.

7.6.6 *Effects of transfer function shape*

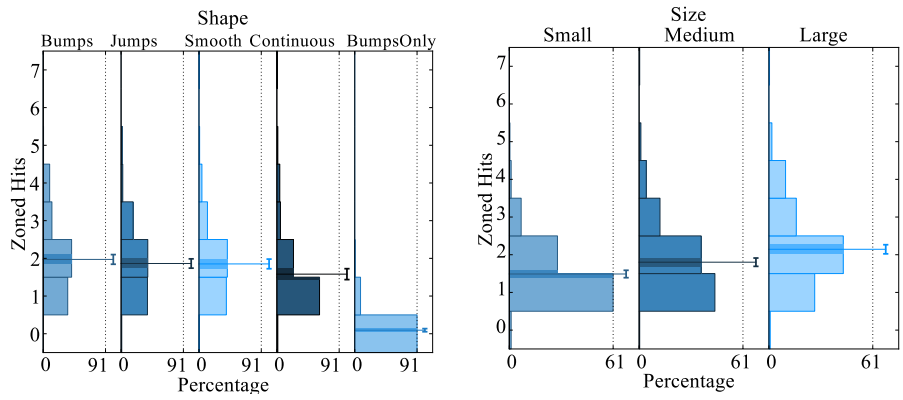
▷ Two more C-D zones beyond baseline were consistently detected.

NUMBER OF ZONED HITS Two zoned hits were made on average, implying that participants detected three different C-D gain levels including the baseline. In 90% of trials with *BumpsOnly* shape, no raw hits, and consequently *zoned hits*, were made (Figure 83a). *Bumps* shape (mean=1.97) outperformed *Jump* (mean=1.86) and *Smooth* (mean = 1.85) shapes by only 6% on average thanks to a slightly larger number of trials with 3 or 4 *zoned hits*.

ANALYSIS OF AVERAGE SPEED We observed comparable distributions of average speed with means of 0.55 ± 0.06 px/ms (95% CI) for all shapes except *BumpsOnly*, which had a mean velocity of 0.42 ± 0.06 px/ms (95% CI). As for SIZE, *Small* resulted in slightly lower speeds with mean of 0.48 ± 0.05 px/ms (95% CI) whereas *Medium* and *Large* have a mean of 0.54 ± 0.05 and 0.55 ± 0.05 px/ms (95% CI), respectively, similar to the behavior in Fitts' pointing task where participants would reach higher velocities for larger movement amplitude.

▷ People detect lasting changes in C-D gain and often miss temporary changes.

HYPOTHESES H1A AND H1B AND H1C While the users perceive larger transitions slightly better (**H1a**) as suggested by the comparison of *Bumps* to *Smooth*, transitions alone are less noticeable and lasting steady behavior of the pointing transfer functions is required for users to consciously differentiate two areas (**H1b**) as indicates the comparison between *BumpsOnly* and other shapes. Contrary to



(a) *BumpsOnly* led to hits in only 10% of cases. *Continuous* led to less hits than discretized shapes.

(b) Each increase of zone size resulted in 18% more zoned hits.

Figure 83 – Mean of *zoned hits*. Each column depicts a histogram of distribution. Darker parts and error bars are 95% CIs.

our hypothesis **H1c**, users neither felt disrupted by abrupt changes caused by *Jumps* shape nor did they misinterpret them as system lags.

7.6.7 *Effects of transfer function discretization*

NUMBER OF ZONED HITS SIZE had the largest impact on the number of *zoned hits* passing from mean 1.49 for *Small* to 1.79 for *Medium*, to 2.15 for *Large* with constant increase by 18% (Figure 83b). Detailed analysis of distributions shows that only the *Large* size resulted in the majority of trials having 2 or more *zoned hits*. Comparing SHAPES, *Continuous* shape to a lower number of zoned hits than other shapes for all SIZES, with 1.58 on average.

DISTRIBUTION OF HITS BY ZONE In order to explain smaller number of *zoned hits* for *BumpsOnly* and *Continuous* shapes, we computed the percentage of trials when a zoned hit was made at each zone for each SHAPE. Since these shapes allow hits outside ±5% zones, we also computed this ratio for hits between zones that gave us a better understanding of participants' behavior when faced with piecewise transfer functions without jumps.

Zones with lower C-D gains are most frequently detected, going up to 58% of trials for discretized shapes (Figure 84). *Jump* shape resulted in higher detection rates for C-D gains of 0.41 (32%) and 0.26 (43%). *Smooth* shape peaked at C-D gain of 0.33 reaching 38% of trials. Although raw hits in *Continuous* shape do not fall properly into zones, hit rate also peaked around C-D gain 0.32. It resulted in roughly equal number of hits within the zone and after it. Interestingly, the percentage of hits between 0.26 and 0.21 zones with this shape is

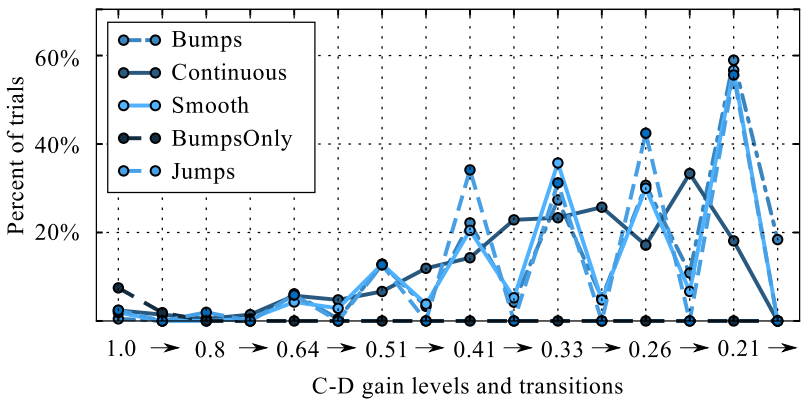


Figure 84 – Percetentage of trials with hits within and between zones. *Jump* shape resulting exclusively in zoned hits by construction only dominates for 0.41 and 0.26 zones. *Continuous* shape result in close numbers of hits within and between the zones.

the largest, while percentages of hits within the zone itself decrease despite the fact that interval range constantly decreases.

HYPOTHESES H2A, H2B The comparison between *Continuous* and other shapes evidences in favor of C-D gain discretization showing that continuous modification of C-D gain is harder to perceive and quantify (**H2a**). The size of the constant C-D gain zone had the most influence on the perception (**H2b**).

▷ *Discretization helps people to separate C-D gain zones.*

7.6.8 *Exploratory analysis: elements of strategy*

BRUSHING Participants had recourse to a *brushing* strategy – i. e. multiple repetitive hovers of a small area – for at least one separation point in 40-50% of the trials, depending on the shape and size (Figure 85). According to debriefing interviews, this might have allowed participants to verify the presence of a separation point when they lacked confidence. Brushing is also present in *Continuous* shape even though no real separation points are present. This suggests that brushing is a general strategy used by participants for detecting the C-D gain change. With *BumpsOnly* shape, brushing was only used in 15–35% of the cases, which could be due to the absence of lasting C-D gain changes being quickly noticed by the participants. The slightly higher percentage of brushing for *Smooth* and *Bumps* for *Small* size may suggest that separation points for these shapes are actually harder to detect with confidence, contrary to our initial hypothesis **H1c**.

PASSES Since instructions clearly allowed participants to hover the strip multiple times, we computed the percentage of *zoned hits* made before completing N full passes (Figure 86). For *Small* size, 3 passes

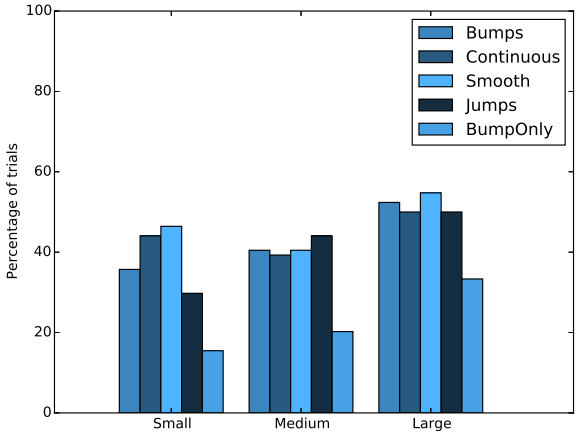


Figure 85 – Brushing was used in half of all trials except *BumpsOnly* shape.

are required to perform 90% of all zoned hits and at least 5 passes to perform 95%. For *Medium* and *Large* size, 2 passes are enough to reach 90% of hits and 3 passes for 95%.

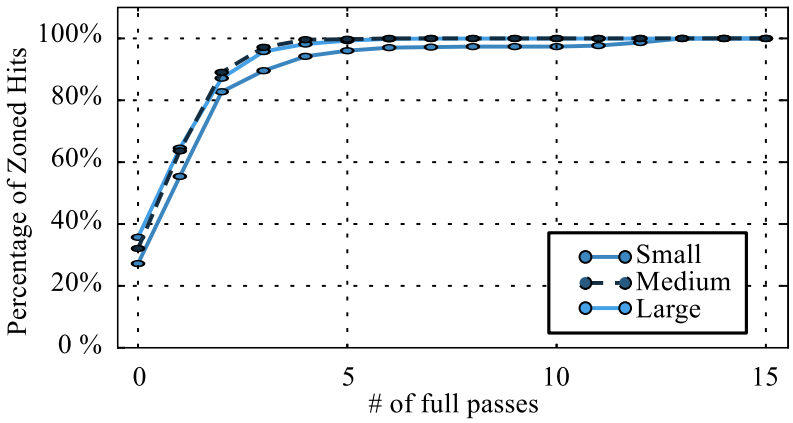


Figure 86 – At least 90% of the all zoned hits were made during first 3 passes over the slowdown area.

ACCELERATION In order to compute the acceleration of the mouse movement in the control space, we took the instantaneous speeds computed from the cinematic log. Using mouse and cursor position data simultaneously, we selected the parts of the trial where participants were moving in the direction with decreasing C-D gain and built linear regression of recorded velocities over time. We interpreted slope of the linear fit as acceleration that gave us less noisy results than averaging instantaneous values of acceleration.

We observed that hovering the slowdown area for the first time in the trial, participants tend to accelerate the mouse in control space to compensate the cursor slowdown in display space caused by the system. This acceleration is not present in *BumpsOnly* shape that does

not feature durable slowdown. However, in other passes, including the last one, there is a tendency for decelerating the mouse movement in addition to the cursor slowdown (Figure 87).

▷ *Users build the mental model of cursor behavior: at first they expect constant speed, but by the end they expect deceleration.*

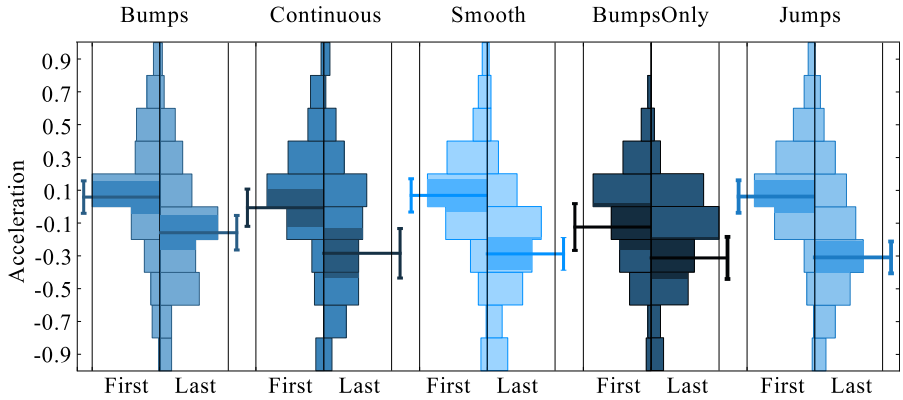


Figure 87 – Acceleration is present only in first passes over the slowdown area. Error bars are 95% CIs.

7.6.9 *Qualitative elements of strategy*

AVOID CLUTCHING All participants planned the movement over slowdown area in order to avoid or minimize clutching. However, they often underestimated the slowdown effect and were forced to clutch in the middle of the movement. Clutching considerably disrupted the perception of the C-D gain change as participants were forced to hover again the position of the cursor at the time of clutch.

LOOKING FOR CHANGE POINT At least 6 participants experienced difficulties deciding where to click in a slowdown area with *Continuous* shape. They performed brushing-like moves with larger amplitude in order to compare cursor behavior in two points. One of them qualified this shape as "very smooth" and explained that he was "looking for a point where it changes", but was not able to identify it due to continuity of the shape. The majority of participants stated after the experiment that they were also looking for a specific point where the C-D gain changes. Eight participants moved cursor back after crossing the separation point if they observed a change in the behavior. Even without performing the brushing, they were looking for a point where this change took place and tried to hit exactly over it despite not being instructed to hit a specific point.

SPEED AND ACCELERATION After practice, participants learned to expect the slowdown, which made them misinterpret the absence of lasting slowdown in *BumpsOnly* as acceleration. Five participants asked if it was present in the experiment despite having read the instructions that stated the opposite.

Ten participants were hovering the strip with different mouse speeds in the control space. A typical scenario involved passing over it with high speed for the first time and then decreasing it for the subsequent passes. Brushing was often constituted of movements with different speed to gain confidence about the separation point.

Six participants avoided crossing the vertical border of either strip or the window by slowing down the mouse when the cursor approached it. This may have led to them misattributing the slowdown in the end of area to their hand movement rather than to the intended system change. One of the participants explained that she "*was not sure if the cursor or [her] hand was slowing down*" thus resigning to hit due to the lack of confidence.

7.6.10 *C-D gain change interpretation*

In the post-experiment debriefing, we asked participants whether they associated the behavior of the cursor with any physical model. Eight participants reported that they did not create any physical model, but rather interpreted C-D gain as a function, exponential in most cases, or a pictured plot of such functions. Other participants invoked cursor stickiness or weight. Interestingly, one of the participants turned the mouse over during the practice and checked whether it was indeed sticky. Even if such abstract mental models evidence in favor of conscious interpretation of the C-D gain change, they may be due to the participants' large exposure to mathematics according to their background.

▷ *Some participants did not have a physical association with the cursor slowdown, but interpreted it abstractly.*

Participants evaluated their willingness to use dynamic C-D gain change in everyday interfaces (web, document processing) as well as in specialized interfaces (critical control systems, medical applications) on a Likert scale. Six participants disagree with the idea of using it in everyday interfaces arguing about time loss and frustration, while only three agree with it. For everyday use, they suggested to use lower C-D gain when performing search along with visual highlight, where C-D gain is associated with the relevance of the search result. On the other hand, all participants agree or strongly agree with the presence of dynamic C-D gain changes in special interfaces, proposing namely to convey limitations and dangerous control values on sliders by making them harder to reach, to notify the user about an irreversible action before they do it, or to communicate the quality of visual element position. These suggestions match exactly the intended use of dynamic C-D gain in *Clint*, yet participants were not specifically instructed about interactive program visualizations.

7.6.11 *Discussion*

Participants detected two slower C-D gain zones different form the base one on average in most conditions, which may be caused by too subtle difference between levels, their elevated number or the lower limit of C-D gain levels. These results corroborate our suggestion that in order to be perceivable alone, changes in C-D gain should be larger than if they are combined with a visualization helping to establish a consistent mental model.

Effects similar to those observed in pseudo-haptics studies appeared in the first trials or the first passes over the slowdown area. This may be due to the mismatch of users' expectations and the actual behavior of the cursor: users expected it to have constant speed when they moved the mouse with constant speed, thus when the cursor slowed down they accelerated the mouse to compensate. Therefore we suggest to use dynamic C-D gain changes sparsely so that users do not accommodate the change making it less noticeable.

We choose to use only decreasing C-D gain since it was less consistent with mouse acceleration, familiar to some participants. However, we did not find difference in behavior between participants who reported using mouse acceleration with those who reported having disabled it. We also did not vary the number of different C-D gain levels in order to better compare continuous and piecewise shapes. Nevertheless, base size variation allowed us to explore the precision limits for C-D gain change-based interface supporting the space discretization proposition.

Finally, we observed that when they are faced with an interface featuring such dynamic changes, users first explore the overall high-level behavior and then analyze particular places of interest, resulting in a "focus+context" strategy. Furthermore, they rarely analyze areas that were not noticed during the first exploratory passes. This suggest that using minimal visual feedback delimiting the places of interest in advance may be beneficial for perception of the change. We thus conducted a second experiment to assess the effect of a background visualization on C-D gain perception.

7.7 C-D GAIN CHANGE WITH BACKGROUND VISUALIZATION

In this second experiment, we studied the perception of C-D gain change with consistent and contradictory background visualization so as to evalute the use of C-D gain to communicate complementary information unrelated to the visual channel. We also decreased the number of different C-D gain levels and studied its effect on change perception.

7.7.1 *Experimental Protocol*

HYPOTHESES

H1 Passive visual feedback mirroring the C-D gain changes (i.e., the zones) yields better detection of the zones in a shorter time.

H2 Passive visual feedback contradicting the C-D gain changes may increase misinterpretation of zones, with erroneous C-D gain separation points detected around visual separation points.

PARTICIPANTS Six participants of the previous experiment and six other unpaid volunteers, 8 male and 4 female, aged 25 on average (SD=3.8) participated in our study. All participants were right-handed and had normal or corrected to normal vision. They reported to use a mouse daily.

TASK AND PROCEDURE The task and procedure are the same as in the previous study, except that the rectangular strip was made of only 4 C-D gain *zones* – with respective relative C-D gain values of 1.0, 0.64, 0.41, and 0.26.

The experiment is a full factorial $[2 \times 3 \times 4]$ within-subject design with SHAPE of the C-D gain function, base SIZE of the zone and number of VISIBLE ZONES as factors. We use the same SIZES of C-D gain zones and the same size randomization method as in the previous experiment, and kept only the *Jumps* and *Bumps* SHAPES conditions (we discarded *Smooth* SHAPE since it yielded results comparative to *Jumps*, and *Continuous* and *BumpsOnly* since they were control conditions).

The number of VISUAL ZONES is 1, 3, 4 or 5. One VISUAL ZONE implies that there is no visible visual separation, only the contour of the strip being visible as in the previous experiment. Otherwise, the interface comprises zone contours and represents C-D gain levels by shades of gray (Figure 88). When four VISUAL ZONES are visible, we ensure that visual zones correspond exactly to C-D gain zones by using the same randomized values for both, assessing consistent background for hypothesis **H1**. For the two other cases, visual channel represents different information, contradicting the C-D gain change. We then randomize zones sizes independently for visual and C-D gain zones in order to prevent participants from establishing a relation between those and assess **H2** hypothesis.

Each participant performed 4 repetitions of the full experiment with randomly permuted condition after a practice block of 8 trials featuring all combinations of VISUAL ZONES and SHAPE for a fixed SIZE. They were instructed that, when the visual feedback was present, it *might or might* not correspond to the cursor behavior, and were asked to concentrate on the latter.

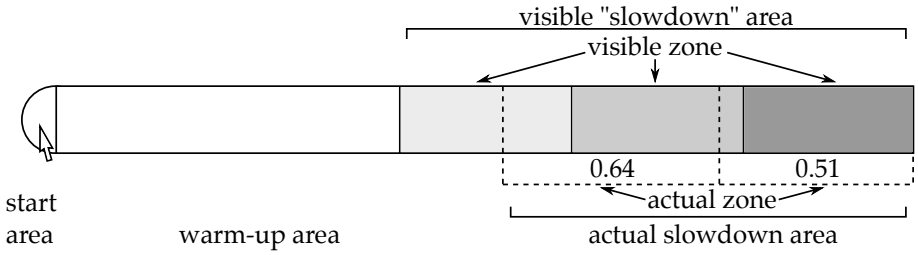


Figure 88 – Participants hovered a horizontal strip divided into zones with decreasing C-D gain with potentially contradictory visual feedback as gray-shaded rectangles.

The experimental setup was identical to the previous study, the only modification in software was the presence of zones visual feedback depending on the VISUAL ZONES number.

We discuss the same measures as in the first experiment, and present them in the same way (graphical representation and estimation). We did not observe an ordering effect between blocks for both *raw* and *zoned* hits, however the duration of the trial decreases by 28% to 37% after each block. This may suggest that the participants were becoming more confident with practice of the task. The number of both *raw* and *zoned* hits was comparable for the users who participated in the previous study and those who did not.

7.7.2 *Effects of visual feedback*

NUMBER OF ZONED HITS On average, around two *zoned hits* were made (Figure 89,left). We observed 14% more *zoned hits* for the BUMPS shape, 31% of which were made close to bump peak. An effect of SIZE, similar to the effect observed before, was found in this study with 22% increase in *zoned hits* between *Small* and *Medium* size and only 9% change between *Medium* and *Large* for all shapes and number of visible zones combined (Figure 89,right).

Bumps shapes outperformed *Jumps* for all SIZES by at least 9% for *Medium* and up to 19% for *Large*. It also resulted in consistently larger *zoned hit* rate for all sizes with increase within the same 9-19% range. The number of *zoned hits* is $9 \pm 0.4\%$ (95% CI) larger with 4 VISUAL ZONES comparing to 1 VISUAL ZONE with *Jumps* shape, while it changes less $3 \pm 0.4\%$ (95% CI) in the same comparison for *Bumps*.

COMPARISON OF RAW AND ZONED HITS The number of *raw hits* practically did not vary between 1,3 and 4 VISIBLE ZONES with differences of -2% and 7% respectively. More importantly, participants did only 5% more *raw hits* with 3 VISIBLE ZONES comparing to 1 VISIBLE ZONE, although in the former case visible and C-D gain zones matched exactly. This result suggests that, knowing in advance about possible mismatch between changes in C-D gain and visualizations,

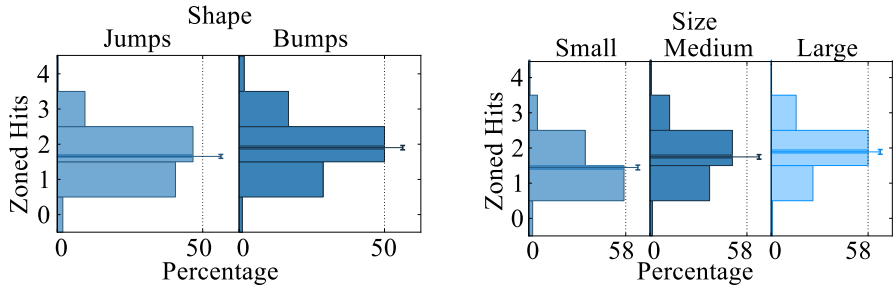


Figure 89 – Number of *zoned hits* was 14% higher for BUMPS. It increased by 22% when changing from SMALL to MEDIUM size, but only by 7% when moving to LARGE size. Darker parts and error bars are 95% CIs.

▷ *Users were able to dissociate visual background from cursor speed changes.*

users do not associate C-D gain to the visual counterpart. However, 10% more *raw hits* were made with 5 VISIBLE ZONES. More detailed analysis of *raw hit* distribution shows that this increase was caused by a larger portion of 2 and 3 *raw hits* in this case (Figure 90,left). While the distribution remains similar, larger number of VISIBLE ZONES is likely encouraging users to look for more than one separation point in the trial.

The number of *zoned hits* varied less, in range of ±3% between 1 VISIBLE ZONE and 3 or 4 VISIBLE ZONES, while for 5 VISIBLE ZONES 9% more *zoned hits* were made (Figure 90,right). Most of the *raw hits* increase for this case is thus reflected in *zoned hits*.

In 15% of trials with 5 VISIBLE ZONES, *raw hits* were made before and after a visual separation point with no actual change in the C-D gain. With *Jumps* shape, 17% of such trials were affected, and 13% for *Bumps*. *Medium* size resulted in 17% of such mishits as well. On the other hand, in 23% of trials with 3 VISIBLE ZONES *raw hits* were made within the same visual zone, with a consistent distribution between SHAPES and ±5% difference between sizes.

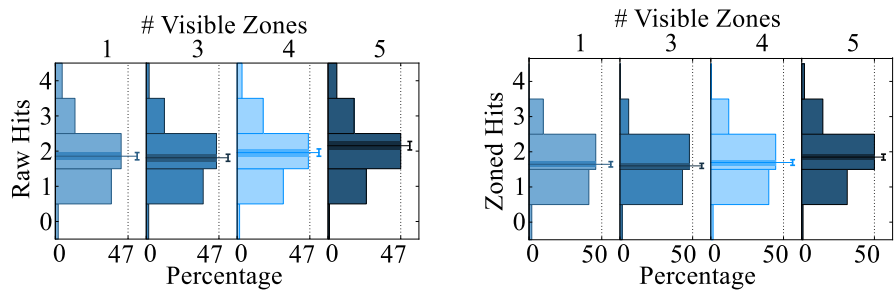


Figure 90 – The number of visible zones impacted more the number of *raw hits*, rather than *zoned hits*

HYPOTHESES H1 AND H2 We did not find evidence that supports **H1** hypothesis: participants found close number of zones in comparable time. This may be due to the instruction requiring to fo-

cus on changes of C-D gain without giving any supplementary hints about feedback consistency. Participants seemed to ignore the visual representation completely. However, we argue that this instruction allowed make tasks consistent between experiments preventing participants from establishing correspondence between visual and C-D gain change instead of observing the latter. We also found evidence against hypothesis **H2** suggesting that, having a precise instruction, participants are able to dissociate visual representation from C-D gain changes and interpret the latter independently. In the same time, larger number of visible zone may have urged participants to look for more changes in the same trial resulting in higher *raw hit* rate.

▷ *Precise instructions about the mismatch helps users dissociate different information channels.*

7.7.3 *Comparison with previous experiment*

NUMBER OF C-D GAIN ZONES Even with smaller number of simultaneously present C-D gain levels and higher difference between them, the same average number of *zoned hits* was made. Further investigation is required as to establishing the just noticeable difference of the C-D gain change and introducing more important slowdowns, with relative C-D gain lower than 0.2.

EFFECT OF SHAPE Comparing to the previous experiment results, we found similar effects of SIZE and no interaction effect between it and SHAPE. Combined, these results support our general observation that pointing transfer function should remain steady during a sufficiently long time interval in order to be explicitly noticed. For *Large* size the average number of hits even decreased comparing to the previous experiment due to the absence of large (more than three) hit numbers. However, better recognition rates were observed with larger transition changes.

BRUSHING we analyzed brushing moves around both C-D gain and visual separation points. Figure 91 shows the ratio of passes over the separation with brushing to total passes; left part of each histogram corresponds to C-D gain separation points, and right correspond to visual separation point. We observed that for 3 and 5 VISIBLE ZONES, participants performed brushing over C-D gain separation points far more often that over visual separation points. For 4 VISIBLE ZONES, these numbers are identical since C-D gain and visual separation points had same coordinates.

7.7.4 *Discussion*

The results of this experiment suggest that the number of recognized C-D gain changes is mostly impacted by the size of the area in which the C-D gain remains constant and less by the number of

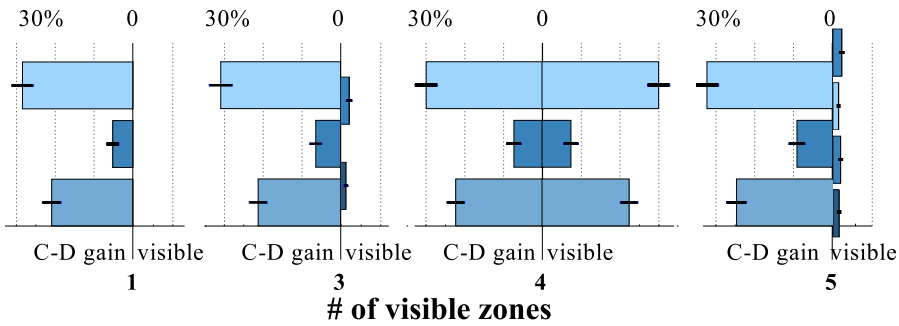


Figure 91 – Participants perform brushing moves around C-D gain separation points, but not around visual separation points when they were different

actually present distinct levels. At least 100 px area may be required for more or less stable recognition of 3 levels including the baseline.

Despite relatively low hit ratios, users are able to interpret C-D gain values independently of the visualization. This suggests that these changes may be used for conveying abstract information to the user, even in visually cluttered interfaces. In this study, we did not focus on cursor acceleration that may be give more recognizable levels.

7.8 VARIABLE C-D GAIN FOR COMMUNICATING INFORMATION IN INTERACTIVE PROGRAM RESTRUCTURING

7.8.1 *Predicting Transformation*

Having evaluated the use of variable C-D gain for information communication, we can integrate this functionality in interactive program restructuring in *Clint* to represent either dependence violation or access locality information. This information is only relevant during the transformation, i. e. when the user performs direct manipulation, and thus is appropriate for communicating with C-D gain, only visible during manipulation. In order to evaluate the effects of the transformation on dependences and access locality, *Clint* has to perform the program transformation in real time while the user is still manipulating the interface. The mapping of manipulations to directives has different starting points for different transformations. For example, SHIFT transformation is triggered by dragging the statement, RESHAPE transformation is triggered by dragging the corner handles of the statement, etc. This allows to determine which transformation is to be performed when the user just started the manipulation. Parameters of the transformations are mapped to the spatial movement of the visible objects. During normal operation, movements length is divided by step scale and rounded to obtain the transformation parameter. For example, moving the polygon left by 23 px left when the distance between points is 16 px corresponds to SHIFT by 1, while

moving it left by 25 px corresponds to SHIFT by 2. It suffices to use ceiling approximation instead of mathematical rounding to find the next possible transformation even from a small movement.

Once the directive and the parameters of the transformation are inferred from the user manipulation, *Clint* performs the transformation in the background and computes the dependence violation and access locality metrics. Using these metrics, it alters the C-D gain around the guessed end position of the manipulation. If the user still has not finished the manipulation, *Clint* continues to compute metrics for the same directive and subsequent integer values of the parameters and to update the C-D gain around the respective positions.

Contrary to other manipulations, moving polygons between coordinate systems may result in numerous β-modifying transformations. While a RESHAPE manipulation may affect only the dependences and locality in the loop nest surrounding the target statement, β-modifying transformations affect the entire polyhedral part of the program. However, they operate on statement level rather than on individual statement instances allowing for simpler effect analyses. Thanks to the mandatory modifier key for polygon/coordinate system manipulation, such transformations are easily distinguishable even before the actual direct manipulation started. Therefore, *Clint* may use different metrics for evaluating results of β-modifying transformations.

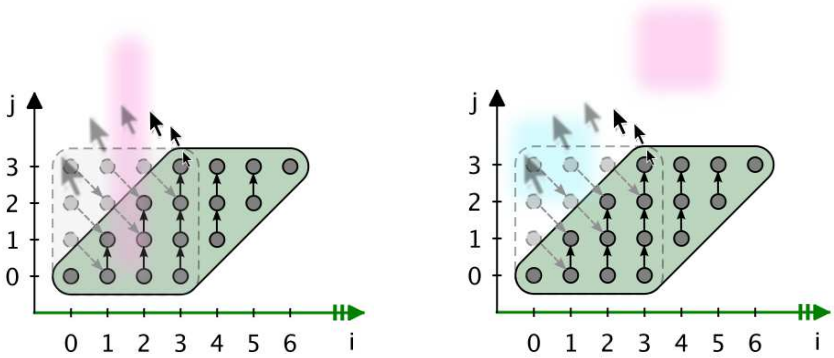
7.8.2 *Representing Dependence Violation*

TRANSFORMATIONS OPERATING WITHIN LOOPS Building on the initial idea of interface "resistance" to the potentially dangerous transformation, we place a slowdown "barrier" with smaller C-D gain around the polygon position that would violate dependences as shown in Figure 92a. This barrier has three different levels of slowdown depending on the number of violated dependences:

1. one dependence is violated by the transformation;
2. more than one dependence is violated, but not all dependences, in which the target statement is involved;
3. all (more than one) dependences, in which the target statement is involved, are violated.

If no dependence is violated, C-D gain is left unmodified. Once the user reaches the program state where the dependence is violated, the C-D gain barrier is removed to allow the user undo the action with no interference. However, other barriers may be created, or the value of existing ones recomputed, for the potential subsequent transformations.

TRANSFORMATIONS MODIFYING STATEMENT NESTING IN LOOPS For β-modifying transformations, the barriers are introduced around



(a) Slowdown barrier for dependence-violating transformation.

(b) Sticky (sloppy) endpoints for locality increasing (decreasing).

Figure 92 – C-D gain modifications in *Clint* interface: light blue overlays show cursor speedup areas, darker purple overlays show cursor slowdown areas; dashed light gray rectangle shows the polygon form after the transformation predicted from small mouse motion is applied; blurred and grayed out mouse cursors show the expected cursor path to complete the transformation.

the coordinate system in which the dependence would cause a violation as depicted in Figure 93a. To simplify real-time computation in this case, we only consider between-loop dependences as follows:

— take the dependence graph for the initial state using β-vectors as node identifiers (already computed for visualization);
— select the dependences that involve the β-vector of the target statement at a depth corresponding to the current projection;
— modify the β-tree (maintained throughout manipulation) according to the expected transformations without performing the transformation itself;
— consider as violated the dependences for which the lexicographical order of dependence source and target β-vectors was inverted.

Before performing the actual transformation on the relational polyhedral representation, *Clint* alters the tree representation of the program part using simple and fast operations to enable fast creation of the slowdown areas.

7.8.3 *Representing Access Locality*

The polyhedral model enables precise instance-wise analysis of the memory accesses, however the computation of spatial and temporal access distances require both the execution-related information, such as physical addresses of the arrays in memory and aliasing information, and hardware-related information, such as cache sizes and memory access timings. Given that the program transformation in *Clint*

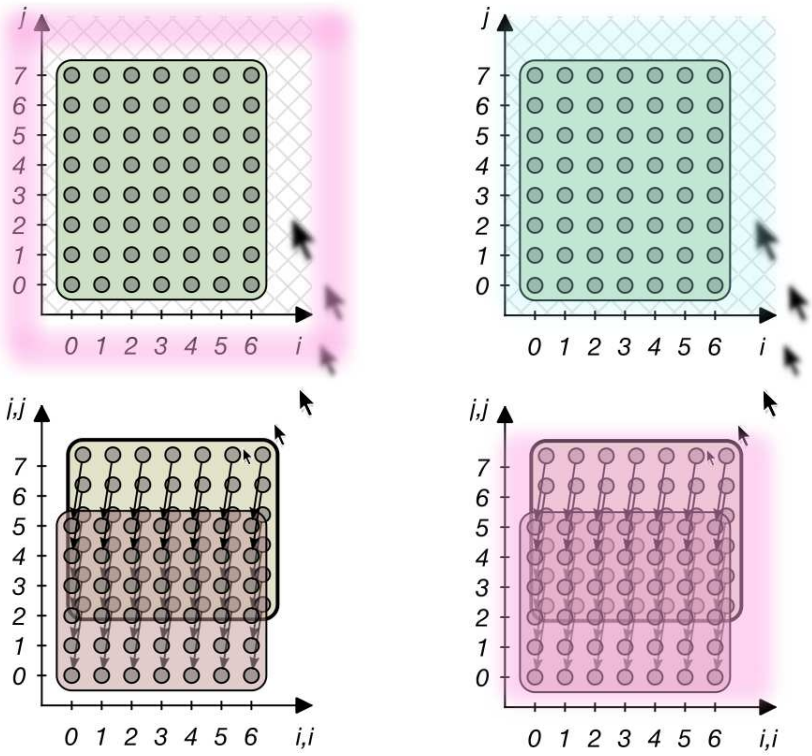
is done statically, without executing the program, and not necessarily on the target architecture, the precision of the access locality information remains limited. Furthermore, performing precise analysis requires heavy computation yet it has to be done in real time in order to provide feedback during the user interation. Therefore, we resorted to a simple heuristic based on data dependences to evaluate access locality. Like any heuristic it is subject to imprecision, but this imprecision matches well the low resolution of the cursor motion feedback we can use.

In the general case, we rely on the notion of *dependence distance vectors* (DDV) for temporal locality. This vector contains along each dimension the number of loop iterations at the corresponding depth that separate consecutive accesses to the same memory address. For example, if the memory address is accessed at the iteration (i, j) and then again at the iteration $(i+1, j-1)$, the dependence distance vector is $(1, -1)$. The elements of this vector may be parametric. The dependence distance may be expressed as a single number rather than a vector by computing the number of iterations of all outer loops, but the final expression may involve parameter multiplication and thus be inadequate for the polyhedral representation. However, dependence distance vectors may be ordered with respect to each other given that parameter values are also ordered.

LOCALITY WITHIN LOOPS Prior to visualizing statements as polygons in *Clint*, we compute the dependence distance vectors for all statements. When the user starts the direct manipulation by dragging the statement, *Clint* automatically predicts and applies the transformation and recomputes the distance vectors only for the dependences involving the current statement. For each dependence, it compares the new distance vector and the old one. If the new distance vector is ordered before the old one, the transformation is considered to *increase* locality. On the contrary, if it is ordered after the old one, it is considered to *decrease* locality. Finally, identical vectors do not influence locality. The definitive metric is the difference between the number of dependences where locality is increased and the number of dependences where locality is decreased by the transformation. If this value is positive, the transformation increases the locality overall, if it is negative, it decreases the locality.

Changing DDV by 1 in the last dimension is unlikely to have strong effect, but we keep the reasoning uniform to simplify computation and avoid platform-specific values.

The cursor motion feedback for locality is based on "stickiness": polygon positions that increase locality are made sticky and those that decrease it are made slippery. Areas are computed relative to the mouse click position, see Figure 92b for an example. We differentiate three two levels in each direction: increase/decrease for 1 or many dependences.



(a) Slowdown barrier for dependence-violating transformation.

(b) Entire coordinate systems are made sticky or sloppy to show locality.

Figure 93 – C-D gain modifications in *Clint* for statement-level transformations: light blue overlays show cursor speedup areas, darker purple overlays show cursor slowdown areas; the target coordinate system features crossing diagonal lines in the background.

LOCALITY BETWEEN LOOPS For the β-modifying transformations FUSENEXT and DISTRIBUTE, that are likely to have stronger effect on the access locality, we consider dependences on the statement/loop level. Although the polyhedral model allows to obtain more precision, this choice is more consistent with the visualization approach we use in *Clint*, namely inter-loop dependences are aggregated rather than presented between each pair of points in the respective iteration domains. The overall locality is considered to increase if a hitherto inter-loop dependences becomes intra-loop and decrease in the case of inverse modification. This heuristic allows to favor the fusion of the loops accessing the same memory locations. The entire coordinate system is considered as the target location and is subject to C-D gain modifications as shown in Figure 93b.

We call inter-loop the dependences between statements in different loops and intra-loop the dependence between statements in the same loop nest.

7.8.4 *Selecting Feedback Source*

Evidently, both information sources cannot be connected simultaneously to the same communication channel, which would be confusing for the user. The feedback should either be associated with the dependence violation *or* with the access locality improvements. While we provide the user with the settings to switch between the two sources or to turn off cursor motion feedback completely, the default option is to represent *access locality*. Indeed, this information is orthogonal to the parallelization task which was the primary design goal of the visualization. Furthermore, the feedback on the dependence violation can still be provided visually without cluttering the visualization: it suffices to highlight the violated dependences in red before the transformation is complete at the same time as visualizing the expected position of the polygon after the transformation.

7.9 GUIDING MANIPULATION AROUND CONSTRAINTS

COMMUNICATING NON-CRITICAL INFORMATION Using the small changes to the pointing transfer functions, we were able to communicate supplementary information about the program. This information is related to the manipulation and therefore is provided only during the manipulation and avoids cluttering the static representation. Although memory access locality is important for program optimization in general, it is not the primary interest of *Clint* visualization targeted at parallelization. While one may argue for providing as much information as possible in a single representation, i. e. combining both the dependence and the locality information visually, we find that it makes the visualization cluttered and barely legible [89]. This kind of supplementary, non-critical information may benefit the user if perceived correctly, but its absence will not impede the interaction.

"HARD" AND "SOFT" LIMITATIONS Thanks to a subtle physical metaphor, we propose to communicate the *soft limitations* of the interface. Contrary to hard limitations which cannot be violated, soft limitations are essentially warnings about potential issues if the manipulation is completed. In case of visual manipulation of polyhedrally represented programs, soft limitations are, for example, dependence violations that will compromise the semantics of the program, but keep it within the model. Hard limitations, on the other hand, will move the program outside of the model. For instance, a general case of loop linearization results in non-polyhedral loop bounds. Soft limitations are frequent in programming and may range from bad practices and antipatterns to deprecated functionality, to undefined behaviors. While programmers tend to avoid these cases, sometimes

they are exploited on purpose. Therefore, these cases should result in subtle warnings for the user rather than being flat out banned.

USING CURSOR MOTION OR SHAPE We chose to use cursor motion, an underused visual element present in all target systems, to communicate supplementary non-critical information. In order to use it for communicating task-related information to the user, we proposed a design space that unifies interaction techniques involving cursor movement from the information-centric perspective. It makes explicit the relation between pointing transfer function morphology and visual changes observable by the user allowing both to analyze the information component of existing techniques and to use the pointing transfer function for communicating abstract information.

Contrary to the cursor shape, extensively used for communicating system state, e. g. busy, and interaction affordances, e. g. draggable or editable, the cursor motion is not already mapped to the pre-existing concepts in the user's mental model. It remains open for interpretation depending on the particular task at hand. Contrary to other visual cues, cursor is already present on screen and does not increase the number of visual objects. It is always shown on top of the visualization. It follows the object the user manipulates making the interaction contextual and object-related. And finally, it is disconnected from the object highlighting that the information it is communicating is related to the *interaction* rather than to the object itself.

IMPRECISE INTERPRETATION Our experimental evaluation demonstrated that cursor motion can be perceived independently from the visual background. Despite the relatively low numbers of perceived changes, it demonstrates the principle of communicating abstract information that is not related to the pointing or cursor movement. Its low resolution makes this channel unsuitable for task-critical information, but it remains reasonable for the secondary, non-crucial information. However, prior to mapping task-related data to the pointing transfer function levels, this data has to be quantified and structured so as to respect the perceptive limitations. More generally, not only the interaction technique should be adapted for the task, but the task information structure should fit the interaction technique.

POSSIBLE FOLLOW-UPS Our studies concentrated on using scalar modifications to the C-D gain since they were sufficient for our task. More advanced interfaces will require exploring other parts of the cursor motion design space, namely including movement direction changes using vector modifications. Direction changes will allow, for example, to guide the user towards the proposed solution in a training interface while still letting them to feel in control, contrary to

transformation animated replay currently implemented in *Clint*. Furthermore, they will let the users to divert from the automatically suggested decision by deviating from the guided path. Separate study should focus on the acceptable limits of the cursor motion modification that does not distract the user from interacting, but is still perceivable. Overall, we find C-D gain modifications a promising modality for communicating supplementary information or guiding users through the interface.

INTERACTING WITH PRIMARY AND SECONDARY DATA Introducing access locality information into the polyhedral parallelization-oriented interface illustrates a larger problem of reconciling multiple, often orthogonal, aspects of the program in a single interactive representation, crucial for interactive program restructuring. Our approach consists in separating program restructuring in different tasks and classifying aspects of program description by order of importance for the particular task. Non-critical aspects can still be provided if they don't obstructed the primary information and can be structured so as to suit the information communication channel still available in the interface. It requires quantifying and adapting both the interaction channel and the underlying model. Furthermore, the user may want to be aware of the effects a program restructuring has on the other aspects that are currently out of focus making the transition between different representations an important part of the restructuring tool design.

Program manipulation is a peculiar case for the instrumental interaction paradigm [23] the program is the object of interest that does not have a canonical form and is not directly accessible to the instruments. It is rather hidden under the stack of representations (or abstractions), e. g. a static control program part is represented as a set of relations in the matrix form, later converted to the set of points in relation, which is in turn represented as points. From the user's point of view, the instruments manipulate the top layer of the representation stack, the visualization, but the expected result lies at the deeper layers. However, the higher-level representation that "converts" the instrument action to other actions on the lower-level representation may be considered an instrument itself.

8 CONCLUSION AND PERSPECTIVES

8.1 DESIGNING TOOLS FOR INTERACTIVE PROGRAM RESTRUCTURING

Using the advanced program representation and analysis tools provided by the polyhedral model, we demonstrated a case of designing the system for interactive program restructuring. Building such system for a particular programming-related task, such as performance optimization through parallelism and locality, requires developing both the program model and the user interaction techniques. In this chapter, we conclude by positioning *interactive program restructuring* with respect to Direct Manipulation [227], Instrumental Interaction model [23] and juxtaposition between visual and textual program representations [70].

UNIFYING TEXTUAL AND VISUAL The idea of interactive program restructuring highlights the *interactivity* and unifies textual and visual representations as long as they offer the user sufficient understanding and manipulative power. This unification is also present in Conversy's theoretical model of program representation perception [70]. In fact, the program representation should be chosen following the two criteria:

— the representation provides *new* information about the program or allows to access the information and reason about it *easier*;
— the representation fully depicts the information relevant for the task and allows for *manipulation*.

Should the textual form be better understandable or easier to manipulate for a specific program restructuring task, e. g. realigning code, it is to be preferred to other representations as shown in chapter 6.

While most existing software visualization techniques support program analysis thanks to tailored representations, they rarely feature enough interactivity to manipulate the program through the visualization. They focus on *visualizing* the software rather than on providing representation to *manipulate* it. On the contrary, in interactive program restructuring, we propose to design the visual representation with the future interaction in mind. Nevertheless, a multitude of novel program representations may be challenging to use in practice, especially if the user has to learn the new technique for each specific task rather than focus on the high-level problem solving. These representations should build on existing practices and leverage domain experts' knowledge and intuition as we demonstrate with graphics-

based transformation vocabulary directly mapped to visual program transformation. A practical demonstration of the benefits and limitations of every representation to the user is of crucial importance for improving acceptance of novel program restructuring tools. In addition to that, new representations are to relate to the existing well-known representations, e. g. code, through a series of supportive techniques such as animated transitions or overlay connectors.

UNCOVERING STRUCTURE In an interactive representation, aspects of the program of interest relevant to the task are mapped to the (primary) notation of the representation, for example graphical properties such as color or size or textual content such as keywords or spacing. This mapping is bidirectional and is subject to numerous limitations. On one hand, changing the represented aspects of the program should remain unambiguous and keep the program well-formed throughout the manipulation. On the other hand, the notation of the representation is subject to human limitations, perceptual (color recognition, just noticeable difference in sizes, etc) and manipulative (pointing precision, typing speed).

Designing a task-oriented interactive program restructuring tool consists in:

— identifying aspects of the program that are necessary for solving the task or relevant to it;
— defining the structure of the program-related information to represent (type of values, ranges, precision, etc.) and model the program in a way that yields the required structure;
— designing an interactive technique that supports both the representation and the manipulation of the required structure.

These three steps are connected by the *information* that represents the program and its *structure*. Individual tool design cases may start with any of the steps as long as the structure of information matches. For example, in the polyhedral program restructuring, we started with the pre-existing program model and created a visualization technique from it, however we extended the model to support program transformation given the visualization.

Due to the lack of canonical representation of a program (or its control and data flow), programmers are often facing multiple tools that represent the same aspects of the program differently, e. g. the list of available classes or the tree showing the inheritance structure, or different representation of different aspects simultaneously, e. g. the code and the list of functions. Interactive representations should ensure changes are propagated back to the program in question and, further, to all connected representations unambiguously. In our system, graphical changes can be immediately reflected in the code and vice versa.

PROGRAM PERFORMANCE AND PROGRAMMER PERFORMANCE The effect of interactive program restructuring techniques is difficult to evaluate. It breaks down into two separate results: effect of program restructuring and effect of interactivity. While the program restructuring may be evaluated according to its goal, e. g. performance or program readability, given appropriate metrics, the evaluation of the interactive techniques involves user testing and is subject to individual differences in expertise and varies substantially with relative task complexity. In cases like polyhedral transformation sequence recovery, restructuring becomes *feasible* thanks to the interactive approach, and its effect depends on the particular program and programmer. An interactive restructuring technique should be considered successful if it increases either the *program performance* by optimizing it or the *programmer performance* by providing more suitable and faster-to-use representations. Decreasing the program analysis and restructuring time allows developers to reduce the number of mistakes and concentrate on more complex optimization problems. Lowering the perceived complexity of the program restructuring lets the developers use it more extensively and thus results in more performant software. In our studies, we observed an effective change of strategy between barely manageable complex code structures and interactive visual representations where the developers started to experiment with transformations more easily and obtained an optimized version instead of being stuck in analysis phase.

COMPILER-ASSISTED PROGRAM TRANSFORMATION Most conventional implementations of the programming languages feature a user interaction loop: the language user develops a program using external tools, such as text editors or IDEs, submits it to the language implementation, compiler or interpreter, and receives its feedback before continuing to develop the program in an external tool. Modern IDEs try to shorten this loop by hiding compiler invocations and providing smoother feedback integration on one hand, and by modeling the program separately during the development. These internal models are often less precise than those of a compiler, but easier to manipulate.

On the other side of the chain, even the most elaborate compiler analyses often lack high-level information about the intended program behavior. This information is sometimes supplied using language extensions, such as compiler pragmas, or compiler flags. However, requesting the developer to communicate this information is hard due to the constantly growing sizes of code bases and the out-of-reach complexity of compiler analyses. Even if the developer can correctly interpret and augment these analyses, he does not necessarily reason on the same level of abstraction while creating or restructuring a program.

We propose a *semi-automatic* approach to program transformation. First, an optimizing compiler automatically creates a program transformation. Then, this transformation is represented in a better understandable way for the developer than internal compiler representations. The developer is able to amend the transformation using the same higher-level description before requesting the compiler to execute the final transformation. This approach gives the developer control on the loop optimization process: the transformation selection is performed on demand and each individual transformation can be disabled if it diverts from the original intention. Furthermore, it does not require to understand the internal functioning of the optimizing compiler and allows to swap optimization algorithms as long as the higher-level representation remains the same.

This *reification* of the program transformation also allows to manipulate the program transformations separately, reuse and share them. With time, it may allow to create transformation recipes or establish best practices [101] that will eventually improve the automatic program transformation.

8.2 CONTRIBUTIONS

This thesis is an interdisciplinary work between human-centered computing and program optimization. In addition to applying human-computer interaction methodology to design and evaluate an interactive system to support optimizing compilation in the polyhedral model,we made the following technical and theoretical contributions to the field of optimizing compilation and empirical contributions to the field of interactive system design.

POLYHEDRAL MODEL First, we defined the structure of the polyhedral scheduling relations by proposing *global validity conditions* that ensure the same amount of computation performed by the program before and after a polyhedral transformation. This structure is less restrictive than the previously existing constraints of scheduling invertibility and unimodularity. Second, we proposed a complete mapping between classical loop-level transformation directives and changes to the polyhedral scheduling relations. Contrary to previous mappings, this mapping is *complete* and allows to transform any globally valid scheduling relation to any other globally valid scheduling relation. Jointly with Bagnères [12], we proposed an algorithm that recovers a sequence of transformation directives transforming an arbitrary polyhedral scheduling relation into another arbitrary scheduling relation given only the initial and the transformed state. Moreover, we designed an interactive program transformation system based on the polyhedral transformation set and sequence recovery algorithm pro-

viding unprecedented control over the polyhedral transformation dissociated from a particular optimization algorithm.

DESIGN OF INTERACTIVE PROGRAM TRANSFORMATION SYSTEMS Our experimental evaluation of the interactive program transformation systems suggests that a visual representation oriented towards loop-level data dependences and program parallelization effectively changes the developer's strategy in program optimization. The results support the idea of mapping modifiable program properties to the modifiable elements of visual representation to create efficient program manipulation systems.

Following up an observation, we conducted an experiment to better understand how developers use text and visual representations simultaneously in a program transformation task. It demonstrated that the preference of developers for textual representations may decrease their performance compared to other representations better matching the task unless benefits of alternative representations are practically demonstrated to the developers as a part of training.

INTERACTIVITY IN SOFTWARE VISUALIZATION Having proposed a design space for software visualization systems and techniques, we identified the lack of interactivity at the program manipulation level due to irreversible mapping between manipulable graphical properties and program abstractions. Using the state-of-the-art polyhedral abstraction for loop-level parallelism in the programs, we exemplified the user-centered development of an interactive program restructuring tool. It required both understanding how certain graphical properties can be used to communicate abstract information, as demonstrated with cursor motion morphology and experiments, and augmenting the model to provide the information in the required form.

Combining multiple representations of a program, which itself contains multiple layers of abstraction, in a single interactive system is a compelling use case for interaction design. It not only requires the program models to support interaction, but may affect the way the interaction itself is modeled. Before proposing future research directions and perspectives, we discuss the implications of interactive program restructuring for interaction models, in particular the Instrumental Interaction model [23].

8.3 INTERACTING WITH INACCESSIBLE OBJECT OF INTEREST

Domain Object is a term for specifying the current object of interest in Instrumental Interaction paradigm.

DOMAIN OBJECTS AND REPRESENTATIONS The Instrumental Interaction paradigm [24] allows to analyze the program restructuring in the polyhedral model even independently of the visual representation. A program is a domain object and program-modifying transformations are instruments that affect it. The paradigm accounts for the attention shift between different domain objects, making the instrument an object of interest itself if necessary for applying meta-instruments. Thus a tool for creating and analyzing program transformations becomes an instrument, and the transformation itself becomes a domain object. However, the transformations we defined operate on the relational form of the polyhedral model and not on the program directly. While we can narrow down the object of interest to a particular aspect or part of the program, such as the control flow in a given loop, it still needs a certain proxy object: a model or a *representation*. The instrumental interaction model stipulates that the representation is the domain object in this case. For example, in case of polyhedral program transformations as instruments, relational representation of the loops is the domain objects. However, our system is specifically designed so as to avoid the user reasoning in terms of polyhedral relations, but rather in terms of code-like entities and control flow. Therefore, we should decompose the object of interest into several representation layers in order to make instruments applicable. Having these layers will also allow us to specify the mapping and control change propagation as discussed above.

A representation is more powerful than a simple subset of the domain object's properties relevant for interaction. In particular,

— it can be obtained by a non-trivial, parameterizable procedure, e. g. include aggregated data;
— it can affect how the interaction instrument is applied, specify or limit its behavior.

An image histogram in a graphical editor is a good example of representation. It aggregates color and brightness data from the entire image and can be parameterized by, e. g. color channel. As a representation, it may serve for interacting with the image itself by, e. .g., color-based selection [60]. Identical histogram manipulations will change the image differently depending on what channel it represents, i. e. the representation *carries* information required for the interaction. Finally, applications may limit changes to the histogram, e. g. photography processor typically impose a certain range for brightness, while the instrument can potentially go beyond these limitations.

INTERACTING WITH AND THROUGH REPRESENTATIONS The original instrumental interaction model defines an instrument as a "*mediator between the user and domain objects*" [23]. Using this definition, our

representation object is, in fact, an *instrument*. However, there are several differences from other interaction instruments, such as scrollbars or graphic primitive creation instruments, namely:

— a representation does not *modify* the domain object;
— a representation is *constantly active* and does not require user action, if not for creating it;
— applying polymorphic instruments to a representation may have different effects depending on whether it is considered domain object or not.

Modification — while a representation could be considered as a special instrument that modifies the domain object to that other instruments can be applied, this interpretation hardly scales for an interactive system with *multiple* representations of the same domain object. It would correspond to having multiple copies of the domain object and a non-trivial *synchronization* procedure when external instruments are applied to representations. This synchronization procedure would become a special instrument-like object with the same properties as the representation.

Activation — an instrument is activated by the user before application, e. g. selecting an instrument from a tool panel corresponds to *temporal* activation. Representations are constantly active and reflect changes in the domain objects independently of user's actions. In a system where the user can create and destroy representations, the fact of creating a new representation would correspond to activating an instrument. However, in systems with fixed representations, it happens independently of user's action or will.

Polymorphism — instruments may be polymorphic, i. e. operate differently on different domain objects. If an instrument is applied to a representation, it may act on the represented domain object or on the properties of the representation. In the former case, the user's object of interest remains the same, while in the latter case, the representation itself becomes an object of interest. For example, a resize instrument in our program restructuring system may either make the visual representation bigger or make the represented loop iterate more.

REPRESENTATION HIERARCHIES AND CHANGE PROPAGATION

Representations can build on another representations rather than on the domain objects directly if they require new data or its structure created by the representation. The polymorphic instruments become applicable to any of the representation or to the initial domain object. Each representation "layer" may either react to the user action or proxy it through to the next level. Multiple different representations can share the same level of the hierarchy providing the user with different ways to interact with the domain objects [99].

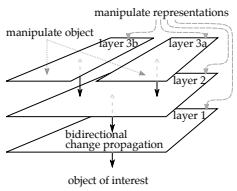


Figure 94 illustrates the representation hierarchy in polyhedral visual program restructuring. The program, or more specifically its

control and data flow, are the central object of interest. The user may interact with either the polyhedral representation stack or with the code representation, but not with the program directly. When the user intends to modify the central object of interest, the changes are propagated down the active part of the hierarchy and back to the other representations.

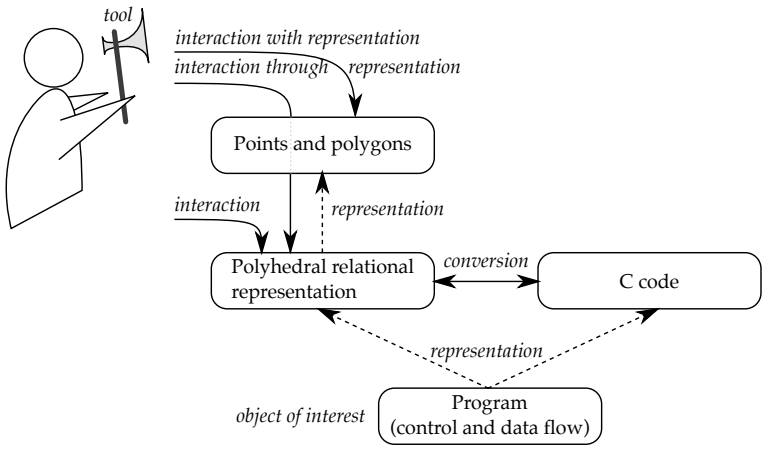


Figure 94 – Using tools to interact with the representation and with the domain object through the representation in interactive program restructuring.

Two complementary views on the multi-layer representation interaction are possible. The user applies the instrument to the deeper levels *through* all representations, focusing on the effect for the domain object, and the changes are propagated from bottom to top in the representation stack. Alternatively, the user applies instruments to the top-level representation and it propagates changes to the deeper levels according to its own rules. In this case, the upper-level representations should be transparent for interaction: a change of the higher-level representation should correspond to the lower-level change such that, if applied directly at the lower level, it would be represented identically to the requested change on the higher level.

A *conversion* procedure allows to map one representation into another without going through a deeper level of hierarchy or the central object of interest.

REPRESENTATIONS AND SUBSTRATES Our notion of representation is close in its spirit to the *information substrate*, which are "*software artifacts that embody content, computation and interaction...*" [142]. Representations embody computation, i. e. the algorithm to create the representation and to propagate the changes back. They may also partially include interaction in a sense that they specify how a particular instrument is applied to the object through them. Rather than embodying content, they represent it in a various shapes thus guiding or limiting the interaction with it.

While information substrates exist on the edge between applications and documents, i. e. on the operating system level, *representations* are initially thought for a single application displaying the same document. However, they are not limited to the same application as long as they represent the same *object*. For example, different file managers executed simultaneously feature different representations of the file system. Moreover, an application may be dedicated to representing an object and such applications combined into a larger interactive system. Further development of the instrumental interaction model and a deeper understanding of the connection between information substrates and representations will allow to design interactive systems that diverge from the typical application-data separation by letting users create, reuse and combine interaction instruments as they do with physical instruments in the real world.

8.4 FUTURE WORK AND PERSPECTIVES

INTERACTIVE PROGRAM MANIPULATION Given multiple representations of the program, of the optimization algorithm and of its result, it is important to focus on their combination and coordination. As suggested by our results, using multiple representations that are not adapted to the task may be less efficient, in particular when the mapping between representations is not obvious. Designing techniques that help users navigate different representations may alleviate this problem. For example, providing morphing animations between two states of the same representation or between two different representations. View coordination is another substantial aspect of multi-representation interfaces that may be considered jointly with navigating in representations.

A far-fetched development of the interactive program transformation is an interactive system constructor: the program is represented as a set of manipulable information sources that can be mapped to the information communication channels (visual cues, audio, haptics, etc.) that feature the matching information structure and afford manipulation. In this case, the developer can construct a tailored interactive view of the relevant aspects of the program and restructure the program through it replacing components at will.

This constructor approach will allow to diverge from the traditional "application" view of the integrated development environments to give developers more power and control over their tools. It is consistent with the instrumental interaction paradigm where program manipulation algorithms and tools as well as representations are instruments that can be used independently in many contexts. Developing interactive program restructuring tools may also help extending the instrumental interaction paradigm to better model the complex multi-

level domain objects with constantly shifting user focus. In a broader perspective, making program manipulation tools substantially easier and more widely available may blur the difference between programming and software appropriation by the extreme users [145, 173], making software easier to adapt, customize and reuse.

SEMI-AUTOMATIC OPTIMIZING COMPILATION An immediate extension of our work on high-level directives for program transformation in *Clay* and *Chlore* consists in a detailed study of the transformation directives, their sets and sequences. Namely, *Clay* set is redundant by design, but other transformation sets with limitations on the particular transformation arguments may be proposed to avoid redundancy. One can design a user study to evaluate whether redundancy or other properties of the transformation set influence the usability of the transformation engine. At the same time, one may study the understanding of the new transformations we proposed in comparison with existing, classical loop transformations.

Transformation sequences recovered by *Chlore* feature another possibility for conducting user studies and making the algorithm output more user-friendly. First, it requires understanding of what the developers find difficult in the transformation. For instance, the trade-off between the transformation length and the complexity of individual transformation semantics should be considered. Second, it requires metrics of transformation sequence quality to be defined following user observation and analysis. Finally, it requires an algorithm that reworks a transformation sequence to improve the metric.

The reification of the program transformation into a script and its subsequent decoupling from the optimizer and the program itself open interesting possibilities for optimization. In particular, transformation scripts can be stored, tweaked and reused for different program parts with similar characteristics without re-analyzing them. The definition of these characteristics as well as the applicability of the transformation scripts to modified program parts is one of the possible research directions.

Decoupling of the transformation from the optimizing algorithm should be extended to other optimization passes of the compilation. Many of them already operate in syntactic terms making it easier to express as directives. However, it may require special representations, textual or visual, to make the developers understand and manipulate these transformations efficiently. With advanced integration into the development environment, developers will be given precise control over the optimization process.

The problem of combining polyhedral and syntactic program transformation has recently come into research focus [225]. Our method

of expressing polyhedral transformation as syntactic directives can make the integration more seamless while maintaining the analysis power of the polyhedral model. From the user's point of view, the program optimization is presented through a unified interface with varying level of precision. He may complete the analyses by higher-level semantic information about the intended program behavior so that the optimizer takes into account this supplementary information.

Making the transformation sequence available in the program rather than a temporary entity inside the optimizing compiler allows to integrate the architecture-related and profiling information more explicitly. Instead of requiring the developer to specify all available information about the program beforehand, it allows him to receive system's feedback on which information is important for optimization. The script-based approach also allows to communicate intent, for example specify why a particular tiling size was chosen manually or why an automatic choice was overridden.

A generalization of the semi-automatic approach may allow to redistribute compilation time and developer's work time. While a compiler continues to run heavy precise analyses, first results can be submitted for the developer to analyze and tweak forming a sort of pipeline in developer-compiler interaction. Furthermore, the heavy analyses to create an optimizing transformation sequence can be run separately from the transformation system, even on a different, more powerful computer system, while a relatively easy transformation application process may happen locally and more often than the analysis giving room for exploration strategy.

INTERACTIVE RESTRUCTURING FOR LEARNING PROGRAMMING

A different development of the interactive program restructuring techniques may target less experienced programmers. Indeed, interactive visualizations allowed to lower the perceptual complexity of an advanced mathematical model of the programs without loosing its power. Applying this idea to other intricate programming concepts may allow to introduce them to the more general public more effectively, and smooth transitions between multiple representations may allow to use the actual production programming languages from the start rather than relying on simplified teaching systems. For example, animated replay of the transformation sequences bears some resemblance with early algorithm animation techniques, but is built on top of a production-level programming system. Understanding the benefits of interactive program restructuring for teaching will require a longitudinal study of the corresponding techniques.

Generalizing the interactive program restructuring, we promote a human-centric approach to designing software development tools that support the entire software life cycle. It consists in building adaptable and tweakable programming tools based on user studies and developer needs rather than on the system requirements. Given available computing power, such tools will allow improving programmer's performance along with the program performance thanks to better support for moving between abstraction levels. They will give the user more control without requiring them to control everything thanks to the human-machine partnership. We propose to revisit the direct manipulation — "*a step beyond programming languages*" that became a common interaction style, — and finally make the step from programming languages to ecosystems of interconnected program manipulation tools. Interactive program restructuring will support developers throughout the software lifecycle by combining software analysis and manipulation in a single consistent interactive system.

A

EXTENDED INFINT DESIGN SPACE

This appendix presents the extension of the InfInt Design Space presented in Chapter 2. In addition to *Information* and *Interaction* dimensions, it includes the description of the visualization itself. The extended version of the tool analysis in the design space is represented in Figure 95 and continued in Figure 96.

The InfInt design space uses the following values of the *Information* dimension:

— code line-based statistics;
— program slicing and dicing;
— object-oriented hierarchies;
— execution traces;
— multithreading or multiprocessing synchronization;
— communication in distributed programming models or environments;
— memory management;
— data structures;
— control structures;
— numeric metrics;
— low-level software aspects.

This dimension corresponds to the horizontal axis on the figures.

Interaction dimension may take the following values and is represented with a color code:

Static — no interaction is possible with this information, the visual representation is a static, non-navigable picture (black);

General — the user is able to interact with the visual representation itself, but not with the program being visualized, using conventional techniques for this kind of visualization (dark red);

Specific — the user is able to interact with the visual representation itself, but not with the program, using interaction techniques that take into account the specificity of the underlying software-related data (blue);

Restructuring — the user is able to change the underlying program by interacting with its visual representation (light green).

A software visualization technique is represented in the extended InfInt design space by a row. If the technique visualizes certain informational aspect of the software, the corresponding cell (intersection of the technique row and *Information* column) describes how this aspect is visualized. Its color shows the level of *Interaction* with this particular informational aspect. If multiple representations are provided for the aspect or it is only partly visualized, it may be specified by

italic text followed by → representing the mapping. The description language used in Figures 95,96 is vaguely inspired by [45]. However, we also use a higher-level description for commonly used visualization techniques [120] with the following abbreviations:

xy — 2D coordinates;
xyz — 3D coordinates;
C — color;
O — opacity;
Sz — sizes;
NL — node-link diagram;
UML — UML class diagram;
Seq — sequence diagram (including UML sequence diagram);
Gtt — Gantt chart;
Scatter — Scatter plot;
Parallel — parallel coordinates plot;
Indexed — indexed (conventional x-y) plot;
Icicle — icicle diagram.

We do not mention basic visual cues (coordinates, color, sizes) if the visualization relies on them, e. g. coordinates and sizes are essential for Gantt chart. In case of multiple cues or visualizations available for the same information aspect, we separate them by commas. If the visual display is a combination of multiple visualizations, we connect them by × symbol. For example, a combination of Gantt chart and a node-link diagram where each segment of the Gantt diagram is a node potentially connected to other nodes, typical for representing traces and communication, is encoded as Gtt × NL.

Although a more detailed analysis using the structure proposed by Card and Mackinlay [45] and the exact mapping to the quantifiable informational aspects of the software may provide more insight into building software visualizations that support interactive restructuring, we argue that higher-level description of the visualizations allows to better compare techniques with each other. As these visualizations are vastly described in the literature, the technique designer may have a good intuition as to the appropriate interactions, e. g. node-link diagram easily allows for rearranging nodes by direct manipulation or for editing links through instrumental interaction.

	Code	Slicing	Hierarchies	Traces	Sync.	Comm.	Memory	Data	Control	Metrics	Low-level
SeeSoft	*indent* →C,O,*xy*								*type*→C,O	C,O	
SeeSlice	C,O	*deps*→NL								C,O	
Sv3D	C,O,*xyz*,*Sz*								*type*→C,O, *xyz*,*Sz*	C,O,*xyz*,*Sz*	
Tarantula	C,O								*tests*,*coverage* →C,O		
SHriMP			NL				*access*→NL		*call*→NL		
JAVAVIS			NL	*single thread* →Gtt	*Synchronized* →Seq			*structure* →NL			
JaVis			NLxUML	Seq	*Synchronized* →Seq				*call* →Seq x UML		
Jinsight			NL	Gtt			*access pattern* →NLxC		*call*→Gtt		
Slice Browser		*deps*→NL									
Krinke 2004		NL,Scatter							*deps*→Scatter		
Mem. Graphs							*access*→NL	*structure* →NL			
Star Diagram							*alloc*,*access* →NL	*structure* →NL	*functions*		
Gthread				Gtt	*mutex*→Venn, *barrier*→ Scatter						

Figure 95 – Extended InfInt Design Space: a row corresponds to a software visualization technique, columns represents values of *Information* dimension, cell entries contain indication of the visual representation color coded by *Interaction*.

	Code	Slicing	Hierarchies	Traces	Sync.	Comm.	Memory	Data	Control	Metrics	Low-level
Jerding 1997						NL x Gtt	*access*→ Scatter				
MAPA							*access*→ Scatter				*iteration* →scatter
CodeCrawler			NL							NL x Sz x C xy	
GCSpy							*allocation* →Scatter				
Beck 2011			NL x Parallel Coordinates							NL x Parallel Coordinates	
Shimba			NL x indexed							indexed	
Vampir				Gtt		NL x Gtt					
3D ISV								*deps* →NL x Scatter			*iteration*→NL x Scatter
SyncTrace				Gtt x SunBurst	NL x Gtt						
Deco							*access*→ Scatter			histogram, pie chart	*iteration* →scatter
Veroy 2013							*allocation* →hive plot				
Tulipse								*deps* →NL x Scatter	*call*→NL		*iteration*→NL x Scatter
Tiled Grace									*Visual Language* →tiles		

Figure 96 – Extended Inflnt Design Space (continued).

B STATISTICAL METHODS

This appendix briefly describes the statistical methods we use to analyze and report data: effect sizes and confidence intervals. It serves only as a presentation and does not provide full calculations or proofs, which can be found in, e.g., [218].

B.1 KEY PROBABILITY DISTRIBUTIONS

NORMAL DISTRIBUTION

Normal distribution is a continuous probability distribution often observed in nature and commonly used for statistical analysis. It describes the values distributed around a certain *mean* value μ, with the probability of a value exponentially decreasing with its distance from the mean. This decrease is characterized by the standard deviation σ.

Central Limit Theorem states that the mean of an infinitely large number of samples drawn from arbitrary distributed random variable is distributed normally, independently of the source distributions, making normal distribution ubiquitous in experiment analysis. Indeed, the measures observed during the human-computer interaction experiments may be influenced by numerous aspects of human behavior or system properties. Therefore, they are likely to be (approximately) normally distributed.

STUDENT'S T DISTRIBUTION

The mean of a finite number n of random samples drawn from the normal distribution is obeying the t_ν distribution, where the parameter $\nu = n - 1$ specifies the *degrees of freedom*. The shape of the probability density function of a t-distribution becomes increasingly close to the normal distribution for larger ν values: for $\nu = \infty$, t-distribution is equivalent to normal distribution (Figure 97).

LOG-NORMAL DISTRIBUTION AND LOG-TRANSFORM HCI experiments often measure time intervals to compare tool efficiency, for example task completion times with different visualizations. Yet, intervals and durations are always non-negative and thus cannot be distributed normally: normal distribution allows for any value arbitrarily far from the mean, although with very low probabilities. A log-normal distribution of a random variable means that its natural logarithm is distributed normally. Log-normal distribution does not allow for negative values and is generally better suited for time intervals.

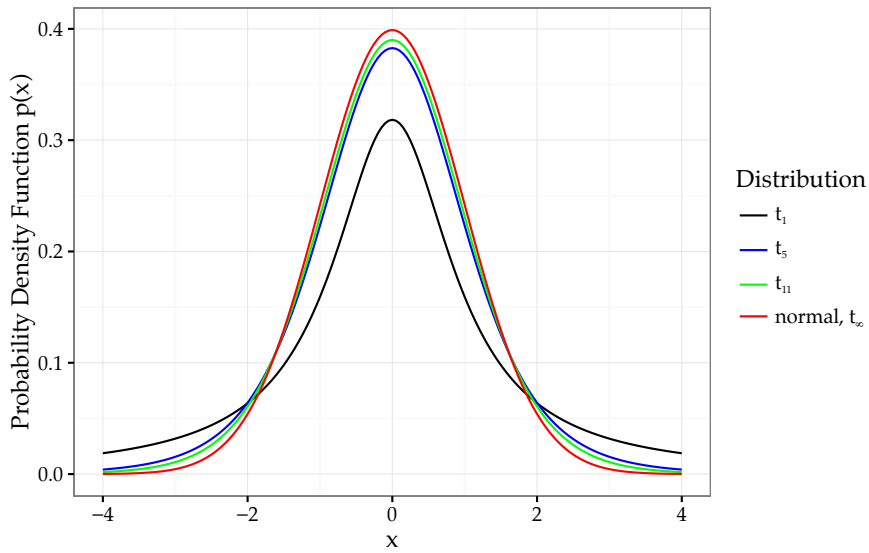


Figure 97 – Probability density functions of the normal distribution with $\mu = 0, \sigma = 1$ and t-distribution with $\nu = 1, 5, 11$.

In practice, we rely on *log-transform* for analyzing time interval data. It consists in taking a natural logarithm of the raw values before analysis and performing the inverse (exponent) manipulation before reporting the results. This results, in particular, in the mean being less affected by individual large values and in confidence intervals being asymmetric.

B.2 CONFIDENCE INTERVALS

Confidence intervals (CIs) help estimating the unknown value, e. g. the mean or difference of means, given the finite number of samples. In particular, a 95% confidence interval of a mean will include the mean of the samples with a 95% probability *if the measurement is repeated with the same conditions.* Note that CIs do not allow to reason about the *true* mean of the unknown value. For symmetric distributions, including normal and t-distributions, confidence intervals are also symmetric and are conventionally written as $\mu \pm c$ meaning the confidence interval is $[\mu - c, \mu + c]$.

For n independent samples $\{s_1, s_2, \ldots, s_n\}$ of a normally distributed random variable with sample mean $m = \frac{1}{n}\sum_{i=1}^{n} s_i$ and sample variance $s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - m)^2$, we defined the *standardized* mean distribution as

$$T = \frac{m - \mu}{s/\sqrt{n}},$$

which has a t-distribution with $(n-1)$ degrees of freedom (*standard* normal distribution has mean 0 and standard deviation 1). To com-

pute the 95% confidence intervals, we need to find the value x such that $\Pr(-x \leqslant T \leqslant x) = 0.95$, i.e. there is 95% probability that the random sample of standardized mean T is within the interval. Conversely, there is 5% probability that a random sample of T is outside $[-x, x]$ interval, i.e. $\Pr(T < -x) + \Pr(T > x) = 0.05$. This 5% probability is called α-value. Given that the t-distribution has central symmetry, we can assume $\Pr(T < -x) = \Pr(T > x) = 0.05/2 = 0.025$. We can compute the value of x from the inverse cumulative probability function for the t-distribution, $F_{11}^{-1}(0.025) \approx 2.2$. Hence,

$$\Pr(-2.2 \leqslant T \leqslant 2.2) = 0.95.$$

In order to obtain intervals around the original (non-standardized) mean μ, we substitute T with its definition

$$\Pr(-2.2 \leqslant \frac{m-\mu}{s/\sqrt{n}} \leqslant 2.2) = 0.95;$$

$$\Pr(m - 2.2\frac{s}{\sqrt{n}} \leqslant \mu \leqslant m + 2.2\frac{s}{\sqrt{n}}) = 0.95,$$

or $\mu = m \pm 2.2\frac{s}{\sqrt{n}}$ in the short form.

B.3 EFFECT SIZES AND HYPOTHESIS TESTING

SYMMETRIC EFFECT SIZE

If an experiment factor affects the normally distributed measures, their means differ $\mu_1 \neq \mu_2$. Yet, it may be complex to straightly complain means. Therefore, we define the *symmetric effect size* as the difference of means normalized by their average

$$\mathrm{ES} = \frac{\mu_1 - \mu_2}{(\mu_1 + \mu_2)/2} \cdot 100\%.$$

Normalizing by the average allows to better account for cases where one mean is substantially larger than the other, which is the common case in experimental data.

CONFIDENCE INTERVAL OF THE EFFECT SIZE

Effect sizes alone do not give a sufficiently robust estimation of the effect since means of measures are subject to uncertainty given finite number of samples in the experiment. Therefore, we compute confidence intervals of the effect size. We use the Tukey method to calculate confidence intervals for the difference of means and than scale it linearly to obtain CIs of the effect size.

The CI for difference of means is obtained from the studentized range distribution q, which characterizes the pairwise difference of means of k samples of size n from the normal distribution,

$$q = \frac{\mathrm{abs}(\mu_1 - \mu_2)}{\hat{\sigma}\sqrt{2/n}},$$

where $\hat{\sigma}^2 = \frac{1}{k}\sum_{i=1}^{k}\frac{k}{n}\sum_j (x_{i,j} - \mu_i)^2$ is the mean squared error, i.e. the sum of squares of differences between the observed value and the mean of *its sample*. The q-distribution is parameterized by two degrees of freedom $k-1$ and $n-k$ that depend on the number and size of samples.

Similarly to the confidence intervals for the means, we can obtain the 95% confidence interval from the inverse cumulative probability function of the q-distribution as

$$(\mu_1 - \mu_2) \pm \frac{\hat{\sigma}}{\sqrt{2}} Q^{-1}_{1-0.95,k-1,n-k}\sqrt{\frac{2(k-1)}{n}}.$$

The endpoints of the interval can be then multiplied by $\frac{2}{\mu_1+\mu_2}$ to obtain the CI of the effect size.

ESTIMATION FOR HYPOTHESIS TESTING

In a hypothesis testing experiment, one formulates a null-hypothesis opposite to the expected results, e. g. using direct manipulation does not decrease program restructuring time. Then the experimental data is analyzed to show that the null-hypothesis is *unlikely* to be true.

Confidence intervals allow to *estimate* the hypothesis in the experiment. In particular, a null-hypothesis may stipulate that the *true* mean is 0 (or any other number). If the 95% confidence interval of the measure mean does not contain 0, there is less than 5% probability that rerunning the experiment will result in a measure mean 0. Hence, the null-hypothesis is likely to be false and may be rejected at $\alpha = 0.05$ level, i.e. 5% probability of rejecting the correct null-hypothesis and observing a *false positive* effect. The farther is 0 from the confidence interval bounds, the less likely is the null-hypothesis.

Similarly, one can estimate whether the change in factor affects the measure using the confidence interval of the effect size with the null-hypothesis of 0 effect size.

Contrary to null hypothesis significance testing, that typically results in a binary confirmation/rejection of the null hypothesis, *estimation* provides more nuanced data interpretation, especially important for small sample sizes and complex measures [82].

BIBLIOGRAPHY

[1] A. Abuthawabeh, F. Beck, D. Zeckzer, and S. Diehl. « Finding Structures in Multi-Type Code Couplings with Node-Link and Matrix Visualizations. » In: *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*. 2013 First IEEE Working Conference on Software Visualization (VISSOFT). Sept. 2013, pp. 1–10.

[2] David Ahlström. « Modeling and Improving Selection in Cascading Pull-down Menus Using Fitts' Law, the Steering Law and Force Fields. » In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '05. New York, NY, USA: ACM, 2005, pp. 61–70.

[3] David Ahlström, Martin Hitz, and Gerhard Leitner. « An Evaluation of Sticky and Force Enhanced Targets in Multi Target Situations. » In: *Proceedings of the 4th Nordic Conference on Human-Computer Interaction: Changing Roles*. NordiCHI '06. New York, NY, USA: ACM, 2006, pp. 58–67.

[4] Francisco J. Almeida-Martínez, Jaime Urquiza-Fuentes, and J. Ángel Velázquez-Iturbide. « VAST: Visualization of Abstract Syntax Trees Within Language Processors Courses. » In: *Proceedings of the 4th ACM Symposium on Software Visualization*. SoftVis '08. New York, NY, USA: ACM, 2008, pp. 209–210.

[5] Jessalyn Alvina, Caroline Appert, Olivier Chapuis, and Emmanuel Pietriga. « RouteLens: Easy Route Following for Map Applications. » In: *Proceedings of the 2014 International Working Conference on Advanced Visual Interfaces*. AVI '14. New York, NY, USA: ACM, 2014, pp. 125–128.

[6] Corinne Ancourt and François Irigoin. « Scanning Polyhedra with DO Loops. » In: *ACM Sigplan Notices*. Vol. 26. ACM, 1991, pp. 39–50.

[7] Ferran Argelaguet, David Antonio Gómez Jáuregui, Maud Marchal, and Anatole Lécuyer. « Elastic Images: Perceiving Local Elasticity of Images through a Novel Pseudo-Haptic Deformation Effect. » In: *ACM Transactions on Applied Perception (TAP)* 10.3 (2013), p. 17.

[8] Anna Armentrout. « A Tool for Designing Java Programs with UML. » In: *Proceedings of the 4th Annual SIGCSE/SIGCUE ITiCSE Conference on Innovation and Technology in Computer Science Education*. ITiCSE '99. New York, NY, USA: ACM, 1999, pp. 180–.

[9] Ronald M. Baecker. « Sorting Out Sorting: A Case Study of Software Visualization for Teaching Computer Science. » In: *Software Visualization: Programming as a Multimedia Experience*. Ed. by John T. Stasko. MIT Press, 1998, pp. 369–381.

[10] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. « The Parma Polyhedra Library: Toward a Complete Set of Numerical Abstractions for the Analysis and Verification of Hardware and Software Systems. » In: *Science of Computer Programming*. Special Issue on Second issue of experimental software and toolkits (EST) 72 (1–2 June 1, 2008), pp. 3–21.

[11] Lénaïc Bagnères and Cédric Bastoul. « Switchable Scheduling for Runtime Adaptation of Optimization. » In: *Euro-Par 2014 Parallel Processing*. Ed. by Fernando Silva, Inês Dutra, and Vítor Santos Costa. Lecture Notes in Computer Science 8632. Springer International Publishing, Aug. 25, 2014, pp. 222–233.

[12] Lénaïc Bagnères, Oleksandr Zinenko, Stéphane Huot, and Cédric Bastoul. « Opening Polyhedral Compiler's Black Box. » In: *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. CGO 2016. New York, NY, USA: ACM, 2016, pp. 128–138.

[13] T. Ball and S. G. Eick. « Visualizing Program Slices. » In: *, IEEE Symposium on Visual Languages, 1994. Proceedings*. , IEEE Symposium on Visual Languages, 1994. Proceedings. Oct. 1994, pp. 288–295.

[14] Michael Balzer, Andreas Noack, Oliver Deussen, and Claus Lewerentz. « Software Landscapes: Visualizing the Structure of Large Software Systems. » In: *Proceedings of the Sixth Joint Eurographics - IEEE TCVG Conference on Visualization*. VISSYM'04. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2004, pp. 261–266.

[15] R. C. Barrett, E. J. Selker, J. D. Rutledge, and R. S. Olyha. « Negative Inertia: A Dynamic Pointing Function. » In: *Conference Companion on Human Factors in Computing Systems*. CHI '95. New York, NY, USA: ACM, 1995, pp. 316–317.

[16] Cedric Bastoul. « Code Generation in the Polyhedral Model Is Easier Than You Think. » In: *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*. PACT '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 7–16.

[17] Cédric Bastoul. *OpenScop: A Specification and a Library for Data Exchange in Polyhedral Compilation Tools.* Technical Report. Orsay, France: Paris-Sud University, Sept. 2011, p. 47.

[18] Cédric Bastoul. « Mapping Deviation: A Technique to Adapt or to Guard Loop Transformation Intuitions for Legality. » In: *Proceedings of the 25th International Conference on Compiler Construction*. CC 2016. New York, NY, USA: ACM, 2016, pp. 229–239.

[19] Cédric Bastoul and Paul Feautrier. « Improving Data Locality by Chunking. » In: *Compiler Construction*. Ed. by Görel Hedin. Lecture Notes in Computer Science 2622. Springer Berlin Heidelberg, Apr. 7, 2003, pp. 320–334.

[20] Cédric Bastoul, Albert Cohen, Sylvain Girbal, Saurabh Sharma, and Olivier Temam. « Putting Polyhedral Loop Transformations to Work. » In: *Languages and Compilers for Parallel Computing*. Ed. by Lawrence Rauchwerger. Lecture Notes in Computer Science 2958. Springer Berlin Heidelberg, Oct. 2, 2003, pp. 209–225.

[21] Olivier Bau and Wendy E. Mackay. « OctoPocus: A Dynamic Guide for Learning Gesture-Based Command Sets. » In: *Proceedings of the 21st Annual ACM Symposium on User Interface Software and Technology*. UIST '08. New York, NY, USA: ACM, 2008, pp. 37–46.

[22] Patrick Baudisch, Edward Cutrell, Ken Hinckley, and Adam Eversole. « Snap-and-Go: Helping Users Align Objects Without the Modality of Traditional Snapping. » In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '05. New York, NY, USA: ACM, 2005, pp. 301–310.

[23] Michel Beaudouin-Lafon. « Instrumental Interaction: An Interaction Model for Designing Post-WIMP User Interfaces. » In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '00. New York, NY, USA: ACM, 2000, pp. 446–453.

[24] Michel Beaudouin-Lafon. « Designing Interaction, Not Interfaces. » In: *Proceedings of the Working Conference on Advanced Visual Interfaces*. AVI '04. New York, NY, USA: ACM, 2004, pp. 15–22.

[25] Michel Beaudouin-Lafon and Henry Michael Lassen. « The Architecture and Implementation of CPN2000, a Post-WIMP Graphical Application. » In: *Proceedings of the 13th Annual ACM Symposium on User Interface Software and Technology*. UIST '00. New York, NY, USA: ACM, 2000, pp. 181–190.

[26] Michel Beaudouin-Lafon and Wendy E. Mackay. « Reification, Polymorphism and Reuse: Three Principles for Designing Visual Interfaces. » In: *Proceedings of the Working Conference on Advanced Visual Interfaces*. AVI '00. New York, NY, USA: ACM, 2000, pp. 102–109.

[27] F. Beck, R. Petkov, and S. Diehl. « Visually Exploring Multi-Dimensional Code Couplings. » In: *2011 6th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*. 2011 6th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT). Sept. 2011, pp. 1–8.

[28] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. « The Polyhedral Model Is More Widely Applicable than You Think. » In: *Compiler Construction*. Springer, 2010, pp. 283–303.

[29] Thomas J. Bergin and Richard G. Gibson, eds. *History of Programming Languages, Volume 2*. 1 edition. New York : Reading, Mass: Addison-Wesley Professional, Feb. 22, 1996. 864 pp.

[30] A. J. Bernstein. « Analysis of Programs for Parallel Processing. » In: *IEEE Transactions on Electronic Computers* EC-15.5 (Oct. 1966), pp. 757–763.

[31] Jacques Bertin. *Semiology of Graphics: Diagrams, Networks, Maps*. University of Wisconsin Press, 1983.

[32] Eric A. Bier and Maureen C. Stone. « Snap-Dragging. » In: *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '86. New York, NY, USA: ACM, 1986, pp. 233–240.

[33] Alan F. Blackwell, Kirsten N. Whitley, Judith Good, and Marian Petre. « Cognitive Factors in Programming with Diagrams. » In: *Artificial Intelligence Review* 15 (1-2 2001), pp. 95–114.

[34] Renaud Blanch, Yves Guiard, and Michel Beaudouin-Lafon. « Semantic Pointing: Improving Target Acquisition with Control-Display Ratio Adaptation. » In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '04. New York, NY, USA: ACM, 2004, pp. 519–526.

[35] Hans-J. Boehm. « Threads Cannot Be Implemented As a Library. » In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '05. New York, NY, USA: ACM, 2005, pp. 261–268.

[36] Uday Bondhugula, Aravind Acharya, and Albert Cohen. « The Pluto+ Algorithm: A Practical Approach for Parallelization and Locality Optimization of Affine Loop Nests. » In: *ACM Trans. Program. Lang. Syst.* 38.3 (Apr. 2016), 12:1–12:32.

[37] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. « A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. » In: *ACM SIGPLAN Notices*. Vol. 43. ACM, 2008, pp. 101–113.

[38] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, Jagannathan Ramanujam, Atanas Rountev, and Ponnuswamy Sadayappan. « Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model. » In: *Compiler Construction*. Springer, 2008, pp. 132–146.

[39] Uday Bondhugula, Oktay Gunluk, Sanjeeb Dash, and Lakshminarayanan Renganarayanan. « A Model for Fusion and Code Motion in an Automatic Parallelizing Compiler. » In: *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*. PACT '10. IBM XL. New York, NY, USA: ACM, 2010, pp. 343–352.

[40] Marat Boshernitsan and Michael S. Downes. *Visual Programming Languages: A Survey*. Technical Rept. UCB/CSD-04-1368. University of California at Berkeley, Berkeley, California, Dec. 1, 2004, p. 25.

[41] Pierre Boulet, Alain Darte, Georges-André Silber, and Frédéric Vivien. « Loop Parallelization Algorithms: From Parallelism Extraction to Code Generation. » In: *Parallel Computing* 24 (3–4 May 1998), pp. 421–444.

[42] Robert W. Bowdidge and William G. Griswold. « Supporting the Restructuring of Data Abstractions Through Manipulation of a Program Visualization. » In: *ACM Trans. Softw. Eng. Methodol.* 7.2 (Apr. 1998), pp. 109–157.

[43] Orlie Brewer, Jack Dongarra, and Danny Sorensen. « Tools to Aid in the Analysis of Memory Access Patterns for FORTRAN Programs. » In: *Parallel Computing* 9.1 (Dec. 1988), pp. 25–35.

[44] M. H. Brown and R. Sedgewick. « Techniques for Algorithm Animation. » In: *IEEE Software* 2.1 (Jan. 1985), pp. 28–39.

[45] S. K. Card and J. Mackinlay. « The Structure of the Information Visualization Design Space. » In: *, IEEE Symposium on Information Visualization, 1997. Proceedings*. , IEEE Symposium on Information Visualization, 1997. Proceedings. Oct. 1997, pp. 92–99.

[46] Stuart K. Card, Jock D. Mackinlay, and Ben Shneiderman, eds. *Readings in Information Visualization: Using Vision to Think*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999.

[47] Sheelagh Carpendale and Yaser Ghanam. « A Survey Paper on Software Architecture Visualization. » In: *Technical Report* (June 19, 2008).

[48] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. « Compiler Optimizations for Improving Data Locality. » In: *ACM SIGPLAN Notices*. Vol. 29. ACM, 1994, pp. 252–262.

[49] P. Caserta and O. Zendra. « Visualization of the Static Aspects of Software: A Survey. » In: *IEEE Transactions on Visualization and Computer Graphics* 17.7 (July 2011), pp. 913–933.

[50] Géry Casiez and Nicolas Roussel. « No More Bricolage!: Methods and Tools to Characterize, Replicate and Compare Pointing Transfer Functions. » In: *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*. UIST '11. New York, NY, USA: ACM, 2011, pp. 603–614.

[51] Géry Casiez, Daniel Vogel, Ravin Balakrishnan, and Andy Cockburn. « The Impact of Control-Display Gain on User Performance in Pointing Tasks. » In: *Human–Computer Interaction* 23.3 (2008), pp. 215–250.

[52] Bay-Wei Chang and David Ungar. « Animation: From Cartoons to the User Interface. » In: *Proceedings of the 6th Annual ACM Symposium on User Interface Software and Technology*. UIST '93. New York, NY, USA: ACM, 1993, pp. 45–55.

[53] Bay-Wei Chang, Jock D. Mackinlay, Polle T. Zellweger, and Takeo Igarashi. « A Negotiation Architecture for Fluid Documents. » In: *Proceedings of the 11th Annual ACM Symposium on User Interface Software and Technology*. UIST '98. New York, NY, USA: ACM, 1998, pp. 123–132.

[54] Olivier Chapuis, Jean-Baptiste Labrune, and Emmanuel Pietriga. « DynaSpot: Speed-Dependent Area Cursor. » In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '09. New York, NY, USA: ACM, 2009, pp. 1391–1400.

[55] P. Chatarasi, J. Shirako, and V. Sarkar. « Polyhedral Optimizations of Explicitly Parallel Programs. » In: *2015 International Conference on Parallel Architecture and Compilation (PACT)*. 2015 International Conference on Parallel Architecture and Compilation (PACT). Oct. 2015, pp. 213–226.

[56] B. Chazelle. « Convex Partitions of Polyhedra: A Lower Bound and Worst-Case Optimal Algorithm. » In: *SIAM Journal on Computing* 13.3 (Aug. 1, 1984), pp. 488–507.

[57] Chun Chen. « Polyhedra Scanning Revisited. » In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '12. New York, NY, USA: ACM, 2012, pp. 499–508.

[58] Chun Chen, Jacqueline Chame, and Mary Hall. *CHiLL: A Framework for Composing High-Level Loop Transformations*. 2008.

[59] Fanny Chevalier, David Auber, and Alexandru Telea. « Structural Analysis and Visualization of C++ Code Evolution Using Syntax Trees. » In: *Ninth International Workshop on Principles of Software Evolution: In Conjunction with the 6th ESEC/FSE Joint*

Meeting. IWPSE '07. New York, NY, USA: ACM, 2007, pp. 90–97.

[60] Fanny Chevalier, Pierre Dragicevic, and Christophe Hurter. « Histomages: Fully Synchronized Views for Image Editing. » In: *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology*. UIST '12. New York, NY, USA: ACM, 2012, pp. 281–286.

[61] Fanny Chevalier, Pierre Dragicevic, Anastasia Bezerianos, and Jean-Daniel Fekete. « Using Text Animated Transitions to Support Navigation in Document Histories. » In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '10. New York, NY, USA: ACM, 2010, pp. 683–692.

[62] Carlos Christensen. « An Example of the Manipulation of Directed Graphs in the AMBIT/G Programming Language. » In: *Symposium on Interactive Systems for Experimental Applied Mathematics: Proceedings of the Association for Computing Machinery Inc. Symposium*. New York, NY, USA: ACM, 1967, pp. 423–435.

[63] Carlos Christensen. « An Introduction to AMBIT/L, a Diagrammatic Language for List Processing. » In: *Proceedings of the Second ACM Symposium on Symbolic and Algebraic Manipulation*. SYMSAC '71. New York, NY, USA: ACM, 1971, pp. 248–260.

[64] Philippe Clauss and Benoît Meister. « Automatic Memory Layout Transformations to Optimize Spatial Locality in Parameterized Loop Nests. » In: *SIGARCH Comput. Archit. News* 28.1 (Mar. 2000), pp. 11–19.

[65] Andy Cockburn and Stephen Brewster. « Multimodal Feedback for the Acquisition of Small Targets. » In: *Ergonomics* 48.9 (July 15, 2005), pp. 1129–1150. pmid: 16251152.

[66] Andy Cockburn and Philip Brock. « Human On-Line Response to Visual and Motor Target Expansion. » In: *Proceedings of Graphics Interface 2006*. GI '06. Toronto, Ont., Canada, Canada: Canadian Information Processing Society, 2006, pp. 81–87.

[67] Andy Cockburn and Bruce McKenzie. « 3D or Not 3D?: Evaluating the Effect of the Third Dimension in a Document Management System. » In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '01. New York, NY, USA: ACM, 2001, pp. 434–441.

[68] « Computer-Implemented Method for Generating a UML Representation from JAVA Source Code. » Pat. US6560769 B1. Jeffrey Allen Moore, Joseph Peter Stefaniak, and Lonnie Dale Sulgrove. U.S. Classification 717/100, 717/106, 717/108, 717/101, 717/118, 707/999.1; International Classification G06F9/44; Co-

operative Classification G06F8/74; European Classification G06F8/74. May 6, 2003.

[69] Stéphane Conversy. « Existe-T-Il Une DiffÉRence Entre Langages Visuels Et Textuels En Termes De Perception? » In: *Proceedings of the 25th Conference on L'Interaction Homme-Machine*. IHM '13. New York, NY, USA: ACM, 2013, 53:53–53:58.

[70] Stéphane Conversy. « Unifying Textual and Visual: A Theoretical Account of the Visual Perception of Programming Languages. » In: *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. Onward! 2014. New York, NY, USA: ACM, 2014, pp. 201–212.

[71] Stéphane Conversy, Stéphane Chatty, and Christophe Hurter. « Visual Scanning as a Reference Framework for Interactive Representation Design. » In: *Information Visualization* 10.3 (July 1, 2011), pp. 196–211.

[72] G. Convertino, J. Chen, B. Yost, Y. S. Ryu, and C. North. « Exploring Context Switching and Cognition in Dual-View Coordinated Visualizations. » In: *International Conference on Coordinated and Multiple Views in Exploratory Visualization, 2003. Proceedings*. International Conference on Coordinated and Multiple Views in Exploratory Visualization, 2003. Proceedings. July 2003, pp. 55–62.

[73] Geoff Cumming. « The New Statistics Why and How. » In: *Psychological Science* (Nov. 12, 2013), p. 0956797613504966. pmid: 24220629.

[74] Geoff Cumming and Sue Finch. « Inference by Eye: Confidence Intervals and How to Read Pictures of Data. » In: *American Psychologist* 60.2 (2005), pp. 170–180.

[75] George Bernard Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1998. 652 pp.

[76] Alain Darte, Yves Robert, and Frederic Vivien. *Scheduling and Automatic Parallelization*. Google-Books-ID: 95bxBwAAQBAJ. Springer Science & Business Media, Dec. 6, 2012. 275 pp.

[77] Alain Darte and Frédéric Vivien. « Optimal Fine and Medium Grain Parallelism Detection in Polyhedral Reduced Dependence Graphs. » In: *International Journal of Parallel Programming* 25.6 (Dec. 1997), pp. 447–496.

[78] Y. Deng, S. Kothari, and Y. Namara. « Program Slice Browser. » In: *9th International Workshop on Program Comprehension, 2001. IWPC 2001. Proceedings*. 9th International Workshop on Program Comprehension, 2001. IWPC 2001. Proceedings. 2001, pp. 50–59.

[79] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. « Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions. » In: *IEEE Journal of Solid-State Circuits* 9.5 (Oct. 1974), pp. 256–268.

[80] Stephan Diehl. *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Springer Science & Business Media, Apr. 24, 2007. 192 pp.

[81] Sebastien Donadio, James Brodman, Thomas Roeder, Kamen Yotov, Denis Barthou, Albert Cohen, María Jesús Garzarán, David Padua, and Keshav Pingali. « A Language for the Compact Representation of Multiple Program Versions. » In: *Languages and Compilers for Parallel Computing*. Ed. by Eduard Ayguadé, Gerald Baumgartner, J. Ramanujam, and P. Sadayappan. Lecture Notes in Computer Science 4339. Springer Berlin Heidelberg, Oct. 20, 2005, pp. 136–151.

[82] Pierre Dragicevic. « Fair Statistical Communication in HCI. » In: *Modern Statistical Methods for HCI*. Ed. by Judy Robertson and Maurits Kaptein. Human–Computer Interaction Series. Springer International Publishing, 2016, pp. 291–330.

[83] Pierre Dragicevic, Stéphane Huot, and Fanny Chevalier. « Gliimpse: Animating from Markup Code to Rendered Documents and Vice Versa. » In: *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*. UIST '11. New York, NY, USA: ACM, 2011, pp. 257–262.

[84] S. Ducasse and M. Lanza. « The Class Blueprint: Visually Supporting the Understanding of Glasses. » In: *IEEE Transactions on Software Engineering* 31.1 (Jan. 2005), pp. 75–90.

[85] Alistair D. N. Edwards. « Visual Programming Languages: The Next Generation. » In: *SIGPLAN Not.* 23.4 (Apr. 1988), pp. 43–50.

[86] S. C. Eick, J. L. Steffen, and E. E. Sumner. « Seesoft-a Tool for Visualizing Line Oriented Software Statistics. » In: *IEEE Transactions on Software Engineering* 18.11 (Nov. 1992), pp. 957–968.

[87] S. G. Eick, T. L. Graves, A. F. Karr, A. Mockus, and P. Schuster. « Visualizing Software Changes. » In: *IEEE Transactions on Software Engineering* 28.4 (Apr. 2002), pp. 396–412.

[88] Christine Eisenbeis and Jean-Claude Sogno. « A General Algorithm for Data Dependence Analysis. » In: *Proceedings of the 6th International Conference on Supercomputing*. ICS '92. New York, NY, USA: ACM, 1992, pp. 292–302.

[89] G. Ellis and A. Dix. « A Taxonomy of Clutter Reduction for Information Visualisation. » In: *IEEE Transactions on Visualization and Computer Graphics* 13.6 (Nov. 2007), pp. 1216–1223.

[90] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C. North, and Gordon Woodhull. « Graphviz— Open Source Graph Drawing Tools. » In: *Graph Drawing*. Ed. by Petra Mutzel, Michael Jünger, and Sebastian Leipert. Lecture Notes in Computer Science 2265. Springer Berlin Heidelberg, Sept. 23, 2001, pp. 483–484.

[91] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. « Dark Silicon and the End of Multicore Scaling. » In: *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. 2011 38th Annual International Symposium on Computer Architecture (ISCA). June 2011, pp. 365–376.

[92] I. Fassi and P. Clauss. « XFOR: Filling the Gap between Automatic Loop Optimization and Peak Performance. » In: *2015 14th International Symposium on Parallel and Distributed Computing*. 2015 14th International Symposium on Parallel and Distributed Computing. June 2015, pp. 100–109.

[93] Paul Feautrier. « Parametric Integer Programming. » In: *Revue française d'automatique, d'informatique et de recherche opérationnelle. Recherche opérationnelle* 22.3 (1988), pp. 243–268.

[94] Paul Feautrier. « Dataflow Analysis of Array and Scalar References. » In: *International Journal of Parallel Programming* 20.1 (1991), pp. 23–53.

[95] Paul Feautrier. « Some Efficient Solutions to the Affine Scheduling Problem. I. One-Dimensional Time. » In: *International Journal of Parallel Programming* 21.5 (Oct. 1992), pp. 313–347.

[96] Paul Feautrier. « Some Efficient Solutions to the Affine Scheduling Problem. Part II. Multidimensional Time. » In: *International journal of parallel programming* 21.6 (1992), pp. 389–420.

[97] Paul Feautrier. « Bernstein's Conditions. » In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Springer US, 2011, pp. 130–134.

[98] Paul Feautrier and Christian Lengauer. « Polyhedron Model. » In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Springer US, 2011, pp. 1581–1592.

[99] Jean-Daniel Fekete and Michel Beaudouin-Lafon. « Using the Multi-Layer Model for Building Interactive Graphical Applications. » In: *Proceedings of the 9th Annual ACM Symposium on User Interface Software and Technology*. UIST '96. New York, NY, USA: ACM, 1996, pp. 109–118.

[100] Norman Fenton and James Bieman. *Software Metrics: A Rigorous and Practical Approach, Third Edition*. CRC Press, Oct. 1, 2014. 602 pp.

[101] Grigori Fursin, Renato Miceli, Anton Lokhmotov, Michael Gerndt, Marc Baboulin, Allen D. Malony, Zbigniew Chamski, Diego Novillo, and Davide Del Vento. « Collective Mind: Towards Practical and Collaborative Auto-Tuning. » In: *Scientific Programming* 22.4 (2014), pp. 309–329.

[102] K. Gallagher, A. Hatch, and M. Munro. « A Framework for Software Architecture Visualisation Assessment. » In: *3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis, 2005. VISSOFT 2005*. 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis, 2005. VISSOFT 2005. 2005, pp. 1–6.

[103] Eleanor Jack Gibson and Anne D. Pick. *An Ecological Approach to Perceptual Learning and Development*. Oxford University Press, 2000. 254 pp.

[104] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. « Semi-Automatic Composition of Loop Transformations for Deep Parallelism and Memory Hierarchies. » In: *International Journal of Parallel Programming* 34.3 (July 21, 2006). URUK, pp. 261–317.

[105] H. H. Goldstine and Adele Goldstine. « The Electronic Numerical Integrator and Computer (ENIAC). » In: *Mathematical Tables and Other Aids to Computation* 2.15 (1946), pp. 97–110. JSTOR: 2002620.

[106] « Special Issue on VIsual Programming. » In: *IEEE Computer* 18.8 (Aug. 1985). Ed. by Robert B Grafton and Tadao Ichikawa, pp. 1–135.

[107] Thomas RG Green and Marian Petre. « When Visual Programs Are Harder to Read than Textual Programs. » In: *Human-Computer Interaction: Tasks and Organisation, Proceedings of ECCE-6 (6th European Conference on Cognitive Ergonomics). GC van Der Veer, MJ Tauber, S. Bagnarola and M. Antavolits. Rome, CUD*. Citeseer, 1992.

[108] Martin Griebl, Paul Feautrier, and Christian Lengauer. « Index Set Splitting. » In: *International Journal of Parallel Programming* 28.6 (2000), pp. 607–631.

[109] Martin Griebl and Christian Lengauer. « The Loop Parallelizer LooPo-Announcement. » In: *Proceedings of the 9th International Workshop on Languages and Compilers for Parallel Computing*. LCPC '96. London, UK, UK: Springer-Verlag, 1997, pp. 603–604.

[110] Paul A. Gross, Micah S. Herstand, Jordana W. Hodges, and Caitlin L. Kelleher. « A Code Reuse Interface for Non-Programmer Middle School Students. » In: *Proceedings of the 15th International Conference on Intelligent User Interfaces*. IUI '10. New York, NY, USA: ACM, 2010, pp. 219–228.

[111] Paul Gross and Caitlin Kelleher. « The Looking Glass IDE for Learning Computer Programming Through Storytelling and History Exploration: Conference Workshop. » In: *J. Comput. Sci. Coll.* 26.1 (Oct. 2010), pp. 75–76.

[112] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. « Polly — Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. » In: *Parallel Processing Letters* 22 (04 Dec. 1, 2012), p. 1250010.

[113] Tobias Grosser, Albert Cohen, Justin Holewinski, Ponuswamy Sadayappan, and Sven Verdoolaege. « Hybrid Hexagonal/Classical Tiling for GPUs. » In: *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. ACM, 2014, p. 66.

[114] Tobias Grosser, J. Ramanujam, Louis-Noel Pouchet, P. Sadayappan, and Sebastian Pop. « Optimistic Delinearization of Parametrically Sized Arrays. » In: *Proceedings of the 29th ACM on International Conference on Supercomputing*. ICS '15. New York, NY, USA: ACM, 2015, pp. 351–360.

[115] Tovi Grossman and Ravin Balakrishnan. « The Bubble Cursor: Enhancing Target Acquisition by Dynamic Resizing of the Cursor's Activation Area. » In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '05. New York, NY, USA: ACM, 2005, pp. 281–290.

[116] Yves Guiard, Renaud Blanch, and Michel Beaudouin-Lafon. « Object Pointing: A Complement to Bitmap Pointing in GUIs. » In: *Proceedings of Graphics Interface 2004*. GI '04. School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada: Canadian Human-Computer Communications Society, 2004, pp. 9–16.

[117] Diana Göhringer and Jan Tepelmann. « An Interactive Tool Based on Polly for Detection and Parallelization of Loops. » In: *Proceedings of Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*. PARMA-DITAM '14. New York, NY, USA: ACM, 2014, 1:1–1:6.

[118] L.M. Haibt. « A Program to Draw Multi-Level Flow Charts. » In: Western Joint Computer Conference. San Francisco, CA, USA, 1959, pp. 131–137.

[119] M. T. Heath and J. A. Etheridge. « Visualizing the Performance of Parallel Programs. » In: *IEEE Software* 8.5 (Sept. 1991), pp. 29–39.

[120] Jeffrey Heer, Michael Bostock, and Vadim Ogievetsky. « A Tour Through the Visualization Zoo. » In: *Commun. ACM* 53.6 (June 2010), pp. 59–67.

[121] Joachim Hohmann. *Der Plankalkül im Vergleich mit algorithmischen Sprachen*. Google-Books-ID: f_dLAAAACAAJ. Toeche-Mittler, 1979. 136 pp.

[122] M. Homer and J. Noble. « Combining Tiled and Textual Views of Code. » In: *2014 Second IEEE Working Conference on Software Visualization (VISSOFT)*. 2014 Second IEEE Working Conference on Software Visualization (VISSOFT). Sept. 2014, pp. 1–10.

[123] Christopher D. Hundhausen, Sean F. Farley, and Jonathan L. Brown. « Can Direct Manipulation Lower the Barriers to Computer Programming and Promote Transfer of Training?: An Experimental Study. » In: *ACM Trans. Comput.-Hum. Interact.* 16.3 (Sept. 2009), 13:1–13:40.

[124] Amy Hurst, Jennifer Mankoff, Anind K. Dey, and Scott E. Hudson. « Dirty Desktops: Using a Patina of Magnetic Mouse Dust to Make Common Interactor Targets Easier to Select. » In: *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology*. UIST '07. New York, NY, USA: ACM, 2007, pp. 183–186.

[125] Alfred Inselberg and Bernard Dimsdale. « Parallel Coordinates. » In: *Human-Machine Interactive Systems*. Ed. by Allen Klinger. Languages and Information Systems. Springer US, 1991, pp. 199–233.

[126] François Irigoin, Pierre Jouvelot, and Rémi Triolet. « Semantical Interprocedural Parallelization: An Overview of the PIPS Project. » In: *ACM International Conference on Supercomputing 25th Anniversary Volume*. New York, NY, USA: ACM, 2014, pp. 143–150.

[127] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[128] Ghita Jalal, Nolwenn Maudet, and Wendy E. Mackay. « Color Portraits: From Color Picking to Interacting with Color. » In: *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. CHI '15. New York, NY, USA: ACM, 2015, pp. 4207–4216.

[129] Nicklas Bo Jensen, Sven Karlsson, and Christian W. Probst. « Compiler Feedback Using Continuous Dynamic Compilation during Development. » In: *International Workshop on Dynamic Compilation Everywhere*. DCE. Vienna: ACM, Jan. 2014, A1–12.

[130] Dean F. Jerding, John T. Stasko, and Thomas Ball. « Visualizing Interactions in Program Executions. » In: *Proceedings of the 19th International Conference on Software Engineering*. ICSE '97. New York, NY, USA: ACM, 1997, pp. 360–370.

[131] Alexandra Jimborean, Philippe Clauss, Benoît Pradelle, Luis Mastrangelo, and Vincent Loechner. « Adapting the Polyhedral Model As a Framework for Efficient Speculative Parallelization. » In: *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '12. New York, NY, USA: ACM, 2012, pp. 295–296.

[132] James A. Jones, Mary Jean Harrold, and John Stasko. « Visualization of Test Information to Assist Fault Localization. » In: *Proceedings of the 24th International Conference on Software Engineering*. ICSE '02. New York, NY, USA: ACM, 2002, pp. 467–477.

[133] Neil D. Jones and Steven S. Muchnick. « Even Simple Programs Are Hard to Analyze. » In: *Proceedings of the 2Nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '75. New York, NY, USA: ACM, 1975, pp. 106–118.

[134] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. New York, NY, USA: John Wiley & Sons, Inc., 1996.

[135] Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. « The Organization of Computations for Uniform Recurrence Equations. » In: *J. ACM* 14.3 (July 1967), pp. 563–590.

[136] B. Karran, J. Trümper, and J. Döllner. « SYNCTRACE: Visual Thread-Interplay Analysis. » In: *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*. 2013 First IEEE Working Conference on Software Visualization (VISSOFT). Sept. 2013, pp. 1–10.

[137] Chantal Keller, Jérémy Bluteau, Renaud Blanch, and Sabine Coquillart. « PseudoWeight: Making Tabletop Interaction with Virtual Objects More Tangible. » In: *Proceedings of the 2012 ACM International Conference on Interactive Tabletops and Surfaces*. ITS '12. New York, NY, USA: ACM, 2012, pp. 201–204.

[138] Wayne Kelly and William Pugh. « A Framework for Unifying Reordering Transformations. » In: (Oct. 15, 1998).

[139] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and David Wonnacott. *The Omega Library Interface Guide*. College Park, MD, USA: University of Maryland at College Park, 1995.

[140] Minjang Kim, Hyesoon Kim, and Chi-Keung Luk. « Prospector: A Dynamic Data-Dependence Profiler to Help Parallel Programming. » In: *HotPar'10: Proceedings of the USENIX Workshop on Hot Topics in Parallelism*. 2010.

[141] Laszlo B Kish. « End of Moore's Law: Thermal (Noise) Death of Integration in Micro and Nano Electronics. » In: *Physics Letters A* 305 (3–4 Dec. 2, 2002), pp. 144–149.

[142] Clemens N. Klokmose, James R. Eagan, Siemen Baader, Wendy Mackay, and Michel Beaudouin-Lafon. « Webstrates: Shareable Dynamic Media. » In: *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. UIST '15. New York, NY, USA: ACM, 2015, pp. 280–290.

[143] Donald E. Knuth and Luis Trabb Pardo. « The Early Development of Programming Languages. » In: *Encyclopedia of Computer Science and Technology*. Ed. by J. Belzer, A.G. Holzman, and A. Kent. Vol. 6. New York: Dekker, 1977, pp. 419–493.

[144] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S. Müller, and Wolfgang E. Nagel. « The Vampir Performance Analysis Tool-Set. » In: *Tools for High Performance Computing*. Ed. by Michael Resch, Rainer Keller, Valentin Himmler, Bettina Krammer, and Alexander Schulz. Springer Berlin Heidelberg, 2008, pp. 139–155.

[145] Andrew J. Ko et al. « The State of the Art in End-User Software Engineering. » In: *ACM Comput. Surv.* 43.3 (Apr. 2011), 21:1–21:44.

[146] Bogdan Korel and Janusz Laski. « Dynamic Program Slicing. » In: *Information Processing Letters* 29.3 (Oct. 26, 1988), pp. 155–163.

[147] Rainer Koschke. « Software Visualization in Software Maintenance, Reverse Engineering, and Re-Engineering: A Research Survey. » In: *Journal of Software Maintenance* 15.2 (Mar. 2003), pp. 87–109.

[148] E. Kraemer and J. T. Stasko. « The Visualization of Parallel Systems: An Overview. » In: *Journal of Parallel and Distributed Computing* 18.2 (June 1993), pp. 105–117.

[149] J. Krinke. « Visualization of Program Dependence and Slices. » In: *20th IEEE International Conference on Software Maintenance, 2004. Proceedings*. 20th IEEE International Conference on Software Maintenance, 2004. Proceedings. Sept. 2004, pp. 168–177.

[150] G. Kurtenbach, T. P. Moran, and W. Buxton. « Contextual Animation of Gestural Commands. » In: *Computer Graphics Forum* 13.5 (Dec. 1, 1994), pp. 305–314.

[151] Leslie Lamport. « The Parallel Execution of DO Loops. » In: *Commun. ACM* 17.2 (Feb. 1974), pp. 83–93.

[152] C. F. J. Lange, M. R. V. Chaudron, and J. Muskens. « In Practice: UML Software Architecture and Design Description. » In: *IEEE Software* 23.2 (Mar. 2006), pp. 40–46.

[153] Edward Lank, Yi-Chun Nikko Cheng, and Jaime Ruiz. « Endpoint Prediction Using Motion Kinematics. » In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '07. New York, NY, USA: ACM, 2007, pp. 637–646.

[154] M. Lanza and S. Ducasse. « Polymetric Views - a Lightweight Visual Approach to Reverse Engineering. » In: *IEEE Transactions on Software Engineering* 29.9 (Sept. 2003), pp. 782–795.

[155] M. Lanza, S. Ducasse, H. Gall, and M. Pinzger. « CodeCrawler - an Information Visualization Tool for Program Comprehension. » In: *27th International Conference on Software Engineering, 2005. ICSE 2005. Proceedings*. 27th International Conference on Software Engineering, 2005. ICSE 2005. Proceedings. May 2005, pp. 672–673.

[156] Michele Lanza and Stéphane Ducasse. « Understanding Software Evolution Using a Combination of Software Visualization and Software Metrics. » In: *Proceedings of LMO 2002 (Langages et Mod{\'e}les {\'a} Objets)*. Langages et Mod{\'e}les {\'a} Objets. Aug. 2002, pp. 135–149.

[157] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer Science & Business Media, May 16, 2007. 213 pp.

[158] P. Larsen, R. Ladelsky, J. Lidman, S. A. McKee, S. Karlsson, and A. Zaks. « Parallelizing More Loops with Compiler Guided Refactoring. » In: *2012 41st International Conference on Parallel Processing*. 2012 41st International Conference on Parallel Processing. Sept. 2012, pp. 410–419.

[159] James Larus. « Spending Moore's Dividend. » In: *Commun. ACM* 52.5 (May 2009), pp. 62–69.

[160] Chris Lattner. « LLVM and Clang: Next Generation Compiler Technology. » In: *The BSD Conference*. The BSD Conference. 2008, pp. 1–2.

[161] Hervé Le Verge. *A Note on Cherniakova's Algorithm*. Research Report 1662. INRIA, 1992.

[162] Hervé Le Verge. « Recurrences on Lattice Polyhedra and their Applications. » Unpublished work based on a manuscript written by H. Le Verge just before his untimely death in 1994. IRISA, Apr. 1995.

[163] A. Lecuyer, S. Coquillart, A. Kheddar, P. Richard, and P. Coiffet. « Pseudo-Haptic Feedback: Can Isometric Input Devices Simulate Force Feedback? » In: *IEEE Virtual Reality, 2000. Proceedings*. IEEE Virtual Reality, 2000. Proceedings. 2000, pp. 83–90.

[164] A. Lecuyer, J. M. Burkhardt, S. Coquillart, and P. Coiffet. « "Boundary of Illusion": An Experiment of Sensory Integration with a Pseudo-Haptic System. » In: *IEEE Virtual Reality, 2001. Proceedings*. IEEE Virtual Reality, 2001. Proceedings. Mar. 2001, pp. 115–122.

[165] Christian Lengauer. « Loop Parallelization in the Polytope Model. » In: *CONCUR'93*. Ed. by Eike Best. Lecture Notes in Computer Science 715. Springer Berlin Heidelberg, Aug. 23, 1993, pp. 398–416.

[166] Amy W. Lim and Monica S. Lam. « Maximizing Parallelism and Minimizing Synchronization with Affine Transforms. » In: *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '97. New York, NY, USA: ACM, 1997, pp. 201–214.

[167] Vincent Loechner. « PolyLib: A Library for Manipulating Parameterized Polyhedra. » Strasbourg, Mar. 17, 1999.

[168] Mark Lundstrom. « Moore's Law Forever? » In: *Science* 299.5604 (Jan. 10, 2003), pp. 210–211. pmid: 12522237.

[169] Anatole Lécuyer. « Simulating Haptic Feedback Using Vision: A Survey of Research and Applications of Pseudo-Haptic Feedback. » In: *Presence: Teleoperators and Virtual Environments* 18.1 (Jan. 27, 2009), pp. 39–53.

[170] Anatole Lécuyer, Jean-Marie Burkhardt, and Laurent Etienne. « Feeling Bumps and Holes Without a Haptic Interface: The Perception of Pseudo-Haptic Textures. » In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '04. New York, NY, USA: ACM, 2004, pp. 239–246.

[171] I. Scott MacKenzie. « Virtual Environments and Advanced Interface Design. » In: ed. by Woodrow Barfield and Thomas A. Furness III. New York, NY, USA: Oxford University Press, Inc., 1995, pp. 437–470.

[172] C. A. Mack. « Fifty Years of Moore's Law. » In: *IEEE Transactions on Semiconductor Manufacturing* 24.2 (May 2011), pp. 202–207.

[173] Wendy E. Mackay. « Triggers and Barriers to Customizing Software. » In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '91. New York, NY, USA: ACM, 1991, pp. 153–160.

[174] Jock Mackinlay. « Automating the Design of Graphical Presentations of Relational Information. » In: *ACM Trans. Graph.* 5.2 (Apr. 1986), pp. 110–141.

[175] J. I. Maletic, A. Marcus, and M. L. Collard. « A Task Oriented View of Software Visualization. » In: *First International Workshop on Visualizing Software for Understanding and Analysis, 2002. Proceedings*. First International Workshop on Visualizing Software for Understanding and Analysis, 2002. Proceedings. 2002, pp. 32–40.

[176] Jonathan I. Maletic, Andrian Marcus, and Louis Feng. « Source Viewer 3D (Sv3D): A Framework for Software Visualization. » In: *Proceedings of the 25th International Conference on Software Engineering*. ICSE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 812–813.

[177] Allen D. Malony, David H. Hammerslag, and David J. Jablonowski. « Traceview: A Trace Visualization Tool. » In: *IEEE Software* 8.5 (1991), pp. 19–28.

[178] Regan L. Mandryk and Carl Gutwin. « Perceptibility and Utility of Sticky Targets. » In: *Proceedings of Graphics Interface 2008*. GI '08. Toronto, Ont., Canada, Canada: Canadian Information Processing Society, 2008, pp. 65–72.

[179] Regan L. Mandryk, Malcolm E. Rodgers, and Kori M. Inkpen. « Sticky Widgets: Pseudo-Haptic Widget Enhancements for Multi-Monitor Displays. » In: *CHI '05 Extended Abstracts on Human Factors in Computing Systems*. CHI EA '05. New York, NY, USA: ACM, 2005, pp. 1621–1624.

[180] Patricia Yancey Martin and Barry A. Turner. « Grounded Theory and Organizational Research. » In: *The Journal of Applied Behavioral Science* 22.2 (Jan. 4, 1986), pp. 141–157.

[181] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. « Improving Data Locality with Loop Transformations. » In: *ACM Trans. Program. Lang. Syst.* 18.4 (July 1996), pp. 424–453.

[182] A. McNair, D. M. German, and J. Weber-Jahnke. « Visualizing Software Architecture Evolution Using Change-Sets. » In: *14th Working Conference on Reverse Engineering, 2007. WCRE 2007*. 14th Working Conference on Reverse Engineering, 2007. WCRE 2007. Oct. 2007, pp. 130–139.

[183] Katharina Mehner. « JaVis: A UML-Based Visualization and Debugging Environment for Concurrent Java Programs. » In: *Software Visualization*. Ed. by Stephan Diehl. Lecture Notes in Computer Science 2269. Springer Berlin Heidelberg, 2002, pp. 163–175.

[184] T. Mens and T. Tourwe. « A Survey of Software Refactoring. » In: *IEEE Transactions on Software Engineering* 30.2 (Feb. 2004), pp. 126–139.

[185] Koert van Mensvoort, Dik J. Hermes, and Maurice van Montfort. « Usability of Optically Simulated Haptic Feedback. » In: *International Journal of Human-Computer Studies* 66.6 (June 2008), pp. 438–451.

[186] Rym Mili and Renee Steiner. « Software Engineering. » In: *Software Visualization (International Seminar)*. Ed. by Stephan Diehl. Lecture Notes in Computer Science 2269. Dagstuhl Castle, Germany: Springer, 2002, pp. 129–137.

[187] Thomas Moher, David Mak, Brad Blumenthal, and Laura Leventhal. « Comparing the Comprehensibility of Textual and Graphic Programs: The Case of Petri Nets. » In: *Empirical Studies of Programmers: Fifth Workshop : Papers Presented at the Fifth Workshop on Empirical Studies of Programmers, December 3-5, 1993, Palo Alto, CA*. Ed. by Curtis R. Cook, Jean C. Scholtz, and James C. Spohrer. Intellect Books, 1993, pp. 137–161.

[188] Andrew F. Monk, Paul Walsh, and Alan J. Dix. « A Comparison of Hypertext, Scrolling and Folding As Mechanisms for Program Browsing. » In: *Proceedings of the Fourth Conference of the British Computer Society on People and Computers IV*. New York, NY, USA: Cambridge University Press, 1988, pp. 421–435.

[189] Gordon E. Moore. « Cramming More Components onto Integrated Circuits. » In: *Electronics* (Apr. 19, 1965), pp. 114–117.

[190] B. A. Myers. « Visual Programming, Programming by Example, and Program Visualization: A Taxonomy. » In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '86. New York, NY, USA: ACM, 1986, pp. 59–66.

[191] Brad A. Myers. « Taxonomies of Visual Programming and Program Visualization. » In: *Journal of Visual Languages & Computing* 1.1 (Mar. 1990), pp. 97–123.

[192] Ralph Müller-Pfefferkorn, Wolfgang E. Nagel, and Bernd Trenkler. « Optimizing Cache Access: A Tool for Source-to-Source Transformations and Real-Life Compiler Tests. » In: *Euro-Par 2004 Parallel Processing*. Ed. by Marco Danelutto, Marco Vanneschi, and Domenico Laforenza. Lecture Notes in Computer Science 3149. Springer Berlin Heidelberg, Aug. 31, 2004, pp. 72–81.

[193] Kumiyo Nakakoji, Yasuhiro Yamamoto, and Yasuharu Koike. « Toward Principles for Visual Interaction Design for Communicating Weight by Using Pseudo-Haptic Feedback. » In: *Proceedings of the 2010 International Conference on The Interaction*

Design. Create'10. Swinton, UK, UK: British Computer Society, 2010, pp. 34–39.

[194] I. Nassi and B. Shneiderman. « Flowchart Techniques for Structured Programming. » In: *SIGPLAN Not.* 8.8 (Aug. 1973), pp. 12–26.

[195] Raquel Navarro-Prieto and Jose J. Canas. « Are Visual Programming Languages Better? The Role of Imagery in Program Comprehension. » In: *International Journal of Human-Computer Studies* 54.6 (June 1, 2001), pp. 799–829.

[196] Renato Lima Novais, André Torres, Thiago Souto Mendes, Manoel Mendonça, and Nico Zazworka. « Software Evolution Visualization: A Systematic Mapping Study. » In: *Information and Software Technology* 55.11 (Nov. 2013), pp. 1860–1883.

[197] Rainer Oechsle and Thomas Schmitt. « JAVAVIS: Automatic Program Visualization with Object and Sequence Diagrams Using the Java Debug Interface (JDI). » In: *Software Visualization*. Ed. by Stephan Diehl. Lecture Notes in Computer Science 2269. Springer Berlin Heidelberg, 2002, pp. 176–190.

[198] Seymour Papert. *Mindstorms: Children, Computers, and Powerful Ideas*. New York, NY, USA: Basic Books, Inc., 1980.

[199] D. Patterson. « The Trouble with Multi-Core. » In: *IEEE Spectrum* 47.7 (July 2010), pp. 28–32, 53.

[200] David A. Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier, Oct. 7, 2011. 857 pp.

[201] Wim De Pauw, Erik Jensen, Nick Mitchell, Gary Sevitsky, John Vlissides, and Jeaha Yang. « Visualizing the Execution of Java Programs. » In: *Software Visualization*. Ed. by Stephan Diehl. Lecture Notes in Computer Science 2269. Springer Berlin Heidelberg, 2002, pp. 151–162.

[202] Marian Petre. « Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming. » In: *Communications of the ACM* 38.6 (June 1995), pp. 33–44.

[203] Matthew D. Plumlee and Colin Ware. « Zooming Versus Multiple Window Interfaces: Cognitive Costs of Visual Comparisons. » In: *ACM Trans. Comput.-Hum. Interact.* 13.2 (June 2006), pp. 179–209.

[204] Sebastian Pop, Albert Cohen, Cédric Bastoul, Sylvain Girbal, Georges-André Silber, and Nicolas Vasilache. « GRAPHITE: Polyhedral Analyses and Optimizations for GCC. » In: *Proceedings of the 2006 GCC Developers Summit*. GCC Developers Summit. 2006, pp. 179–197.

[205] L. N. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache. « Iterative Optimization in the Polyhedral Model: Part I, One-Dimensional Time. » In: *International Symposium on Code Generation and Optimization (CGO'07)*. International Symposium on Code Generation and Optimization (CGO'07). Mar. 2007, pp. 144–156.

[206] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and John Cavazos. « Iterative Optimization in the Polyhedral Model: Part Ii, Multidimensional Time. » In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '08. New York, NY, USA: ACM, 2008, pp. 90–100.

[207] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, P. Sadayappan, and Nicolas Vasilache. « Loop Transformations: Convexity, Pruning and Optimization. » In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '11. New York, NY, USA: ACM, 2011, pp. 549–562.

[208] B. A. Price, I. S. Small, and R. M. Baecker. « A Taxonomy of Software Visualization. » In: *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences, 1992*. Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences, 1992. Vol. ii. Jan. 1992, 597–606 vol.2.

[209] Blaine A. Price, Ronald M. Baecker, and Ian S. Small. « A Principled Taxonomy of Software Visualization. » In: *Journal of Visual Languages & Computing* 4.3 (Sept. 1993), pp. 211–266.

[210] Tony Printezis and Richard Jones. « GCspy: An Adaptable Heap Visualisation Framework. » In: *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA '02. New York, NY, USA: ACM, 2002, pp. 343–358.

[211] William Pugh. « The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis. » In: *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*. Supercomputing '91. New York, NY, USA: ACM, 1991, pp. 4–13.

[212] Andreas Pusch and Anatole Lécuyer. « Pseudo-Haptics: From the Theoretical Foundations to Practical System Design Guidelines. » In: *Proceedings of the 13th International Conference on Multimodal Interfaces*. ICMI '11. New York, NY, USA: ACM, 2011, pp. 57–64.

[213] Zenon Pylyshyn. « Some Puzzling Findings in Multiple Object Tracking: I. Tracking without Keeping Track of Object Identities. » In: *Visual Cognition* 11.7 (Oct. 1, 2004), pp. 801–822.

[214] Fabien Quilleré, Sanjay Rajopadhye, and Doran Wilde. « Generation of Efficient Nested Loops from Polyhedra. » In: *International Journal of Parallel Programming* 28.5 (Oct. 2000), pp. 469–498.

[215] Patrice Quinton. « The Systematic Design of Systolic Arrays. » In: *Automata Networks in Computer Science*. Ed. by F Soulié, Y Robert, and M Tchuenté. Manchester, UK: Manchester University Press, 1987, pp. 229–260.

[216] Mitchel Resnick et al. « Scratch: Programming for All. » In: *Commun. ACM* 52.11 (Nov. 2009), pp. 60–67.

[217] J. C. Roberts. « State of the Art: Coordinated Multiple Views in Exploratory Visualization. » In: *Fifth International Conference on Coordinated and Multiple Views in Exploratory Visualization, 2007. CMV '07*. Fifth International Conference on Coordinated and Multiple Views in Exploratory Visualization, 2007. CMV '07. July 2007, pp. 61–71.

[218] Judy Robertson and Maurits Kaptein, eds. *Modern Statistical Methods for HCI*. Human–Computer Interaction Series. Springer International Publishing, 2016. 350 pp.

[219] G. C. Roman and K. C. Cox. « A Taxonomy of Program Visualization Systems. » In: *Computer* 26.12 (Dec. 1993), pp. 11–24.

[220] Gruia-Catalin Roman and Kenneth C. Cox. « Program Visualization: The Art of Mapping Programs to Pictures. » In: *Proceedings of the 14th International Conference on Software Engineering*. ICSE '92. New York, NY, USA: ACM, 1992, pp. 412–420.

[221] Gabe Rudy, Malik Murtaza Khan, Mary Hall, Chun Chen, and Jacqueline Chame. « A Programming Language Interface to Describe Transformations and Code Generation. » In: *Languages and Compilers for Parallel Computing*. Ed. by Keith Cooper, John Mellor-Crummey, and Vivek Sarkar. Lecture Notes in Computer Science 6548. Springer Berlin Heidelberg, Oct. 7, 2010, pp. 136–150.

[222] Michael L. Scott. *Programming Language Pragmatics, Fourth Edition*. 4 edition. Morgan Kaufmann, Dec. 25, 2015. 992 pp.

[223] Clifford A. Shaffer, Matthew Cooper, and Stephen H. Edwards. « Algorithm Visualization: A Report on the State of the Field. » In: *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education*. SIGCSE '07. New York, NY, USA: ACM, 2007, pp. 150–154.

[224] Clifford A. Shaffer, Matthew L. Cooper, Alexander Joel D. Alon, Monika Akbar, Michael Stewart, Sean Ponce, and Stephen H. Edwards. « Algorithm Visualization: The State of the Field. » In: *Trans. Comput. Educ.* 10.3 (Aug. 2010), 9:1–9:22.

[225] J. Shirako, L. N. Pouchet, and V. Sarkar. « Oil and Water Can Mix: An Integration of Polyhedral and AST-Based Transformations. » In: *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*. SC14: International Conference for High Performance Computing, Networking, Storage and Analysis. Nov. 2014, pp. 287–298.

[226] B. Shneiderman. « The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations. » In: *, IEEE Symposium on Visual Languages, 1996. Proceedings*. , IEEE Symposium on Visual Languages, 1996. Proceedings. Sept. 1996, pp. 336–343.

[227] Ben Shneiderman. « Direct Manipulation: A Step Beyond Programming Languages. » In: *Proceedings of the Joint Conference on Easier and More Productive Use of Computer Systems. (Part - II): Human Interface and the User Interface - Volume 1981*. CHI '81. New York, NY, USA: ACM, 1981, pp. 143–.

[228] Ben Shneiderman. « The Future of Interactive Systems and the Emergence of Direct Manipulation. » In: *Behaviour & Information Technology* 1.3 (July 1, 1982), pp. 237–256.

[229] David Canfield Smith. *PYGMALION: A Creative Programming Environment*. Technical Rept. ADA016811. Stanford University, June 1975, p. 199.

[230] *Sorting Out Sorting (Video)*. University of Toronto, 1981.

[231] Ryan Stansifer. *Presburger's Article on Integer Airthmetic: Remarks and Translation*. Technical Report TR84-639. Cornell University, Computer Science Departement, Sept. 1984, p. 20.

[232] J. T. Stasko. « Tango: A Framework and System for Algorithm Animation. » In: *Computer* 23.9 (Sept. 1990), pp. 27–39.

[233] J. T. Stasko and E. Kraemer. « A Methodology for Building Application-Specific Visualizations of Parallel Programs. » In: *Journal of Parallel and Distributed Computing* 18.2 (June 1993), pp. 258–264.

[234] J. T. Stasko and C. Patterson. « Understanding and Characterizing Software Visualization Systems. » In: *, 1992 IEEE Workshop on Visual Languages, 1992. Proceedings*. , 1992 IEEE Workshop on Visual Languages, 1992. Proceedings. Sept. 1992, pp. 3–10.

[235] John T. Stasko and Qiang Alex Zhao. *Visualizing the Execution of Threads-Based Parallel Programs*. 1995.

[236] John Stasko. *Software Visualization: Programming as a Multimedia Experience*. MIT Press, 1998. 596 pp.

[237] M. A. Storey, C. Best, and J. Michand. « SHriMP Views: An Interactive Environment for Exploring Java Programs. » In: *9th International Workshop on Program Comprehension, 2001. IWPC 2001. Proceedings*. 9th International Workshop on Program Comprehension, 2001. IWPC 2001. Proceedings. 2001, pp. 111–112.

[238] Ivan E. Sutherland. « Sketch Pad a Man-Machine Graphical Communication System. » In: *Proceedings of the SHARE Design Automation Workshop*. DAC '64. New York, NY, USA: ACM, 1964, pp. 6.329–6.346.

[239] Herb Sutter and James Larus. « Software and the Concurrency Revolution. » In: *Queue* 3.7 (Sept. 2005), pp. 54–62.

[240] T. Systa, Ping Yu, and H. Muller. « Analyzing Java Software by Combining Metrics and Program Visualization. » In: *Software Maintenance and Reengineering, 2000. Proceedings of the Fourth European*. Software Maintenance and Reengineering, 2000. Proceedings of the Fourth European. Feb. 2000, pp. 199–208.

[241] Jie Tao, Thomas Dressler, and Wolfgang Karl. « An Interactive Graphical Environment for Code Optimization. » In: *Computational Science–ICCS 2007*. Springer, 2007, pp. 831–838.

[242] Alexandru Telea and David Auber. « Code Flows: Visualizing Structural Evolution of Source Code. » In: *Computer Graphics Forum* 27.3 (May 1, 2008), pp. 831–838.

[243] M. Termeer, C. F. J. Lange, A. Telea, and M. R. V. Chaudron. « Visual Exploration of Combined Architectural and Metric Information. » In: *3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis, 2005. VISSOFT 2005*. 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis, 2005. VISSOFT 2005. 2005, pp. 1–6.

[244] Larry Tesler. « A Personal History of Modeless Text Editing and Cut/Copy-Paste. » In: *interactions* 19.4 (July 2012), pp. 70–75.

[245] A.R. Teyseyre and M.R. Campo. « An Overview of 3D Software Visualization. » In: *IEEE Transactions on Visualization and Computer Graphics* 15.1 (Jan. 2009), pp. 87–105.

[246] S. R. Tilley and H. A. Muller. « Using Virtual Subsystems in Project Management. » In: *, Proceeding of the Sixth International Workshop on Computer-Aided Software Engineering, 1993. CASE '93.* , Proceeding of the Sixth International Workshop on Computer-Aided Software Engineering, 1993. CASE '93. July 1993, pp. 144–153.

[247] Ramakrishna Upadrasta. « Sub-Polyhedral Compilation Using (Unit-)Two-Variables-Per-Inequality Polyhedra. » PhD thesis. Université Paris Sud - Paris XI, Mar. 13, 2013.

[248] Hans Vandierendonck, Sean Rul, and Koen De Bosschere. « The Paralax Infrastructure: Automatic Parallelization with a Helping Hand. » In: *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*. PACT '10. New York, NY, USA: ACM, 2010, pp. 389–400.

[249] Nicolas Vasilache, Albert Cohen, and Louis-Noel Pouchet. « Automatic Correction of Loop Transformations. » In: *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*. PACT '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 292–304.

[250] Nicolas Vasilache, Cedric Bastoul, Albert Cohen, and Sylvain Girbal. « Violated Dependence Analysis. » In: *Proceedings of the 20th Annual International Conference on Supercomputing*. ICS '06. New York, NY, USA: ACM, 2006, pp. 335–344.

[251] Steven R. Vegdahl. « Using Visualization Tools to Teach Compiler Design. » In: *Proceedings of the Fourteenth Annual Consortium on Small Colleges Southeastern Conference*. CCSC '00. USA: Consortium for Computing Sciences in Colleges, 2000, pp. 72–83.

[252] Anand Venkat, Manu Shantharam, Mary Hall, and Michelle Mills Strout. « Non-Affine Extensions to Polyhedral Code Generation. » In: *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. ACM, 2014, p. 185.

[253] Sven Verdoolaege. « Isl: An Integer Set Library for the Polyhedral Model. » In: *Mathematical Software – ICMS 2010*. Ed. by Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama. Lecture Notes in Computer Science 6327. Springer Berlin Heidelberg, Sept. 13, 2010, pp. 299–302.

[254] Sven Verdoolaege and Tobias Grosser. « Polyhedral Extraction Tool. » In: *Impact 2012*. Second International Workshop on Polyhedral Compilation Techniques, in conjuction with HiPEAC 2012. Paris, France, Jan. 23, 2012.

[255] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. « Polyhedral Parallel Code Generation for CUDA. » In: *ACM Trans. Archit. Code Optim.* 9.4 (Jan. 2013), 54:1–54:23.

[256] Sven Verdoolaege, Serge Guelton, Tobias Grosser, and Albert Cohen. « Schedule Trees. » In: IMPACT - 4th Workshop on Polyhedral Compilation Techniques, associated with HiPEAC. ACM, Jan. 20, 2014.

[257] R. L. Veroy, N. P. Ricci, and S. Z. Guyer. « Visualizing the Allocation and Death of Objects. » In: *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*. 2013 First IEEE

Working Conference on Software Visualization (VISSOFT). Sept. 2013, pp. 1–4.

[258] Stefan-Lucian Voinea. « Software Evolution Visualization. » PhD Thesis. Eindhoven: Technische Universiteit Eindhoven, Oct. 1, 2007. 181 pp.

[259] Colin Ware. *Information Visualization: Perception for Design*. 3rd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012.

[260] Mark Weiser. « Program Slicing. » In: *Proceedings of the 5th International Conference on Software Engineering*. ICSE '81. Piscataway, NJ, USA: IEEE Press, 1981, pp. 439–449.

[261] Mark Weiser. « Some Computer Science Issues in Ubiquitous Computing. » In: *Commun. ACM* 36.7 (July 1993), pp. 75–84.

[262] R. Wettel and M. Lanza. « Visualizing Software Systems as Cities. » In: *4th IEEE International Workshop on Visualizing Software for Understanding and Analysis, 2007. VISSOFT 2007*. 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis, 2007. VISSOFT 2007. June 2007, pp. 92–99.

[263] K. N. Whitley. « Visual Programming Languages and the Empirical Evidence For and Against. » In: *Journal of Visual Languages & Computing* 8.1 (Feb. 1, 1997), pp. 109–142.

[264] J. A. Wise, J. J. Thomas, K. Pennock, D. Lantrip, M. Pottier, A. Schur, and V. Crow. « Visualizing the Non-Visual: Spatial Analysis and Interaction with Information from Text Documents. » In: *Information Visualization, 1995. Proceedings.* Information Visualization, 1995. Proceedings. Oct. 1995, pp. 51–58.

[265] Michael Joseph Wolfe. *High Performance Compilers for Parallel Computing*. Ed. by Carter Shanklin and Leda Ortega. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[266] Yi Wen Wong, Tomasz Dubrownik, Wai Teng Tang, Wen Jun Tan, Rubing Duan, Rick Siow Mong Goh, Shyh-hao Kuo, Stephen John Turner, and Weng-Fai Wong. « Tulipse: A Visualization Framework for User-Guided Parallelization. » In: *Euro-Par 2012 Parallel Processing*. Springer, 2012, pp. 4–15.

[267] Jingwei Wu and Margaret-Anne D. Storey. « A Multi-Perspective Software Visualization Environment. » In: *Proceedings of the 2000 Conference of the Centre for Advanced Studies on Collaborative Research*. CASCON '00. Mississauga, Ontario, Canada: IBM Press, 2000, pp. 15–.

[268] X. Xie, D. Poshyvanyk, and A. Marcus. « Visualization of CVS Repository Information. » In: *13th Working Conference on Reverse Engineering, 2006. WCRE '06*. 13th Working Conference on Reverse Engineering, 2006. WCRE '06. Oct. 2006, pp. 231–242.

[269] Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan. « POET: Parameterized Optimizations for Empirical Tuning. » In: *2007 IEEE International Parallel and Distributed Processing Symposium*. 2007 IEEE International Parallel and Distributed Processing Symposium. Mar. 2007, pp. 1–8.

[270] Yijun Yu and Erik H. D'Hollander. « Loop Parallelization Using the 3D Iteration Space Visualizer. » In: *Journal of Visual Languages & Computing* 12.2 (Apr. 2001), pp. 163–181.

[271] Tomofumi Yuki, Gautam Gupta, DaeGon Kim, Tanveer Pathan, and Sanjay Rajopadhye. « Alphaz: A System for Design Space Exploration in the Polyhedral Model. » In: *Languages and Compilers for Parallel Computing*. Springer, 2012, pp. 17–31.

[272] Kang Zhang, Tom Hintz, and Xianwu Ma. « The Role of Graphics in Parallel Program Development. » In: *Journal of Visual Languages & Computing* 10.3 (June 1999), pp. 215–243.

[273] Thomas Zimmermann and Andreas Zeller. « Visualizing Memory Graphs. » In: *Software Visualization*. Ed. by Stephan Diehl. Lecture Notes in Computer Science 2269. Springer Berlin Heidelberg, 2002, pp. 191–204.

[274] Oleksandr Zinenko, Cédric Bastoul, and Stéphane Huot. « Manipulating Visualization, Not Codes. » In: International Workshop on Polyhedral Compilation Techniques (IMPACT). Jan. 19, 2015, p. 8.

[275] Oleksandr Zinenko, Stéphane Huot, and Cédric Bastoul. « Clint: A Direct Manipulation Tool for Parallelizing Compute-Intensive Program Parts. » In: *Visual Languages and Human-Centric Computing (VL/HCC), 2014 IEEE Symposium on*. IEEE, 2014, pp. 109–112.

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)
université PARIS-SACLAY

Titre : Restructuration Interactive des Programmes
Mots-clés : visualisation des logiciels, optimisation des programmes, techniques d'interaction, modèle polyédrique

Le développement des logiciels et leur restructuration deviennent de plus en plus complexes à cause de l'adoption massive des architectures parallèles, ce qui nécessite une expertise considérable de la part des développeurs. Bien que des nombreux modèles et langages de programmation permettent de créer des programmes efficaces, ils n'offrent pas de support spécifique à la restructuration des programmes existants afin d'en augmenter l'efficacité. En même temps, les approches automatiques sont trop conservatives et insuffisamment précises pour atteindre une partie substantielle de la performance du système sans que le développeur aie à fournir des information sémantiques supplémentaires. Pour répondre à ces défis, nous adoptons l'approche de la restructuration interactive des programmes qui lie la manipulation semi-automatique des programmes avec la visualisation des logiciels. Dans cette thèse, l'approche de restructuration interactive est illustrée par l'extension du modèle polyédrique — une représentation des programmes moderne et puissante — pour permettre la manipulation de haut niveau ainsi que par la conception et l'évaluation d'une interface visuelle à manipulation directe pour la restructuration des programmes. Cette interface visualise l'information qui n'était pas immédiatement accessible dans la représentation textuelle et permet de manipuler des programmes sans en réécrire le code. Nous proposons également une représentation de l'optimisation de programme, calculée automatiquement, telle que le développeur puisse la comprendre et réutiliser facilement ainsi que la modifier d'une manière textuelle ou visuelle dans le cadre du partenariat homme-machine. Afin de représenter plusieurs aspects de la restructuration des programmes, nous concevons et évaluons une nouvelle interaction qui permet de communiquer l'information supplémentaire et non-cruciale pour la tâche à accomplir. Après une étude empirique de la distribution d'attention des développeurs face aux représentations textuelles et visuelles des programmes, nous discutons des implications pour la conception des outils d'aide à la programmation dans le cadre du modèle d'interaction instrumentale. La restructuration interactive des programmes est supposée faciliter la manipulation des programmes dans le but d'optimisation, la rendre plus efficace et plus largement adoptée.

Title: Interactive Program Restructuring
Keywords: software visualization, program optimization, interaction techniques, polyhedral model

Software development and program manipulation become increasingly complex with the massive adoption of parallel architectures, requiring significant expertise from developers. While numerous programming models and languages allow for creating efficient programs, they fall short at helping developers to restructure existing programs for more effective execution. At the same time, automatic approaches are overly conservative and imprecise to achieve a decent portion of the systems' performance without supplementary semantic information from the developer. To answer these challenges, we propose the interactive program restructuring approach, a bridge between semi-automatic program manipulation and software visualization. It is illustrated in this thesis by, first, extending a state-of-the-art polyhedral model for program representation so that it supports high-level program manipulation and, second, by designing and evaluating a direct manipulation visual interface for program restructuring. This interface provides information about the program that was not immediately accessible in the code and allows to manipulate programs without rewriting. We also propose a representation of an automatically computed program optimization in an understandable form, easily modifiable and reusable by the developer both visually and textually in a sort of human-machine partnership. To support different aspects of program restructuring, we design and evaluate a new interaction to communicate supplementary information, not critical for the task at hand. After an empirical study of developers' attention distribution when faced with visual and textual program representation, we discuss the implications for design of program manipulation tools in the instrumental interaction paradigm. We expect interactive program restructuring to make program manipulation for optimization more efficient and widely adopted.