Speculative rewriting of recursive programs as loop candidates for efficient parallelization and optimization using and inspector-executor mechanism

Salwa Kobeissi

Université de Strasbourg

École Doctorale Mathématiques, Sciences de l'Information et de l'Ingénieur (MSII)

Laboratoire des Sciences de l'Ingénieur, de l'Informatique et de l'Imagerie (ICube)

THÈSE présentée par :

Salwa KOBEISSI

Soutenue le : **24/06/2021**

Pour obtenir le grade de : **Docteur de l'Université de Strasbourg**

Discipline / Spécialité : **Informatique**

Speculative Rewriting of Recursive Programs as Loop Candidates for Efficient Parallelization and Optimization Using an Inspector-Executor Mechanism

THÈSE dirigée par :

Philippe CLAUSS Professeur, Université de Strasbourg, France

RAPPORTEURS :

Christophe ALIAS Chargé de Recherche, INRIA, École Normale Supérieure de Lyon, France

Denis BARTHOU Professeur, Institut National Polytechnique de Bordeaux, France

---

EXAMINATEURS :

Alexandra JIMBOREAN Chargée de Recherche, Université de Murcie, Espagne

Pierre-Etienne MOREAU Professeur, Mines Nancy, Université de Lorraine, France

To my marvellous mother,
who brightens my world with her pure heart and beautiful spirit,
whose brilliant eyes inspire my dreams and aspiration...

To my great father,
whose heart of gold is my treasure,
whose hands are engraved with lessons and powerful stories of perseverance and determination...

To my dear brothers,
who always have my back,
whose success stories boost my motivation...

To my love, my soulmate,
my home, the essence of my happiness,
my partner who holds my hand every step of the way and nurtures my ambition,
the one and only who flies with me beyond the boundaries of my imagination...

Acknowledgements

Throughout my educational journey and, particularly, my Ph.D. studies, I have received invaluable assistance and support that I would like to acknowledge.

I would like to express my sincere gratitude and appreciation to my advisor, Philippe Clauss, for his mentorship, support and encouragement. It has been a great honor to work with him and learn from him and his immense research experience. His energy and enthusiasm for research and life, sense of humor, understanding and trust have always motivated me throughout this challenging journey.

I would also like to express my gratitude to the reporters Christophe Alias and Denis Barthou and the examiners Alexandra Jimborean and Pierre-Etienne Moreau for their acceptance to serve on my Ph.D. thesis committee, their precious time and their insightful review of my research work.

My gratitude extends to the French Institute for Research in Computer Science and Automation (Inria) for the three-year funding opportunity to undertake my doctoral studies at the University of Strasbourg. Besides, I am grateful for the teaching opportunities that I have been granted at the University of Strasbourg throughout my studies, particularly for the ATER position at the Department of Computer Science at the Robert Schuman University Institute of Technology (IUT Robert Schuman) which has not only enriched my experience in academia, but has also helped funding my final Ph.D. year. A big thank you to my colleagues at the IUT Robert Schuman, particularly Pierre Kraemer and Mathieu Zimmermann, for their great support and cooperation.

In addition, I would like to convey my heartfelt gratitude to my colleagues and friends at the ICube Laboratory. First of all, I would like to thank the members of my research team ICPS. I would like to thank Jens Gustedt, Vincent Loechner, Stephane Genaud, Alain Ketterlin, Cédric Bastoul, Berenger Bramas, Arthur Charguéraud and the others for welcoming me among them. I will not forget the good times together, especially the canoe trip and the barbecue. Particularly, I would also like to thank Alain for his help and collaboration. Besides, I can never forget my friends and ex-colleagues Harenome Ranaivoarivony-Razanajato and Maxime Schmitt whom I would like to thank for their help, encouragement, all the fun and fond memories. Also, I would like to thank my friends Raquel Lazcano, Marek Felšöci and Paul Cardosi, with whom I shared the office B229, for the motivation and the pleasant time that we spent together full of profound conversations. I would like to thank them for staying by my side even after we were separated by distance. Second, I would like to thank my partners at the Association for Young Researchers at ICube (AJCI) for the good times. Third, I would like to thank my very first ICube friends, Chifaa Dahik and Hassan Mortada, for listening, supporting me and cheering me up.

Since my first steps on my educational path, I have had the chance to be inspired and

encouraged by many great educators: Wafaa Fakih, Mohamad Jaber, Samer Habre, the late Mohsen Jawad and many others. I would like to express my deep appreciation for them and their help that has shaped my path.

I am grateful beyond words for my family, my strong support system that I am lucky to have.

I am extremely grateful to have my mother Alia, the greatest mother and the strongest and kindest woman in the world, my first teacher, my forever-friend and the main source of my dreams and ambitions. I would like to thank her for the countless things that she has done for our sake, her love, selflessness, dedication to our education and future, non-stop encouragement and support, wisdom, precious advice and optimism in life.

I am very grateful for my father Lotfi who is a great man of value who has taught me and showed me how to be true to myself, stand up for what I believe in and achieve my goals through hard work, eagerness and perseverance.

I am also thankful for my brothers Oussama, Sameh and Mohammad because they have always believed in me and supported me. Having them by my side and learning from their success stories have influenced much my development and path in life. In fact, Oussama was the one that advised me to major in computer science. I would also like to thank my cousin Samer who invested in me and my education.

Furthermore, I would like to thank the other members of my family, my family-in-law, relatives, and friends who have always encouraged me to achieve my goals and dreams in life.

At last but not least, a special thank you to the one who means the whole world to me, to the one and only, my love and husband, Mohamed Jawad. The very first time we ever met, the first conversation we ever had was about doctoral studies. Ever since then, we have been together, I have been pursuing my studies, and he has always been there for me. I am very grateful for his unconditional love, caring, understanding, inexhaustible patience, powerful support, guidance, inspiration and motivation.

Contents

List of Figures

List of Listings

Abstract

In this thesis, we introduce Rec2Poly, a framework for speculative rewriting of recursive programs as affine loops that are candidates for efficient optimization and parallelization. Rec2Poly seeks a polyhedral-compliant run-time control and memory behavior in recursions making use of an offline profiling technique. When it succeeds to model the behavior of a recursive program as affine loops, it can use the affine loop model to automatically generate an optimized and parallelized code based on the inspector-executor strategy for the next executions of the program. The inspector involves a light version of the original recursive program whose role is to collect, generate and verify run-time information that is crucial to ensure the correctness of the equivalent affine iterative code. The executor is composed of the affine loops that can be parallelized or even optimized using the polyhedral model.

Résumé

Dans cette thèse, nous proposons Rec2Poly, un cadriciel pour la réécriture spéculative des programmes récursifs sous forme de boucles affines qui sont candidates à une parallélisation et une optimisation efficaces. Rec2Poly cherche un flot de contrôle dynamique et un comportement mémoire conformes au modèle polyédrique dans les récursions, en utilisant une technique de profilage hors ligne. Lorsqu'il réussit à modéliser le comportement d'un programme récursif sous forme de boucles affines, il peut utiliser le modèle de boucle affine pour générer automatiquement un code optimisé et parallélisé basé sur la stratégie inspecteur-exécuteur pour les prochaines exécutions du programme. L'inspecteur implique une version allégée du programme récursif d'origine dont le rôle est de collecter, générer et vérifier les informations d'exécution qui sont essentielles pour garantir l'exactitude du code itératif affine équivalent. L'exécuteur est composé des boucles affines qui peuvent être parallélisées voire optimisées à l'aide du modèle polyédrique.

Chapter 1

Introduction

"Any sufficiently advanced technology is indistinguishable from magic."
— Arthur C. Clarke

Throughout history, technological evolution has obviously influenced us and the development of our societies, economies and the whole world. Since the mid-twentieth century, with the beginning of the information age, the impact of the new technology has become radical. Technological advancement has dramatically re-shaped essential aspects of our lives and re-scaled to smaller dimensions our time and space. It has drastically accelerated our world, facilitated our lives on many levels, and in some cases, just like magic, made the impossible possible elevating our expectations and ambitions. Today more than ever, we find ourselves living in a fast-paced, fast-evolving and a demanding world with relatively limited resources. We find ourselves in need to keep up with its ever-increasing progress speed and efficiently satisfy its demands and attain our aspirations.

For the last decades, among the most prominent technological innovations, computers have become an integral part of this world. They have become indispensable for us in our daily life and to perform almost every effective and efficient work in any domain whether it is scientific, educational, medical, industrial, agricultural or financial... Moreover, along with other technologies, they have played a salient role extending our world to a new dimension, a virtual one. Virtual communities, virtual workplaces (through telecommuting) and virtual reality have got real; telecommuting, for instance, has helped numerous workers stay productive while safe and helped many businesses survive during the Covid-19 pandemic since 2019. Computers have spread everywhere taking different sizes and forms and serving various functionalities; we can find them embedded in our cars or washing machines, wearable as smart watches, handheld smart phones, general purpose laptops and desktops, supercomputers only existing in a few specific laboratories on Earth, or computers exclusively designed for spaceships with a mission to the Moon or Mars... Accordingly, computers have always needed to be up-to-date with the world's development and cope with humankind ambitious expectations. The research community and the computer industry have been devoting continuous efforts to make computers much more robust, intelligent and optimized; so, they can process as much data as possible, solve efficiently more complex problems and handle modern sophisticated applications while satisfying the ever-increasing demand for speed and high performance.

1.1 Computing Performance: Growth and Challenges

1.1.1 Hardware Perspective

In general, today, the substantial improvement in computer performance goes to decades of optimizations and upgrades mainly dedicated to the hardware in accordance with Moore's law and Dennard scaling. Moore's law was the outcome of predictions based on observations made by Gordon Moore in 1965 [92] and 1975 [93] about the future of integrated electronics and the semiconductor industry; it implied that the number of components or transistors in an integrated circuit would double about every two years. As for Dennard scaling, also known as MOSFET scaling, it was a scaling theory presented by Robert H. Dennard et al. in 1974 [37]; it suggested that as transistors would get smaller, both of the voltage and current would decrease such that transistors power density would stay constant. Moore's law combined with Dennard scaling meant that the performance per watt would also grow exponentially with the transistors density. Ever since then, keeping up with Moore's law, manufacturers have managed to miniaturize, speed up and cheapen transistors, and added more of them to the integrated circuits which has exponentially boosted the clock rate and the whole computer performance. In fact, by comparing today's Intel's 14 nanometer processors to the first microprocessors, performance has been boosted by about 3,500 times, energy efficiency has been improved by 90,000 times and price per transistor is decreased by about 60,000 times [97]. However, after more than a half century of considerable advancement in computing power, this strategy is reaching its peak. This is because, as transistors are becoming too small, Dennard scaling techniques are becoming less reliable and more difficult and challenging to use due to physical limitations [50], e.g., current leakage, increasing heat dissipation and overheating of the computer chips for which the cooling process requires high cost and power.

Alternatively, in order to satisfy the increasing demand for speed, hardware designers have combined multiple cores in a processor and introduced the new generation of multi-core processors and, then, multi-processor systems. Today, multi-processor computers are powerful and prevalent in all domains including high performance computing. Nevertheless, these computers still do not run at their full capacity because many of their processors may be not wholly utilized. In order to fully exploit their powerful hardware and obtain an optimal efficiency, the software must also be optimized and parallelized consistently with the underlying processor architecture.

1.1.2 Software Perspective: Parallel Computing

Parallel Programming Languages

For the purpose of adapting software for the new underlying hardware, many languages and extensions have been developed to design and write parallel programs (e.g., MPI [45], OpenMP [35] and POSIX Threads [98], etc.). However, implementing a reliable parallel software is still a notoriously difficult task for programmers. Facilitating this task can be achieved by automating programs' parallelization and optimization process at the level of compilers.

Compilers

A compiler is a computer program that interprets the source code of a program and translates the programming language in which it is written into another one. Also, compilers typically offer various optimization opportunities for programs. The purpose is to make programs execute better in terms of performance, memory usage and energy consumption. So, on a hardware platform, through a compiler, a software undergoes many optimizing transformation phases; eventually, the final executable code and the initial source code may differ, yet stay semantically equivalent. The optimization passes applied to the code often modify or remove instructions and control and data structures. Nevertheless, optimizations can apply at many different levels, and they may even transform the global structure of the code; for example, modern mainstream compilers (e.g., Clang, GCC) can eliminate tail-recursive calls by transforming them into loops.

Moreover, some advanced compilers and optimizers may go even further to automatically parallelize sequential codes by detecting parallel regions in them and automatically applying the corresponding transformations.

Polyhedral Optimizers On the one hand, many of such optimizers are based on the *Polyhedral Model* (e.g., Polly [52], Pluto [21]). The *Polyhedral Model* is a source-level (static) mathematical framework that contributes a substantial abstraction and representation for programs, particularly, affine loop nests accessing multi-dimensional arrays through affine array references, i.e., programs with static control parts (SCoPs). This framework provides powerful analysis, and aggressive loop automatic optimizing and parallelizing transformations (e.g., loop tiling, loop skewing, loop interchange). In the area of program compilation and optimization of imperative codes, loops are significant targets because they are common in programs and usually responsible for a huge compute-intensive part of the whole programs' executions. Yet, many programs may still not take advantage of the polyhedral optimizations due to either superficial languages idiosyncrasies (e.g., while loops), or radical language differences (e.g., recursive functions). However, sometimes, it turns out that loops, that do not have an affine structure at compile-time and do not fit into the polyhedral model, may actually exhibit a polyehdral-compliant behavior at run-time for at least great portions of the program execution. Therefore, there may still be optimization opportunities hidden at compile-time that can be uncovered and seized as soon as the run-time behavior is discovered.

Thread Level Speculation Systems On the other hand, there is the *Thread Level Speculation* technique that speculatively executes parallel regions of the code before knowing all input values and dependencies. Simultaneously, the sequential code is executed in parallel in a separate thread. At run-time, memory accesses are tracked and verification is performed, and in case an invalid speculation is proved (e.g., dependency violation), a recovery mechanism is performed. Recovery involves aborting the invalid speculative threads and re-initiating the sequential code from the last consistent point. This technique has been mainly dedicated to optimizing loop-structures. However, its success is not guaranteed and the offered performance gain may not be great due to unbalanced load, invalid speculations, the straightforward loop optimizations it offers unlike those of the polyhedral optimizers and inter-thread communications in the process of detect-

ing dependency violations.

Automatic speculative POLyhedral Loop Optimizer In this regard, Apollo a speculative polyhedral optimizer [132, 84] combines both approaches presented above. It has been implemented to capture transient polyhedral behaviors of statically non-affine loops by using dynamic profiling, and leverage the powerful polyhedral tools to optimize them aggressively at run-time.

But, what about the non-loop structures like recursive functions that cannot benefit from these automatic optimization and parallelization techniques? Are there alternatives in their cases that offer satisfying performance gains?

1.2 Motivation: Recursion Optimization

In a program execution, recursive functions, like loops, are also among the most noteworthy time-expensive structures responsible for a great part of the whole execution. In general, recursions implement complex algorithms, even for high performance computing, scanning and processing huge data structures, e.g., matrices, graphs and trees. Notwithstanding that recursive functions are interesting candidates for optimization and parallelization, they, unlike loops, do not benefit from advanced powerful automatic parallelization and optimization techniques. In the literature dealing with recursion optimization, recursive functions can be:

- handled directly as they are [53, 85]. They are mainly parallelized based on task parallelization such that several invocations are run simultaneously when the data dependencies among the invocations allows it. Besides, there exist recent techniques that allow polyhedral modeling of recursive invocations.

- firstly transformed into loops as it is always possible to replace a recursion by an equivalent loop nest and vice versa [5]. Usually, static recursion-to-loop transformation generates loops whose structures are complex and not affine and, thus, cannot benefit from further advanced optimizations.

However, the existing techniques do not capture when recursive codes can be rewritten as *affine* loops which are the most suitable candidates for efficient data locality and powerful polyhedral optimizing and parallelizing transformations. Correspondingly, the motivation of this thesis has been to take this chance and discover the optimization opportunities the polyhedral model can offer to recursions whose run-time behavior is affine.

1.3 Contributions

In this thesis, we present our speculative polyhedral recursion optimizer *Rec2Poly*.
Rec2Poly is a Clang/LLVM based framework devoted to the automatic transformation of recursive codes into sequences of affine loop nests. It involves a static analysis and code preparation phase to collect compile-time information about a target recursive code and

prepare it for the next phases. Rec2Poly also involves an offline profiling phase that detects a polyhedral-compliant behavior of the recursive code at run-time. The profiling phase includes instrumentation of the control and the memory behavior of the functions involved in the target recursions and *Nested Loop Recognition*. In case the recursive behavior trace can be modeled as an affine loop model, *Rec2Poly* commences its code generation phase and replaces all the execution flow related to recursive functions by a semantically equivalent affine iterative code based on the obtained model.

Furthermore, *Rec2Poly* applies polyhedral optimizing and parallelizing transformations to the so-generated sequences of loop nests.

However, since this transformation is based on offline profiling information, the behavior of the code may differ which means that the correctness of this transformation is not guaranteed for a different input data. Thus, *Rec2Poly*'s approach is speculative which requires a fast run-time verification mechanism to prove the validity of the performed transformations. Accordingly, *Rec2Poly* generates a verification process based on the *inspector-executor* paradigm [119, 112] in the final code generation phase. At run-time, the inspector verifies that the affine loops are valid regarding the current execution context and, in case they are valid, the executor launches the optimized parallel loops. Also, to save time-overhead, *Rec2Poly* executes the original recursive code in parallel with the inspector, so, in case an unpredicted behavior is detected, the original recursive code proceeds its execution while all the other parallel threads are aborted.

This approach is considered as dynamic code rewriting. It has been inspired by the Apollo approach, but instead of handling statically non-affine loops, it handles non-loop structures, recursions. *Rec2Poly*'s originality is twofold:

1. it seeks a polyhedral-compliant run-time behavior in recursions, and
2. it uses an inspector-executor scheme to verify not only memory access patterns as usual, but also the control flow against that of the affine loops.

1.4 Thesis Organization

This manuscript is organized as follows. Chapter 2 introduces the key concepts on which our study is based on including the polyhedral model and its optimizations, speculative loop optimization, and trace modeling as polyhedral loops with the Nested Loop Recognition Algorithm. Chapter 3 constitutes the literature and the state of the art of recursions and their optimization techniques. Then, Chapter 4 presents the main contribution of this thesis and all about the so-called Rec2poly framework. Chapter 5 shows benchmarks and experimental results of our approach. Finally, this manuscript ends with Chapter 6 with conclusions and future perspectives.

Chapter 2

Background

"In practical life we are compelled to follow what is most probable; in speculative thought we are compelled to follow truth"
— Baruch Spinoza

"Mathematics, rightly viewed, possesses not only truth, but supreme beauty."
— Bertrand Russell

In this chapter, we introduce the background and concepts behind our work. In Section 2.1, we give an overview of the polyhedral model by presenting its theoretical mathematical notations, its main characteristics and components, and some of the notable existing polyhedral-based optimizers and softwares and their capabilities and limitations. Then, in Section 2.2, we introduce the speculative loop optimization approach with the inspector-executor paradigm and the Apollo framework. Finally, in Section 2.3, we present the Nested Loop Recognition algorithm that is used to model traces as polyhedral loops.

2.1 The Polyhedral Model

The polyhedral or polytope model [21, 44] is a mathematical and geometrical framework providing a powerful abstraction and representation of nested-loop programs that enables advanced analysis and sophisticated transformations and optimization heuristics. This model is dedicated to loop nests that have a specific structure satisfying restrictions on loop bounds and data accesses at compile time; these loop nests are known as affine loop nests or static control parts (SCoPs).

At some point during compilation, all traditional compilers represent source codes in intermediate representations as abstract syntax trees (AST) or control flow graphs (CFG) where usually their basic components are the instructions or statements composing the code. However, these traditional representations are not sufficient to support powerful optimization and auto-parallelization because parallelism, specifically within loops, occurs among statements instances rather than among statements; they are not adequate to model dependences among instances and to enable complex loop optimizing transformations (e.g., loop tiling, interchange, skewing, etc.) on the time-expensive loop-structures.

In comparison, the polyhedral model is distinguished because it handles each loop iteration or each statement instance separately. It associates each iteration or tuple of loop indices values with a lattice point contained in a $\mathbb{Z}$-polyhedron bounded by affine inequalities. Representing loop nest as polyhedra helps reasoning about various advanced valid transformations and optimizations that can be performed.

The polyhedral framework includes an accurate data dependence analysis that is performed among the statements instances across different iterations of a target loop nest. This analysis step is vital to prove the correctness of the optimizing transformations. If the transformations do not violate the data dependences, then the optimized version of the loop nest is semantically equivalent to the original loop nest.

The loop nest optimizations are linear transformations of the polyhedra representing the iteration domains. Also, these transformations can be defined as scheduling matrices specifying re-orderings of statement instances execution with respect to each other across different iterations. Consequently, they convert the source polyhedra into another equivalent form. Such analysis and transformations can be performed using linear programming tools.

The theory of this model originated decades ago from a series of seminal contributions starting by "The Organization of Computations for Uniform Recurrence Equations" by Karp, Miller, and Winograd in 1968 [61]. Since the 1980s, many techniques, tools and libraries have been developed making possible applying polyhedral optimizations on loop nests like PIP [39], Polylib [81, 145], Omega [63, 107], ISL [142] and Pluto [21]. The research community has continued with its efforts extending this concept and its application and creating and improving the polyhedral techniques [107, 21, 142, 22, 42, 51, 73, 10, 17]. Accordingly, the polyhedral model has revolutionized the domain of automatic optimization and parallelization of iterative programs. Now, even current production compilers make use of the polyhedral model to compile programs for multi-core architectures such as R-Stream [44], GCC whose polyhedral framework is Graphite [104] and Clang-LLVM that implements Polly [102] for polyhedral optimizations[1].

In what follows in this section, we define basic mathematical notations on which the polyhedral model stands. Then, we explain in more details the polyhedral representation of programs defining the notions of *static control parts*, *iteration domains*, *access functions* and *scheduling functions*. Moreover, we discuss dependence analysis and optimizations in the polyhedral model. Finally, we present some polyhedral optimizers including those that we used in this work, their capabilities and their limitations.

2.1.1 Mathematical Background and Notations

The following definitions and notations are based on the book entitled "Theory of Linear and Integer Programming" by Alexander Schrijver [123].

Definition 2.1.1. (*Affine Function*). A Function $f : \mathbb{K}^m \rightarrow \mathbb{K}^n$ is affine $\iff$ $\exists$ a matrix $A \in \mathbb{K}^{n \times m}$ and a vector $\vec{b} \in \mathbb{K}^n$ such that $\forall \vec{x} \in \mathbb{K}^m$:

$$f(\vec{x}) = A\vec{x} + \vec{b}$$

In other words, an affine function is a linear function with a translation such that a linear function is affine whereas the converse is not necessarily true.

[1] Check out `https://polyhedral.info/` to know more about the polyhedral community.

Definition 2.1.2. (*Affine Hyperplane*). An affine hyperplane is an affine subspace of dimension $(n-1)$ of an affine n-dimensional space $\mathbb{K}^n$. The affine hyperplane is the set of all vectors $\vec{x} \in \mathbb{K}^n$ defined by the linear equation:

$$\vec{a} \cdot \vec{x} = b, \text{ where } \vec{a} \in \mathbb{K}^n \ (\vec{a} \neq \vec{0}) \text{ and } b \in \mathbb{K}$$

In general, a hyperplane is a subspace with a dimension that is less than that of its ambient space by only one. For instance, the hyperplanes of a three-dimensional space are planes, those of a two-dimensional space are lines, and those of one-dimentional space are points.

Definition 2.1.3. (*Affine half-space*). A hyperplane that splits the space into two half-spaces, defined by the following inequalities

$$\vec{a} \cdot \vec{x} \leq b \text{ and } \vec{a} \cdot \vec{x} \geq b, \text{ where } \vec{a} \in \mathbb{K}^n \ (\vec{a} \neq \vec{0}) \text{ and } b \in \mathbb{K}$$

Definition 2.1.4. (*Convex Polyhedron*). A convex polyhedron is the intersection of a finite number of affine half-spaces. A convex polyhedron $P \subset \mathbb{K}^n$ is defined by a set of vectors in $\mathbb{K}^n$ bounded by a set of affine contraints represented by matrix $A \in \mathbb{K}^{m \times n}$ and a constraint vector $\vec{b} \in \mathbb{K}^m$:

$$P = \{\vec{x} \in \mathbb{K}^n \mid A\vec{x} + \vec{b} \geq \vec{0}\}$$

Definition 2.1.5. (*Parametric Polyhedron*). A parametric polyhedron denoted by the set $P(\vec{p})$ is a polyhedron parametrized by the vector of symbolic values $\vec{p} \in \mathbb{K}^p$. It can be defined by a matrix $A \in \mathbb{K}^{m \times n}$, a matrix of symbolic coefficients $B \in \mathbb{K}^{m \times p}$ and a vector $\vec{b} \in \mathbb{K}^m$:

$$P(\vec{p}) = \{\vec{x} \in \mathbb{K}^n \mid A\vec{x} + B\vec{p} + \vec{b} \geq \vec{0}\}$$

Definition 2.1.6. (*Polytope*). A polytope is a bounded polyhedron.

Definition 2.1.7. ($\mathbb{Z}$-*Polyhedron*). It is polyhedron whose extreme points are of type integer ($\mathbb{K} = \mathbb{Z}$). Also, it can be defined by the set of points of an affine lattice or integer points in a polyhedron.

In our context, we are exclusively interested in the points with integer coordinates contained in $\mathbb{Z}$-polyhedra. A more accurate and detailed presentation of the polyhedral model can be found in "Polyhedron Model" [44].

2.1.2 Polyhedral Representation of Programs

As mentioned earlier, one of the main features that distinguishes the polyhedral model from the other classical program representations is that it provides information about different execution instances of statements at compile time (representing them as points of a $\mathbb{Z}$-polyhedron) whenever possible.

SCoP: The Scope of the Polyhedral Model

A target loop structure that fits the polyhedral model must have an execution control flow and data dependences that are analyzable statically at compile time.

Accordingly, the polyhedral model is particularly dedicated for specific code parts made up of sequences of affine for-loop nests where:

- Each loop has a unique iterator and loop bounds are affine expressions of the enclosing loops iterators and parameters; a parameter is an integer variable whose value is unknown at compile time but remains constant at run-time.
- The only allowed control structures other than the for-loops are conditionals which are also affine expressions of the surrounding loop iterators and parameters; function calls and instructions that may break the control flow, e.g., break, goto or return, are not allowed.
- All other instructions are data access functions and memory instructions accessing either simple scalar variables or multi-dimensional array elements referenced using affine functions on the outer loops indices and parameters.

Note that these loop nests may be either perfect or imperfect. Generally, a loop nest is considered perfect if and only if all of its statements uniquely appear in the body of the innermost loop; otherwise, it is considered as an imperfect loop nest. Listings 2.1 and 2.2 show examples of a perfect loop nest and an imperfect one respectively.

```
for ( i = 0 ; i < N ; i++ )
    for ( j = 0 ; j < N ; j++ )
        A[i][j] = B[i][j];//S(i,j)
```

Listing 2.1 – Perfect Loop Nest

```
for ( i = 0 ; i < N ; i++ ) {
    C[i]=0;//S1(i)
    for ( j = 0 ; j < N ; j++ )
        C[i]+=A[i][j]*B[j];//S2(i,j)
}
```

Listing 2.2 – Imperfect Loop Nest

Such code parts whose control and data dependences can be statically analyzed are called static control parts or SCoPs for short [40, 42, 99, 17]. The definition of a SCoP can be summarized as follows.

Definition 2.1.8. (*Static Control Part* (*SCoP*)). A SCoP is defined as a maximal set of consecutive statements, including data access functions and memory instructions to scalar variables and multi-dimensional arrays, where the only allowed control structures are if-statements and for-loops such that array indices, conditions and loop bounds are affine functions of the enclosing loop iterators and parameters; parameters are symbolic values/variables defined outside a SCoP whose values are unknown at compile time but remains constant during its execution.

An example of a valid SCoP is the iterative matrix multiplication shown in Listing 2.3. In this SCoP, there is a nest of three for-loops such that i, j and k are the iterators corresponding to the outer, middle, and inner loops respectively. Also, there are two statements $S1(i, j)$, enclosed by the two outer loops, and $S2(i, j, k)$ in the body of the innermost loop such that their memory accesses are affine expressions of their enclosing loops.

```
for ( i = 0 ; i < N ; i++ )
    for ( j = 0 ; j < N ; j++ ) {
        C[i][j] = 0 ; //S1(i,j)
        for ( k = 0 ; k < N ; k++ )
            C[i][j] += A[i][k] * B[k][j] ; //S2(i,j,k)
    }
```

Listing 2.3 – Example of a Valid SCoP: Matrix Multiplication Kernel

The Polyhedra of Iteration Domains

The polyhedral model represents at compile time the whole execution of a SCoP, i.e., all dynamic instances of its statements. A single dynamic instance of a statement is described by an iteration vector that is defined as follows.

Definition 2.1.9. (*Iteration vector*). An iteration vector of a statement S is the vector containing values of the iterators of all its surrounding loops. Each statement is executed once for each of the iteration vectors. Correspondingly, an iteration vector describes an occurrence of a certain statement at a specific iteration in a loop nest, i.e., a dynamic instance of this statement. An iteration vector $\vec{x} \in \mathbb{Z}^n$ of a statement S can be defined as:

$$\vec{x} = (x_1, ..., x_n), \text{ where } n \text{ is the depth of the enclosing loop nest.}$$

$S(\vec{x})$ denotes a dynamic instance of $S \in \mathbb{Z}^n$ that can be represented geometrically using the coordinates of the corresponding iteration vector $\vec{x}$, i.e., $(x_1, ..., x_n)$.

For instance, consider the affine loop nest or SCoP in Listing 2.4. It is composed of two nested loops and an affine conditional statement enclosing a statement $S1$. The iterator of the outer loop is i and that of the inner loop is j. The outer loop iterates from 1 to an upper bound N such that N is a loop parameter, and the inner loop iterates from 1 until its iterator j hits the upper bound $N+i-2$ which is an affine expression in terms of the outer loop iterator i and the parameter N. Statement $S1$ is only executed when the loop nest is executed and all the existing affine conditions are satisfied which are: $1 \leq i \leq N$, $1 \leq j \leq N+i-2$, $i \leq j+3$ and $i \leq N-j+4$.

Then, the two-dimensional vectors of the form (i, j) where the values of iterators i and j are valid with respect to the loop nest execution constraints are iteration vectors of $S1$. Assuming that N is even and taking into consideration all the constraints, the set of all points of the execution instances of $S1$ in the order of execution would be as follows:

$$\begin{array}{l}
(1,1),\ (1,2),\ (1,3),\ \ldots,\ (1,N-1), \\
(2,1),\ (2,2),\ (2,3),\ \ldots,\ (2,N-1),\ (2,N), \\
(3,1),\ \ldots\ldots\ldots\ldots\ldots\ldots,\ (3,N),\ (3,N+1), \\
(4,1),\ \ldots\ldots\ldots\ldots\ldots\ldots,\ (4,N), \\
(5,2),\ \ldots\ldots\ldots\ldots,\ (5,N-1), \\
\ldots\ldots\ldots\ldots\ldots\ldots, \\
(\frac{N}{2}+3, \frac{N}{2}),\ (\frac{N}{2}+3, \frac{N}{2}+1)
\end{array}$$

All the execution points or instances of statement $S1$ constitute the so-called iteration domain of $S1$ [17, 14] which can be expressed using the following mathematical expression:

$$D^{S1} = \{(i,j) \in \mathbb{Z}^2 | (1 \leq i \leq N) \wedge (1 \leq j \leq N+i-2) \wedge (i \leq j+3) \wedge (i \leq N-j+4)\}$$

Accordingly, the iteration domain is expressed using a system of affine inequalities that represent the corresponding statement's execution constraints. Also, the iteration domain can be illustrated geometrically as a set of points within a parametric $\mathbb{Z}$-polyhedron bounded by these inequalities.

The polytope embodying the iteration domain of $S1$ in the affine loop nest shown in Listing 2.4 is graphically displayed in Figure 2.1. It is two-dimensional because $S1$ is in the body of a loop nest made up of only two loops. Every lattice point or a dot appearing in the two-dimensional Cartesian coordinate system within the polytope corresponds to a dynamic instance of the statement $S1$.

```
for ( i = 1 ; i <= N ; i++ )
     for ( j = 1; j <= N+i-2; j++ )
         if ( i <= j+3 && i <= N-j+4 )
                 S1(i,j);
```

Listing 2.4 – SCoP / Affine Loop Nest

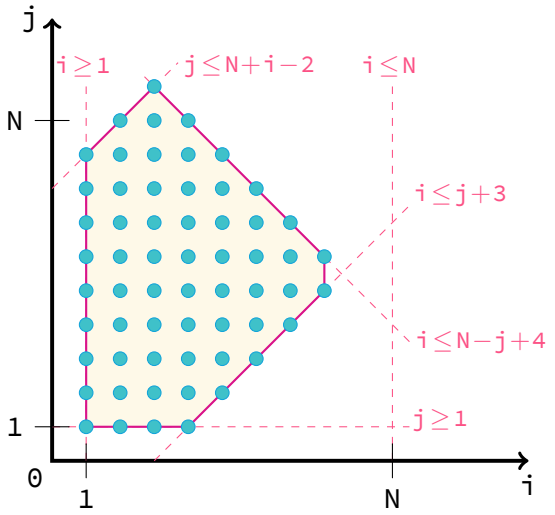


Figure 2.1 – Iteration Domain Polyhedron

The iteration domain is the essence of the polyhedral model, and it can be more precisely defined and expressed using a matrix representation as follows.

Definition 2.1.10. (*Iteration Domain*). The iteration domain of a statement S enclosed by n loops is the set of integer lattice points corresponding to the execution instances of S modeled as an n-dimensional parametric $\mathbb{Z}$-polyhedron, $D^S(\vec{p}) \subseteq \mathbb{Z}^n$, bounded by affine inequalities representing the execution constraints such that:

$$D^S(\vec{p}) = \{\vec{x} \in \mathbb{Z}^n \mid A\vec{x} + B\vec{p} + \vec{b} \geq \vec{0}\}$$

where $\vec{x}$ is the iteration vector, $\vec{p}$ is the vector of parameters, A is the iterators coefficient matrix, B is the symbolic (parametric) coefficients matrix and $\vec{b}$ is a constant vector, all encoded from the corresponding system of inequalities.

In what follows, on the left, we show the domain constraints of statement $S1$ in Listing 2.4 represented as a system of inequalities and, on the right, we show the corresponding iteration domain expression using matrices encoding the constraints in a canonical form.

$$\begin{cases} i-1\geq 0 \\ j-1\geq 0 \\ -i+N\geq 0 \\ i-j+N-2\geq 0 \\ -i+j+3\geq 0 \\ -i-j+N+4\geq 0 \end{cases} \quad D^{S1}=\left\{ \begin{pmatrix} i \\ j \end{pmatrix} \in \mathbb{Z}^2 \;\middle|\; \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 0 \\ 1 & -1 \\ -1 & 1 \\ -1 & -1 \end{bmatrix} \cdot \begin{pmatrix} i \\ j \end{pmatrix} + \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \end{bmatrix} N + \begin{bmatrix} -1 \\ -1 \\ 0 \\ -2 \\ 3 \\ 4 \end{bmatrix} \geq \vec{0} \right\}$$

The following representation resulting from merging the coefficient matrices together and the vectors, is a more compact and a more commonly used representation of the iteration domain:

$$D^{S1}=\left\{ \begin{pmatrix} i \\ j \end{pmatrix} \in \mathbb{Z}^2 \;\middle|\; \left[\begin{array}{cc|c|c} 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & -1 \\ -1 & 0 & 1 & 0 \\ 1 & -1 & 1 & -2 \\ -1 & 1 & 0 & 3 \\ -1 & -1 & 1 & 4 \end{array}\right] \cdot \left(\begin{array}{c} i \\ j \\ \hline N \\ \hline 1 \end{array}\right) \geq \vec{0} \right\}$$

Point in Polyhedron: Points to Memory

Each instance of a statement or a point in the corresponding iteration polyhedron points to exact memory locations accessed by this instance. This way all the information required to perform the accurate dependence analysis are acquired.

Definition 2.1.11. (*Access Functions*). The access functions of a statement S are the set of affine functions that map all the statement instances with the exact memory locations to which they read or write. Access functions of statement S can be represented as:

$$f^S_{\{R,W\}}(\vec{x}) = F\vec{x} + \vec{f}$$

where $F \in \mathbb{Z}^{d\times n}$ is the coefficient matrix of array subscripts, d is the dimension of the array, n is the enclosing loop nest depth, $\vec{x}$ is an iteration vector of S, $\vec{f}$ is a constant vector $\in \mathbb{Z}^d$ and, finally, R and W precise whether the statement is read or write respectively.

In the polyhedral model, memory statements access one or more multi-dimensional arrays through affine subscripts or, even, scalar variables that are treated, in this context, as zero-dimensional arrays.

Note that a more compact representation of access functions is usually implemented on computers, i.e.:

$$f^S(\vec{v}) = F'\vec{v}, \quad \text{where } \vec{v}= \begin{pmatrix} \vec{x} \\ \vec{p} \\ 1 \end{pmatrix}$$

$\vec{p}$ is the vector of parameters, F' is a coefficient matrix $\in \mathbb{Z}^{d\times(n+p+1)}$ and $p = |\vec{p}|$.

In this SCoP, shown in Listing 2.5, instruction $S1$ is surrounded by three loops with indices i, j and k and parameter N, and it accesses five memory locations: it reads from $C[i][j]$, $A[i][k]$, $B[N-k-1]$ and $alpha$ and writes to $C[i][j]$. A and C are two-dimensional arrays, B is a one-dimensional array and variable $alpha$ is treated as a zero-dimensional array.

```
for (i = 0 ; i < N ; i++ )
    for ( j = 0 ; j < N ; j++ )
        for ( k = 0 ; k < N ; k++ )
            C[i][j] += A[i][k] * B[N-k-1] + alpha; //S1
```

Listing 2.5 – SCoP Example to Illustrate Access Functions

The access functions corresponding to $S1$ are listed below:

$$f_{W^C}^{S1} = f_{R^C}^{S1} = \left[\begin{array}{ccc|c|c} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{array}\right].\begin{pmatrix} i & j & k & N & 1 \end{pmatrix}^{\intercal} \longmapsto C[i][j]$$

$$f_{R^A}^{S1} = \left[\begin{array}{ccc|c|c} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{array}\right].\begin{pmatrix} i & j & k & N & 1 \end{pmatrix}^{\intercal} \longmapsto A[i][k]$$

$$f_{R^B}^{S1} = \left[\begin{array}{ccc|c|c} 0 & 0 & -1 & 1 & -1 \end{array}\right].\begin{pmatrix} i & j & k & N & 1 \end{pmatrix}^{\intercal} \longmapsto B[N-k-1][j]$$

$$f_{R^x}^{S1} = \left[\begin{array}{ccc|c|c} 0 & 0 & 0 & 0 & 0 \end{array}\right].\begin{pmatrix} i & j & k & N & 1 \end{pmatrix}^{\intercal} \longmapsto \&(alpha)[0]$$

Scheduling

The iteration domain and access functions are still not sufficiently expressive to describe the whole execution of an affine loop nest. For now, we only know which statements instances are actually executed and where exactly they touch the memory. Yet, there is no provided information about the initial execution order of these instances which is essential for applying loop transformations and optimizations. Loop transformations involve reordering the dynamic instances of statements, for this reason, their initial order is a prerequisite and must be formally specified.

The polyhedral model determines the execution order of instances by associating a logical execution date, i.e., a multi-dimensional timestamp as proposed by Feautrier [42] and later by Kelly and Pugh [62] to every statement instance in a SCoP. Timestamps define instances orders; using them enables sorting the statements instances according to the initial order of the sequential execution. Scheduling functions, defined below, are used to assign logical execution dates to statements instances.

Definition 2.1.12. (*Scheduling Function*). The scheduling function of a statement S a.k.a the affine schedule of S is a function that maps each dynamic instance of S to a logical timestamp defining the relative execution order between statements in the SCoP:

$$\forall \vec{x} \in D^S, \theta^S(\vec{x}) = \vec{t}$$

The associated timestamps allow to order the instances of the statements according to the lexicographical order.

Definition 2.1.13. (*Lexicographical order*). The lexicographical order, denoted by $\ll$, is defined as:

$$(a_1, ..., a_n) \ll (b_1, ..., b_n) \iff \exists i : 1 \leq i \leq n, \forall j : 1 \leq j < i, a_j = b_j \wedge a_i < b_i$$

An instance $S(\vec{x})$ executes before another instance $S'(\vec{y})$ iff $\theta^S(\vec{x}) \ll \theta^{S'}(\vec{y})$.

If two statements map to the same logical execution date, they can be executed in parallel or any arbitrary order with respect to each other.

Although it may seem that the iteration vectors can serve as multidimensional timestamps and that the iteration domain may adequately describe the order of dynamic instances of statements, this is not true; this is because they do not capture the textual ordering of statements, i.e., the execution order of statements regarding their position w.r.t. the other statements within the loop nest.

```
for (i = 0; i < N; i++){  //L1
   S1;
   for (j = 0; j < N; j++) {  //L2
       S2;
       S3;
   }
   S4;
}
```

Listing 2.6 – SCoP Example to Illustrate Scheduling

For instance, consider the loop nest in Listing 2.6 composed of two loops; we refer to the outer loop with iterator i as $L1$ and to the inner loop with iterator j as $L2$. On one hand, statements $S1$ and $S4$ are enclosed by $L1$ only, and they have the same iteration vectors (composed of values of i only) during the whole execution despite the fact that they are not actually executed at the same time: $S1$ executes first, loop $L2$ executes second, then comes the turn of $S4$. On the other hand, $S2$ and $S3$ are enclosed in $L2$ and they have the same iteration vectors through the whole execution. However, $S2$ executes before $S3$. Moreover, all of $S1$, $S2$, $S3$ and $S4$ execute for all values of iterator i, so at each distinct iteration, their iteration vectors have the same value of i. In this case, iteration vectors fail to indicate the correct relative execution order

Scheduling functions solve this issue by interleaving the iteration vectors with constant values representing the relative textual order of statements at each common loop level. A schedule can be obtained using the abstract syntax tree of the SCoP with Dewey Decimal Numbers [42]. A number on the path from a source node to a destination node represents the position of the statement/loop corresponding to the latter w.r.t. the loop corresponding to the prior. The AST of the SCoP of 2.6 is shown in Figure 2.2.

Below to the left we have the timestamps of $S1$, $S2$, $S3$ and $S4$ respectively:

- $\theta^{S1}((i,j)) = (0,i,0,0,0)$
- $\theta^{S2}((i)) = (0,i,1,j,0)$
- $\theta^{S3}((i)) = (0,i,1,j,1)$
- $\theta^{S4}((i,j)) = (0,i,2,0,0)$

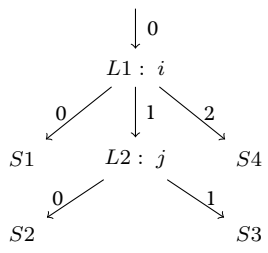


Figure 2.2 – Abstract Syntax Tree

Furthermore, an affine schedule of a statement S can be represented using a matrix and can be redefined as:

$$\theta^S(\vec{v}) = \Theta^S \vec{v} \quad \text{where } \vec{v} = \begin{pmatrix} \vec{x} \\ \vec{p} \\ 1 \end{pmatrix}, \ \forall \vec{x} \in D^S$$

Θ^S is the scheduling matrix of S such that $\Theta^S \in \mathbb{Z}^{d^t \times (n+p+1)}$ with $d^t = |\vec{t}| = 2n+1$, $p = |\vec{p}|$ and n is the loop nest depth.

A normalized representation of the scheduling matrix has been proposed by Cohen et. al. [32, 14] encoding it as follows:

$$\Theta^S = \left[\begin{array}{ccc|ccc|c} 0 & \dots & 0 & 0 & \dots & 0 & \beta^S \\ A^S_{1,1} & \dots & A^S_{1,n} & \Gamma_{1,1} & \dots & \Gamma^S_{1,p} & \Gamma^S_{1,p+1} \\ 0 & \dots & 0 & 0 & \dots & 0 & \beta^S_2 \\ A^S_{2,1} & \dots & A^S_{2,n} & \Gamma_{2,1} & \dots & \Gamma^S_{2,p} & \Gamma^S_{2,p+1} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ A^S_{n,1} & \dots & A^S_{n,n} & \Gamma_{n,1} & \dots & \Gamma^S_{n,p} & \Gamma^S_{n,p+1} \\ 0 & \dots & 0 & 0 & \dots & 0 & \beta^S_{n+1} \end{array}\right]$$

The scheduling matrix encodes both static and dynamic information about the original schedule of a statement S. It can be also used to precisely express the composition of different transformations as any modification to this matrix changes the execution order of the corresponding statement instances. It is made up of three sub-matrices: A^S, Γ^S and β^S such that:

1. $A^S \in \mathbb{Z}^{n \times n}$ is the iteration ordering matrix operating on the iteration vectors.

2. $\Gamma^S \in \mathbb{Z}^{n \times (p+1)}$ is the parametrized matrix, i.e., the matrix of loops parameters.

3. $\beta^S \in \mathbb{Z}^{n+1}$ is the statement-scattering vector that encodes the relative textual position of the statements in the loop nest.

Accordingly, we can also re-describe the original execution order of the statements in Listing 2.6 using this matrix format. The sub-matrices composing the scheduling matrices corresponding to $S1$, $S2$, $S3$ and $S4$ are listed below:

$$A^{S1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad A^{S2} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad A^{S3} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad A^{S4} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$\Gamma^{S1} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \quad \Gamma^{S2} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \quad \Gamma^{S3} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \quad \Gamma^{S4} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

$$\beta^{S1} = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^\intercal \quad \beta^{S2} = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}^\intercal \quad \beta^{S3} = \begin{bmatrix} 0 & 1 & 1 \end{bmatrix}^\intercal \quad \beta^{S4} = \begin{bmatrix} 0 & 2 & 0 \end{bmatrix}^\intercal$$

Schedules: Transformations Triggers

Generally, loop transformations require reordering or restructuring the loops in a loop nest. The polyhedral model enables such loop transformations by reordering the execution instances of the statements in the loop nests of interest.

Interesting sophisticated affine transformations can be achieved by applying a sequence of simpler transformations (e.g., loop interchange, or skewing). They are specified by affine scheduling functions that alter the original polyhedra of the statements iteration domains into new polyhedra containing the same points, but in a different execution order in a new coordinate system [11].

In scheduling functions, scheduling matrices are exploited to express different execution orders of the corresponding statements besides the initial sequential order.

"Matrices act. They don't just sit there."— William Gilbert Strang [128]

The modifications applied to the sub-matrices (A^S,β^S and Γ^S) of the scheduling matrix Θ^S of statement S in an affine loop nest impact directly the original execution order representation of the instances of S which enables applying directly loop transformations.

Popular uni-modular loop transformations, e.g., loop interchange, reversal, and skewing are possible by modifying the iteration vectors, i.e., modifying iteration ordering coefficients in A^S.

For instance, consider the simple affine loop nest Listing 2.7 composed of an outer loop whose iterator is i and an inner loop whose iterator is j enclosing one statement $S1$. ${\theta_o}^{S1}(\vec{v})$ (below the listing) such that $\vec{v} = (i \ j \ 1)^\intercal$ describes the original execution order of $S1$'s instances using the scheduling matrix in normalized form. In Figure 2.3, we have the polyhedron describing the iteration domain of $S1$ with the execution flow of its dynamic instances marked by arrows. Note that since there are no loop parameters, the scheduling matrix is simplified.

```
for ( i = 1; i <= 3; i++ )
    for ( j = 1; j <= 2; j++ )
        A[i][j]=...; //S1
```

Listing 2.7 – Affine Loop Nest

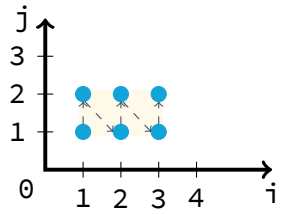


Figure 2.3 – S1 Iteration Domain Polyhedron

$$\theta_o^{S1}(\vec{v}) = \left[\begin{array}{cc|c} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{array}\right] \vec{v} = \begin{pmatrix} 0 \\ i \\ 0 \\ j \\ 0 \end{pmatrix}$$

In order to perform a certain loop skewing, it is sufficient to modify, in the original scheduling matrix, the iteration ordering sub-matrix, as:

$$A^{S1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \text{ to } \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$$

This change can be viewed in the scheduling matrix used in the new scheduling function

responsible for skewing: $\theta_s^S(\vec{v})$. Listing 2.8 shows the affine loop nest after performing loop skewing and the polyhedron in Figure 2.4 is the corresponding polyhedron. The transformed polyhedron contains the same instances points as the initial polyhedron but in a different order.

$$\theta_s^{S1}(\vec{v}) = \left[\begin{array}{cc|c} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \end{array}\right] \vec{v} = \begin{pmatrix} 0 \\ i \\ 0 \\ i+j \\ 0 \end{pmatrix}$$

```
for ( i = 1; i <= 3; i++ )
    for ( j = 1+i; j <= 2+i; j++ )
        A[i][j-i]=...; //S1
```

Listing 2.8 – Loop Nest After Loop Skewing

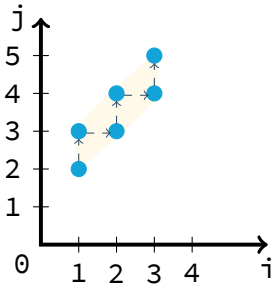


Figure 2.4 – Polyhedron after Loop Skewing

On the other hand, other loop transformations like shifting are expressed using the coefficients of component Γ^S, and transformations like Loop fission, loop fusion, and code motion are applied by modifying the textual order encoded in β^S.

Furthermore, although other interesting transformations like tiling require altering the iteration domain, they are possible, and can be still expressed using scheduling functions.

After this brief presentation of the optimizing transformations and how they can be enabled in the polyhedral model, we discuss next the validity of these transformations and when they can be reliably enabled.

2.1.3 Dependence Analysis

The validity or legality of polyhedral transformations and optimizations must be verified and ensured. The new scheduling must preserve the semantics of the original program which can be guaranteed by proving that data dependences among statements are not violated [64, 18].

Definition 2.1.14. (*Data Dependence*). Two statements S and S' are said to be data dependent, iff there exist instances $S(\vec{x})$ of S and $S'(\vec{y})$ of S' such that they both access the same memory location and at least one of these accesses is a write access. The existence of a dependence can be formally expressed using the Bernstein Condition as follows:

$$[W(S) \cap R(S')] \cup [R(S) \cap W(S')] \cup [W(S) \cap W(S')] \neq \varnothing$$

such that $W(S_i)$ and $R(S_i)$ represent the sets of memory locations written and read by the statement S_i respectively and there exists a feasible run-time execution path from S to S'. If $S(\vec{x})$ is executed before $S'(\vec{y})$ in the initial sequential order ($\theta^S(\vec{x}) \ll \theta^{S'}(\vec{y})$), then S' is considered to be dependent on S such that S is the source of this dependence and S' is its destination, sink or target.

Generally, the semantics of a program are preserved if the initial execution order of dependent statements remains unchanged even after program transformation and parallelization. For example, if in the original program an instance $S(\vec{x})$ writes to a memory

location that is supposed to be read afterwords by another instance $S'(\vec{y})$, then any transformation that reorders these instances and executes $S'(\vec{y})$ before $S(\vec{x})$ is invalid because the value read by $S'(\vec{y})$ may be wrong. Conversely, reordering independent statements arbitrarily will not tamper with the original program semantics.

There exist several kinds of data dependences that must be respected which are distinguished according to the order and the type of the memory accesses performed (read or write) as follows:

- ***RAW***: read-after-write, flow or true dependence:
  Statement S' is flow dependent on S, denoted as $S\delta S' \iff W(S) \cap R(S') \neq \varnothing$,
  i.e., S writes to a memory location before S' reads from it.

- ***WAR***: write-after-read or anti-dependence:
  Statement S' is anti-dependent on S, $S\delta^{-1}S' \iff R(S) \cap W(S') \neq \varnothing$,
  i.e., S reads a memory location before S' overwrites it.

- ***WAW***: write-after-write or output dependence:
  Statement S' is output dependent on S, $S\delta^{o}S' \iff W(S) \cap W(S') \neq \varnothing$,
  i.e., S writes to a memory location before S' overwrites it.

There may also occur a read-after-read (***RAR***) or input dependence in which all involved statements instances access a memory location only for reading without modifying the memory; accordingly, it is not considered as an actual dependence since reordering arbitrarily the execution of two reads will not affect the semantics of the program. Nevertheless, RARs can be still analyzed for the sake of improving data locality which is interesting for some optimisations.

There are multiple techniques to eliminate WAR and WAW dependences[26], e.g., renaming, expansion, node splitting, etc., which enable more optimisation opportunities. However, RAW dependence cannot be removed by these techniques, and the polyhedral model does not adopt any of them.

For example, let us consider the Listing 2.9. There are two statements $S1$ and $S2$ in a loop nest of depth equal to two. We refer to the outer loop, i.e., the first loop in the nest whose iterator is i as $L1$ and to the inner loop or the second loop whose iterator is j as $L2$. Besides, we have P as a parameter.

The memory location $A[i][j+1]$ read by the instance $S1$ at iteration $(i, j) = (x_1, x_2)$ such that $1 \leq x_1 < P$ and $1 \leq x_2 < P$ is later modified as $A[i][j]$ by another instance of $S1$ at iteration (y_1, y_2) such that $y_1 = x_1$ and $y_2 = x_2 + 1$. Then, $S1$ is (WAR) anti-dependent on itself ($S1\ \delta^{-1}\ S1$).

The value of $A[i][j]$ modified by the instance $S1$ at iteration (x_1, x_2) is later read as $A[i-1][j+1]$ by instance of $S2$ at iteration (y_1, y_2) such that $y_1 = x_1+1$ and $y_2 = x_2-1$. So, $S2$ is said to be (RAW) flow dependent on $S1$ such that $S1$ and $S2$ are the source and the target of this dependency respectively ($S1\ \delta\ S2$).

Also, the value of $B[i][j]$ read by $S1$ at iteration (x_1, x_2) is rewritten later by an instance of $S2$ at the same iteration $(y_1, y_2) = (x_1, x_2)$ which means that there is an anti-dependence (WAR) of $S2$ on $S1$ ($S1\ \delta^{-1}\ S2$).

Usually, a data dependence between statements can be visualized in a data dependence graph.

Definition 2.1.15. (*Dependence Graph*). A dependence graph $G = (V, E)$ is a directed graph composed of a set of vertices V representing the statements and edges E representing the data dependences. Every edge $e = (v, v') \in E$ where v and $v' \in V$ corresponds to a dependence from the source statement represented by v to the destination statement represented by v'.

The dependence graph corresponding to the statements of Listing 2.9 is shown in Figure 2.5.

```
for (i = 1 ; i < P ; i++ )
   for ( j = 1 ; j < P ; j++ ) {
       A[i][j] = B[i][j] + A[i][j+1]; //S1
       B[i][j] = A[i-1][j+1] +1; //S2
   }
```

Listing 2.9 – SCoP to Illustrate Dependence Vectors

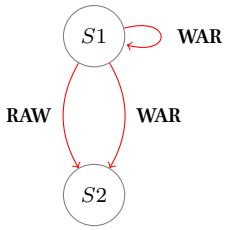


Figure 2.5 – Dependence Graph

In order to perform a dependence analysis precise enough to enable applying correct loops transformations, information about dependences among statements across iterations must be accurately expressed.

In what follows we present two popular compact representations of data dependence required for dependence analysis: dependence vectors and dependence polyhedron.

Dependence Vector

In loop dependence analysis as presented in [7], a dependence between two statements S and S' in a loop nest is characterized by: a distance vector , a direction vector and the level of dependence.

Definition 2.1.16. (*Distance vector*). A distance vector signifies the distance between the source and the destination statements of a data dependence. Suppose that statement S' depends on a statement S, and let $S'(\vec{y})$ and $S(\vec{x})$ denote a pair of their instances respectively. Then, $S'(\vec{y})$ depends on $S(\vec{x})$ such that $S(\vec{x})$ is executed before $S'(\vec{y})$ and $\theta^S(\vec{x}) \ll \theta^{S'}(\vec{y})$. Accordingly, the distance vector is defined as:

$$\vec{d} = (d_1, d_2, ..., d_n) = \vec{y} - \vec{x} \quad \text{where } n \text{ is the enclosing loop nest depth.}$$

Definition 2.1.17. (*Direction vector*). The direction vector indicates the direction of the dependence. it can be deduced from the sign of the distance vector and it can be represented by:

$\vec{\sigma} = \text{sig}(\vec{d}) = (\text{sig}(d_1), \text{sig}(d_2), ..., \text{sig}(d_n))$, such that $\forall$ element $d_i \in \{d_1, ..., d_n\}$:

$$\text{sign}(d_i) = \begin{cases} +\ or\ 1 & if\ d_i \geq 0 \\ -\ or\ -1 & if\ d_i \leq 0 \\ 0 & if\ d_i = 0 \end{cases}$$

Although direction vectors are less precise and less informative than the distance vectors, in some situations, they can be sufficient to apply effectively some loop transformations like loop parallelization and interchange.

The leading element of a vector is its first non-zero element of this vector. The sign of the leading element determines the lexicographic sign of the corresponding vector. If it is positive, then the vector is considered to be lexicographically positive. Otherwise, the vector is either negative or null.

Definition 2.1.18. (*Dependence Level*). Given the direction vector $\vec{d} = (d_1, d_2, ..., d_n)$ corresponding to a dependence of statement S' on S in a loop nest of depth n, the level of $\vec{d}$, i.e., the level at which the dependence occurs, is defined as:

$$lev(\vec{d}) = \begin{cases} l \text{ where } 1 \leq l \leq n & \text{if } \vec{d} \neq \vec{0} \text{ and } d_l > 0 \ (d_l \text{ is the leading element}) \\ n+1 & \text{if } \vec{d} = \vec{0} \end{cases}$$

The distance and direction vectors must always be lexicographically non-negative. Hence, the level is always non-negative; this allows determining if the dependence between the corresponding statements is loop-carried/loop-dependent, and, if so, helps identifying the exact loop in the nest carrying this dependence.

Definition 2.1.19. (*Loop-carried vs. Loop-independent Dependence*). The dependence of S' on S is considered to be loop-carried, i.e., carried by the l^{th} loop in the loop nest enclosing S and S' if $1 \leq l \leq n$. It occurs when two different iterations of this loop touch the same memory location. On the other hand, if $l = n+1$, i.e., $\vec{d} = \vec{0}$, then the dependence is loop independent, i.e., not carried by any loop.

For example, reconsider the Listing 2.9:

- Statement $S1$ is said to be (WAR) anti-dependent on itself with:
  - distance vector $\vec{d} = (x_1, x_2 + 1) - (x_1, x_2) = (0, 1)$
  - direction vector $\sigma = (0, +)$
  - level $lev(\vec{d}) = 2 \implies$ this dependence is carried by loop $L2$.
- Statement $S2$ is (RAW) flow dependent on $S1$ with:
  - distance vector $\vec{d} = (x_1 + 1, x_2 - 1) - (x_1, x_2) = (1, -1)$
  - direction vector $\sigma = (+, -)$
  - level $lev(\vec{d}) = 1 \implies$ this dependence is carried by loop $L1$
- Also, $S2$ is (WAR) anti-dependent on $S1$ with:
  - distance vector $\vec{d} = (x_1, x_2) - (x_1, x_2) = (0, 0)$
  - direction vector $\sigma = (0, 0)$
  - level $lev(\vec{d}) = 3 \implies$ It is a loop-independent dependence.

Dependence Polyhedron

For each edge e in the dependence graph (example in Figure 2.5), the exact conflicting dynamic instances responsible for this dependence relation between statements can be expressed as a polyhedron, called the dependence polyhedron P_e proposed by Feautrier [40] and revisited in [33, 140, 105]. Suppose that statements S and S' are the source and the target of a dependence; their dependence polyhedron is a subset of the Cartesian product of their iteration domains defined by the following constraints :

- The instances responsible for the dependence must exist, i.e., their iteration vector must belong to the iteration domain: $\vec{x} \in D^S$ and $\vec{y} \in D^{S'}$
- They must access exactly the same memory location: $f^S(\vec{x}) = f^{S'}(\vec{y})$
- The source statement instance must be executed before that of the destination: $\theta^S(\vec{x}) \ll \theta^{S'}(\vec{y})$

Then, the dependence between S and S' can be expressed using the notation $\delta(S, S', P_e)$.

Let us reconsider the example in Listing 2.9, we can represent the anti-dependence (WAR) of $S1$ on itself as a dependence polyhedron P_e containing all conflicting instances $S1((x_1, x_2))$ and $S1((y_1, y_2))$ defined by the constraints represented by the inequalities and equalities below:

$$P_e = \left\{ ((x_1, x_2), (y_1, y_2)) \;\middle|\; \begin{array}{l} 1 \leq x_1, x_2, y_1, y_2 < P \\ y_1 = x_1,\ y_2 = x_2 + 1 \\ x_1 \leq y_1 \end{array} \right\} \begin{array}{l} \rightarrow \text{instances exist} \\ \rightarrow \text{same mem. location accessed} \\ \rightarrow \text{source before target} \end{array}$$

This representation is very useful since all the dependent instances are represented in a polyhedron which can be also expressed using a matrix describing a set of affine constraints.

2.1.4 Legal Polyhedral Optimizing Transformations and Parallelization

The polyhedral model optimizes loop nests by transforming them to exhibit parallelism or to improve data locality.

As mentioned earlier, the polyhedral model applies loops transformations by reordering statements instances within these loops. Such transformations are expressed as a set of scheduling matrices. Nevertheless, these transformations and schedules must be legal respecting the semantical constraints defined by data dependences.

Definition 2.1.20. (Legality Property/Precedence Condition/Dependence Satisfaction): Given a dependence $\delta(S, S', P_e)$, the schedule expressed by θ^S and $\theta^{S'}$ is legal if and only if:

$$\forall(\vec{x}, \vec{y}) \in P_e, \theta^S(\vec{x}) \ll \theta^{S'}(\vec{y})$$

Mainly, there are two approaches to ensure the legality of polyhedral model transformations:

1. A posteriori approach that assures the legality of the transformation that is either chosen by the user or obtained semi-automatically [29, 49] based on a violation analysis [140]. This legality check is performed making use of (the inverse of) the legality property which is for $\delta(S, S', P_e)$:

$$\exists(\vec{x}, \vec{y}) \in P_e, \theta^{S'}(\vec{y}) \ll \theta^{S}(\vec{x})$$

   The set satisfying this condition represents the pairs of dependent statements instances whose order is inversed. If this set is empty, then the data dependences are preserved and the new schedule is legal. This check must be performed for all dependence polyhedra. The emptiness check is achieved by a Fourier-Motzkin elimination restricted to integer solutions [123, 106]. In case the set is not empty, i.e., the transformation is illegal, Vasilache [139] proposed a solution to correct it by loop-shifting.

2. A priori approach that automatically finds legal transformations only. This is achieved by building the space of legal transformations or schedules by taking into consideration all legality constraints corresponding to all the dependences polyhedra based on affine form of Farkas lemma [123] and Fourier-Motzkin elimination [41]. This approach is adopted by the affine scheduling algorithms[41, 42, 51, 21, 71, 72].

For instance, Listing 2.10 shows the kernel for matrix-vector product composed of two affine nested loops: $L1$, the outermost loop in the nest whose iterator is i, and $L2$, the innermost loop whose iterator is j. $L1$ and $L2$ both iterate starting a lower bound 0 to an upper bound N which is a loop parameter. There are two statements: $S1$ enclosed by $L1$ and $S2$ enclosed by $L2$ and $L1$. In this example, we exclusively focus on the true dependence of $S2$ on $S1$ ($\delta(S1, S2, P_e)$). Due to this dependence, not every transformation can be a legal transformation.

The corresponding dependence polyhedron P_e is shown to the right of this listing.

```
for ( i = 0 ; i < N ; i++ ) { //L1
    C[i]=0; //S1
    for ( j = 0; j < N ; j++ ) //L2
        C[i]+=A[i][j]*B[j]; //S2
}
```

$$P_e = \left\{ ((x_1), (y_1, y_2)) \;\middle|\; \begin{array}{l} 0 \leq x_1, y_1, y_2 < N \\ y_1 = x_1 \\ x_1 \leq y_1 \end{array} \right\}$$

Listing 2.10 – SCoP: Matrix-Vector Product

Figure 2.6 illustrates the polyhedra of the iteration domains of $S1$ and $S2$. Since $S1$ is enclosed by one loop, its iteration domain is one-dimensional, i.e., an affine segment, and since $S2$ is enclosed by two loops, its polyhedron is two-dimentional, i.e., an affine bounded plane (square in this case). These polyhedra also show the execution order of the statements instances and the data dependences between them.

The original schedules of $S1$ and $S2$, $\theta_o^{S1}(\vec{v})$ and $\theta_o^{S2}(\vec{v})$ such that $\vec{v} = (i\ j\ 1)^\intercal$ are as follows.

$$\theta_o^{S1}(\vec{v}) = \left[\begin{array}{cc|c|c} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right] \left(\begin{array}{c} i \\ j \\ \hline N \\ \hline 1 \end{array}\right) = \begin{pmatrix} 0 \\ i \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

$$\theta_o^{S2}(\vec{v}) = \left[\begin{array}{cc|c|c} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right] \left(\begin{array}{c} i \\ j \\ \hline N \\ \hline 1 \end{array}\right) = \begin{pmatrix} 0 \\ i \\ 1 \\ j \\ 0 \end{pmatrix}$$

First, the same schedule for $S1$ is kept and that of $S2$ is modified by interchanging the columns of the original iteration ordering matrix component of the scheduling matrix enabling a program transformation known as loops interchange.

The new scheduling function of $S2$ is:

$$\theta_i^{S2}(\vec{v}) = \left[\begin{array}{cc|c|c} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right] \left(\begin{array}{c} i \\ j \\ \hline N \\ \hline 1 \end{array}\right) = \begin{pmatrix} 0 \\ i \\ 1 \\ j \\ 0 \end{pmatrix}$$

Figure 2.7 shows the corresponding transformed polyhedra of the iteration domains of $S1$ and $S2$ with the new execution order and dependences of their instances. The new polyhedra reveal that loops interchange is illegal. The data dependence of $S2$ on $S1$ after rescheduling must be respected by preserving the precedence order as specified by the dependence polyhedron i.e., for every iteration of $L1$, instances of $S1$ must be executed before that of $S2$. However, some data dependences are inversed as some instances of $S2$ are executed before those of $S1$ on which they depend. Yet, other transformations are still possible like loop fission. So, second, we perform a loop fission on the original loop nest. The original scheduling function of $S1$ remains unchanged.

Loop fission can be expressed by the following new schedule of $S2$:

$$\theta_{fi}^{S2}(\vec{v}) = \left[\begin{array}{cc|c|c} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right] \left(\begin{array}{c} i \\ j \\ \hline N \\ \hline 1 \end{array}\right) = \begin{pmatrix} 0 \\ i \\ 1 \\ j \\ 0 \end{pmatrix}$$

The rescheduled polyhedra are shown in Figure 2.8. Data dependences are all respected, and, so this transformation is legal in this case.

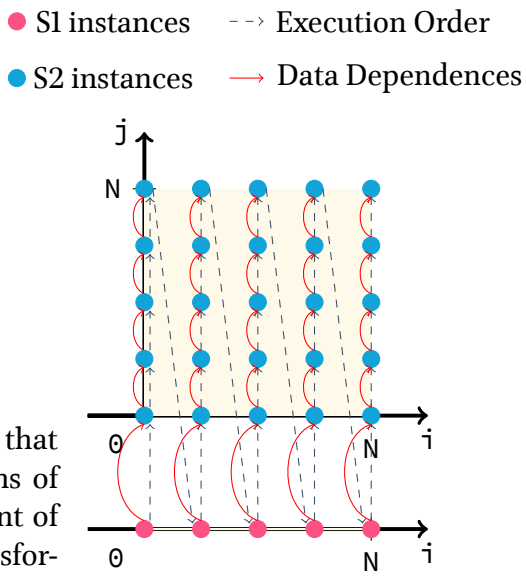


Figure 2.6 – Polyhedra

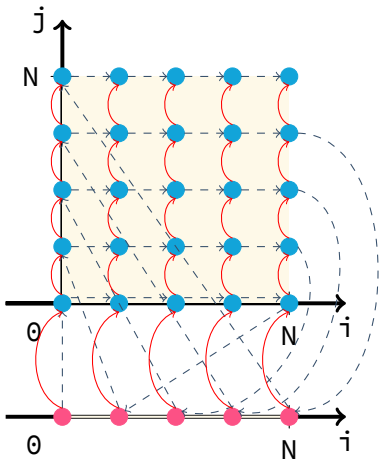


Figure 2.7 – Illegal Polyhedra

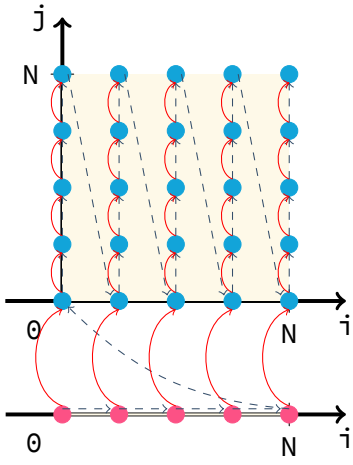


Figure 2.8 – Legal Polyhedra

Parallelism

After a legal transformation is chosen for the affine loop nest, parallelization opportunities can be furtherly investigated.

In the previous section, we defined the notions of loop independent and loop carried dependences which are significant for determining if the loops involved in the dependences can be parallel. A loop carrying no dependence can be made parallel.

This can be determined by augmenting the dependences polyhedra with an additional constraint to test if a certain loop at a certain level in the loop nest carries these dependences. If all augmented dependences polyhedra are empty, this means that this particular loop carries no dependence. Hence, it can exhibit parallelism.

```
#pragma omp parallel for
for ( i = 0 ; i < N ; i++ )
   C[i]=0; //S1
#pragma omp parallel for
for ( i = 0 ; i < N ; i++ )
   for ( j = 0; j < N ; j++ )
       C[i]+=A[i][j]*B[j]; //S2
```

Listing 2.11 – SCoP: Matrix-Vector Product Transformed and Parallelized

In the matrix-vector product example, the dependences are loop independent. $L1$ does not carry any dependence, so it can be parallelized.

In Listing 2.11, we show the matrix-vector product code after performing loop fission. $L1$ is splitted into two loops that can be parallelized using OpenMP.

The polyhedral model is greater than our presentation. More details and information about it, its powerful analysis, transformations and code generation can be found in the literature as in[11, 13, 105, 138].

2.1.5 Polyhedral Tools

In our study, we take advantage of numerous polyhedral tools and libraries that we present in what follows.

As explained in a later chapter, the fruit of our work is a tool developed based on Clang-LLVM. For that reason, our first choice for enabling polyhedral transformations was Polly.

Polly[2] [69] is a high-level loop and data-locality optimizer infrastructure for LLVM that is enabled at the level of the LLVM intermediate representation (IR). It provides analysis and classical legal loop transformations, especially loop tiling and fusion for the purpose of enhancing data-locality. Moreover, Polly is capable of detecting SIMDization opportunities and generating OpenMP parallel code.

However, to optimize more complex code parts, instead of Polly, we use Pluto (An automatic parallelizer and locality optimizer for affine loop nests)[3] [21, 20] which is a popular automatic polyhedral loop optimizer. It performs source-to-source C reliable

[2]All about Polly at https://polly.llvm.org/

[3]All about Pluto at http://pluto-compiler.sourceforge.net/

codes transformations to apply parallelism and enhance data locality simultaneously. Also, other than source codes, it can take as an input and handle a polyhedral OpenScop representation of a loop nest. Pluto is well-known for performing affine transformations that enable efficient tiling besides the other classical loop optimization, and it automatically generates parallel OpenMP codes. Pluto uses Clan, Candl and Cloog-ISL, for scanning the C code, computing dependences and generating the output code. Some limitations of Pluto were addressed and tackled in PLUTO+[2].

In our tool, we do not apply polyhedral transformations at the source level. Instead, we apply the optimizations at the IR level which Pluto cannot handle without OpenScop.

OpenScop [12] is a specification that defines a file format and data structures representing affine loop nests or SCoPs.

Clan standing for Chunky Loop ANalyzer [14] is a software/library which translates affine loop nests/SCoPs existing in high level (C, C++, Java, etc.) programs into the polyhedral representation previously defined, OpenScop. This representation may be reliably used by other polyhedral tools to perform powerful analyses and optimizations.

Candl as short for Chunky ANalyzer for Dependences in Loops[13] is a library and a software devoted to data dependence analysis. It computes the dependence graph and dependence polyhedra for a SCoP.

Cloog or Chunky Loop Generator [9] is a software and library used to generate code for scanning parametrized $\mathbb{Z}$-polyhedra.

2.1.6 Limitations

The polyhedral model offers powerful analyses and aggressive program optimizations. It is quite useful because it tackles a set of the most compute-intensive and time-consuming structures, loops.

However, the polyhedral model is still limited as classical polyhedral optimizers are exclusively effective for a particular kind of loop nests which are affine for-loop nests. On the other hand, generally, a complex program may contain more than just affine for-loops, e.g., for-loops may have unknown bounds, involve breaks, exits and function calls or include indirections and pointers, and, also, there may be while-loops, etc. In the polyhedral model, such code parts cannot be recognized as valid SCoPs, and they cannot be accurately analyzed and reliably optimized at compile-time. This is because the dynamic memory behavior of such structures cannot be precisely determined at compile-time, so is the case with the existing data dependences. Therefore, a transformation statically chosen and applied to such codes may turn to be illegal at run-time; accordingly, classical static optimizers do not optimize these codes.

Yet, there exists, Apollo [132, 84, 131, 83], an optimizer based on the polyhedral model, discussed later in this chapter, that was implemented to capture statically non-affine loop nests that exhibit affine dynamic memory behaviors compliant with the polyhedral model. Then, using such an approach, these codes can be analyzed at run-time and aggressively optimized using the polyhedral model whenever possible.

Nevertheless, loops are not the only compute-intensive structures that may exist in a program: what about recursions?

For instance, consider the recursive matrix-vector product kernel in Listing 2.12. As discussed earlier in this chapter, the iterative version of the matrix-vector product per-

fectly fits in the polyhedral model; however, the following recursive version, although it touches the memory in a similar fashion at run-time, it is out of the scope of the polyhedral model. In general, even though there exist some attempts to allow polyhedral modeling of recursive calls, which are addressed in Chapter 3, recursive structures are so different from what the polyhedral model recognizes and analyzes in a code.

```
void MatrixVectorProduct(int A[N][N], int B[N]) {
  static int row=0, column=0;
  if (row >= N)
    return;
  if(column < N){
    C[row]+=A[row][column]*B[column];
    column++;
    MatrixVectorProduct(A, B);
  }
  columns=0;  row++;
  MatrixVectorProduct(A, B);
}
```

Listing 2.12 – Recursive Matrix-Vector Product

In this thesis, we focus on introducing the recursive structures to the realm of the polyhedral model, so that they benefit from its advanced analysis and optimization powers. For this purpose, we implemented Rec2Poly, presented later in this manuscript.

2.2 Speculative Loop Optimization

Another popular approach for optimizing loop structures is to optimistically execute their iterations in parallel as if parallelism is guaranteed to respect the semantics of the original sequential code. Then, later, at run-time, the validation is taken care of. This approach is called *thread-level speculation*.

Thread-level speculation (TLS) or speculative thread-level parallelization [111, 125, 126] is a known run-time technique adopted by compilers mainly to optimize loops by parallelizing them without even knowing all the required information and the precise dependences at compile-time.

A simplified outline of the TLS system is shown in figure 2.9.

A typical TLS system is composed of two phases: the first phase is performed at compile time (statically), and the second one is performed at run-time (dynamically).

At compile time, static analysis is firstly performed to get all the information obtainable and the data dependences detectable, so that these dependences are either statically resolved or taken into consideration for the transformations. It is also possible to perform an offline profiling to gather some accessible dynamic information supposing that the execution behavior is the same at run-time. Additionally, some systems generate expressible conditionals to uncover hidden dependences later at run-time. Then, at this phase, the code blocks that must be executed sequentially and those that can be executed simultaneously are identified.

Accordingly, loops iterations are speculatively executed simultaneously on parallel threads. Some systems, at run-time, before directly parallelizing the loops, perform a

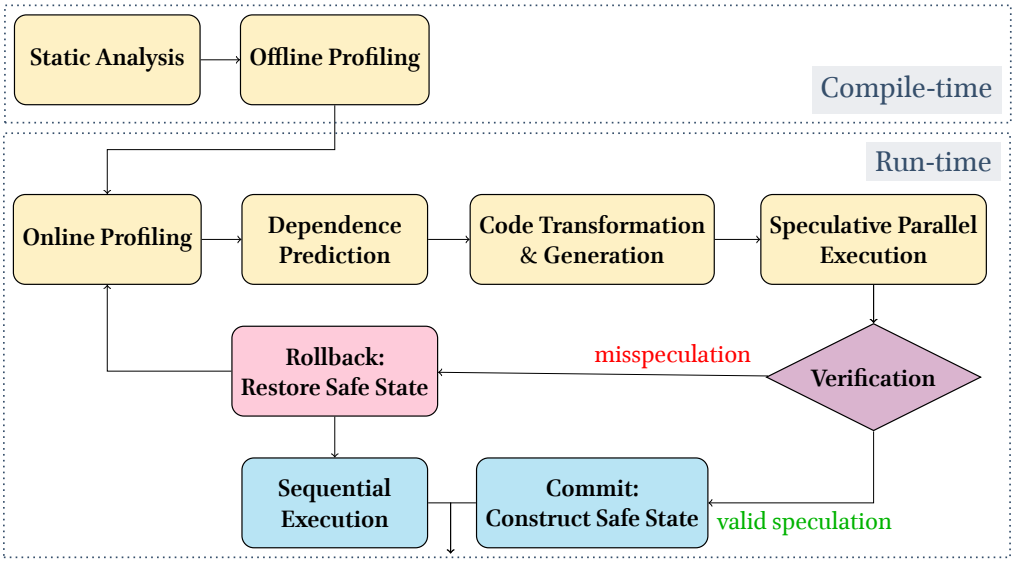


Figure 2.9 – Thread Level Speculation System

partial online profiling of the code to determine additional dynamic information and undiscovered dependences. Then, an optimizing transformation is chosen based on both the dynamic and the static information obtained. The optimization applied is basically a simple parallelization achieved by slicing the outermost loop into slices executed on parallel threads.

However, since this parallelization technique depends on either inaccurate static analysis or partial dynamic profiling, some dependences may still be ambiguous, and it is probable to encounter unsafe run-time operations. For this reason, the correctness of this parallel execution must be verified at run-time.

At some points during execution, threads are monitored and validated by ensuring that there is no data dependence violation. If their execution is valid, then the temporary states of the threads are committed saving the speculative data updates to the global memory forming the safe state.

On the other hand, if a misspeculation or misprediction is detected in particular threads, the invalid iterations are cancelled and the speculative updates are discarded. Then, a rollback and backup operations are initiated to reach an earlier (most recent) safe state instead of restarting the whole loop execution all over again. Usually, the faulty threads re-execute the invalid iterations using another valid execution order (usually sequentially) starting from the restored safe state. Also, it is possible to re-execute, in parallel again, the non-valid iterations using a new schedule/transformation based on a new online profiling.

```
while ( condition ) {
    A[i] = ...  ;
        ...
     ... = A[j] ;
}
```

Listing 2.13 – While Loop

For instance, consider the while loop code shown in the Listing 2.13. It accesses elements in array A using references i and j. There may exist data dependences across this loop iterations as there may occur conflicting accesses to the same elements of A. Traditional static compilers may not be capable of detecting all the possible existing dependences in such a loop, thus they do neither transform nor parallelize it.

In figure 2.10, we illustrate an example of a possible sequential execution of this loop iterations. Each iteration is presented by a block and the arrows between the blocks represent the control flow of these iterations. We assume that this loop iterates four times only.

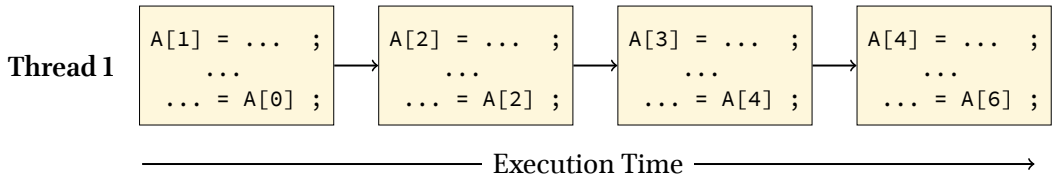


Figure 2.10 – Sequential Execution

In the original sequential execution, we notice that there is a (WAR) anti dependence occurring between the third and the fourth (last) iteration. The third iteration reads from $A[4]$, then the fourth iteration writes to the very same memory location.

A classical TLS system does parallelize such a loop without knowing that such a dependence actually occurs between the iterations. In this example, a speculative parallelization of this loop is performed by simultaneously executing its iterations, each on a separate thread. The speculative parallel execution of the while loop is shown in figure 2.11

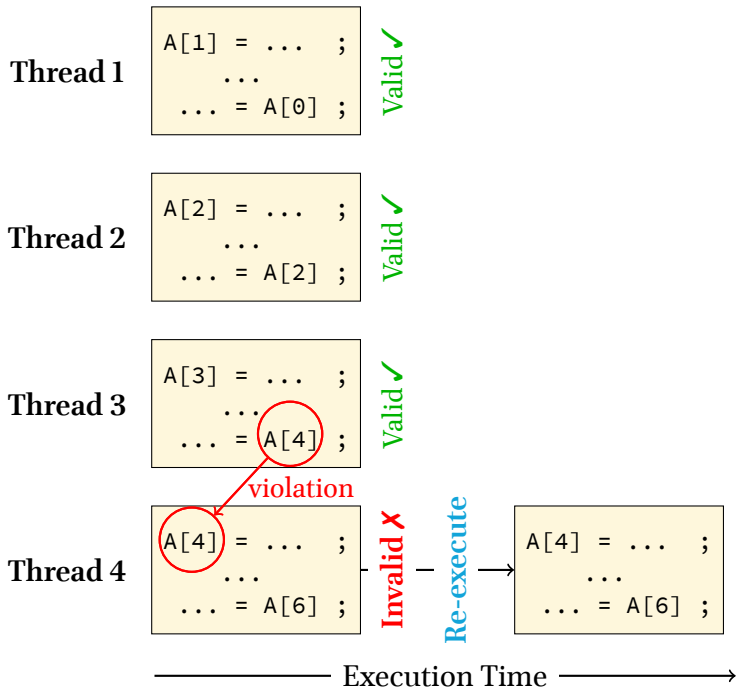


Figure 2.11 – Speculative Parallel Execution

During the speculative execution, threads 1, 2 and 3 access different locations in

memory meaning that there is no violation of any dependences. So, their parallel execution is valid, and once they successfully terminate, their updates can be committed. Yet, thread 4, executing at the same time with the rest of the threads, writes to $A[4]$ which is supposed to be read by thread 3 before this write/update. Hence, conflicting memory accesses are encountered and a misspeculation is detected; the third and the fourth iteration cannot be executed in parallel. Then, the computations done by thread 4 are cancelled, and it re-executes again after thread 3 completes its computations to ensure that it respects the existing dependence and preserves the original program semantics.

The efficiency of TLS systems depend on the misspeculations occuring during the optimized code execution; the less there are misspeculations, the better the performance of the parallelized code is.

There exist two main types of TLS systems: (1) hardware-based and (2) software-based. Hardware-based approaches utilize customized hardware that supports TLS. On the contrary, software-based systems do not require a special support from the underlying hardware. Also, in comparison with software-based systems, hardware-based systems are more performant and efficient, but they are not as evolved to process general purpose programs. Hardware-based sytems include Intel's Transactional Synchronization Extensions [148], The IBM's Blue Gene/Q Chip [55] and Rock [28], etc. As for the software-based TLS systems, there are many works including the LRPD test [111], Softspec [23], POSH [74], etc.

In general, traditional TLS systems are still not considered aggressive loop optimizers because:

- they may not improve data locality
- they may miss significant parallelization opportunities or only apply simple loop transformations before parallelization
- exhaustive misspeculations still weigh them down

Nevertheless, there exist outsanding TLS software-based systems like VMAD [57, 58] and Apollo [132, 84, 131, 83], the enhanced successor of VMAD, which made their mark in this domain because they pushed the limitations restraining TLS systems by speculatively enabling powerful polyhedral optimizations.

In this thesis, we push the limits of TLS systems even further to also cover non-loop structures, recursions in particular. The use of TLS in the realm of recursions is original and new which distinguishes our work more. In a later chapter, we present in details our tool, Rec2Poly, which adopts a speculative verification technique known as the inspector-executor strategy.

In what follows in this section, we introduce the inspector-executor mechanism, and then we choose to present, out of the speculative loop optimizers, Apollo that was our inspiration to combine both of the speculative approach and the polyhedral model.

2.2.1 Inspector-Executor Mechanism

The inspector-executor mechanism is a well-known technique in the domain of compilation and optimization of programs used for guiding code transformations. It has been

seen and used in earlier works that are exclusively dedicated to loops speculative parallelization [110, 36].

This technique involves two main processes:

1. **Inspector**

2. **Executor**

Both of the inspector and the executor can be automatically constructed from the original loops to optimize. Whenever it is possible, the original loop is broken up into two loops where:

- The first is executed by the inspector process: it is a light version of the loop that helps monitoring the run-time execution and performing light computations to guide loop transformations or simply verify their correctness. It calculates the memory addresses that are supposed to be accessed, so it can precisely compute data dependences and obtain loop upper bounds and other loop-related information. This information is required to ensure a valid execution of the second loop. Note that the inspector may also be optimized and parallelized to run as fast as possible for a lowest possible time overhead.

- The second loop is executed by the executor process: it is a loop that actually performs the existing memory accesses and computations. If possible, this loop can be even reliably optimized and parallelized based on the information collected by the inspector.

There is an interesting recent study by Strout et. al. [129] that reviews the history and current state-of-the art for inspector–executor strategies. Also, it presents how the Sparse Polyhedral Framework (SPF) enables the composition of inspector–executor transformations, etc. SPF [130] is a framework that specifies run-time reordering transformations and algorithms for generating efficient inspector-executor code automatically which permits implementing such transformations. Their work optimizes applications that manipulate sparse data structures containing memory reference patterns that are unknown at compile time due to indirect access.

Based on the state-of-the-art by Strout et. al. [129], the inspector–executor approach was firstly introduced by Saltz et. al. [89, 120, 103].

The inspector was used to collect run-time information used by the executor in order to guide distributed memory parallelization. In [89], the inspector's role was to collect dependences information. In later works [103, 68, 147, 121], inspectors became capable of collecting more information, e.g., the processors occupied by data elements, communication schedules, loop iteration partitions, etc; then, executors could gather non-local data, perform computations and, whenever necessary, scatter the results.

Moreover, the inspector-executor paradigm was used to guide shared memory parallelization by combining static and runtime dependence testing; for example, the LPRD test [112] speculatively detected parallelization, reduction, and privatization opportunities using a compressed bit map marking dependences at runtime. Furthermore, the inspector-executor strategy was used to allow partial parallelization, i.e., parallelization

in presence of dependences, such that the inspector detected partial parallelism by computing all the dependences for a loop and placing iterations into irregular wavefronts [109, 141].

Besides parallelization, optimizations for improving data locality like improving data reuse, reorganizing computations and performing data and iteration reordering transformations, especially for sparse codes, also relied on the inspector–executor paradigm. In this area, there existed many studies, of which we mention [38, 90].

Additionally, many studies combined available inspector-executor optimizations and parallelizations; for instance, distributed memory parallelization inspector–executor transformation with affine transformations enabling vector/polyhedral optimizations [113].

However, a sequential inspector was considered respectively costly in an optimized code. For this reason, there were conducted many research studies that parallelized the inspector code [120, 70, 110]. Other studies investigated where an inspector code could be placed to minimize the number of inspector executions [38, 121], developed partial redundancy elimination [121, 3] and developed inter-procedural anaylses to discover effective points where the inspector could be called [3], etc. Such inspector optimization and parallelization approaches are still relevant and can still be adopted to generate efficient inspector-executor code.

Note that a remarkable work that uses the inspector-executor strategy is presented next in this section which is Apollo [132, 84, 131, 83].

Rec2poly, the recursion optimizer introduced in this thesis in Chapter 4, uses this mechanism for the first time to guide recursion-to-loop transformations instead of loop-to-loop transformations. Also, we investigate many inspector optimizations in Rec2Poly which are explained later in Chapter 4.

2.2.2 Speculative Polyhedral Optimization with Apollo

Apollo[4] short for the Automatic speculative POLyhedral Loop Optimizer [132, 84, 131, 83] is a Clang-LLVM based compiler framework dedicated to automatic speculative polyhedral optimization and parallelization of loop nests composed of any kind of loops (for, while...) existing in C/C++ programs. Its objective is to dynamically optimize loop nests that exhibit a run-time behavior that is compliant with the polyhedral model. So, Apollo extends the applicability of the polyhedral model to loop nests that do not have an affine structure at compile-time.

It makes use of a an online profiling phase to build a speculative prediction model that describes the dynamic memory behavior of the loop nest. If the behavior is affine, then it chooses and applies efficient polyhedral transformation, optimizations and parallelizations at run-time.

Furthermore, Apollo needs to verify that the speculative loop transformation is still valid at run-time, so it involves a verification strategy that is partially based on the inspector-executor paradigm.

As mentioned earlier, Rec2Poly extends the Apollo approach to handle recursive codes. Like Apollo, Rec2Poly makes use of a profiling technique, but to discover an affine behavior for recursive functions. Rec2Poly goes even further in the profiling phase by discov-

[4]All about Apollo at `https://webpages.gitlabpages.inria.fr/apollo`

ering the control behavior in addition to the memory behavior. The control behavior is required in our case since we are transforming recursive functions to a totally different structure, loops. Finally, Rec2Poly applies the inspector-executor mechanism to verify the recursive code against a loop model; For this reason, Rec2Poly needs to verify both of the control and memory behaviors, whereas Apollo uses this technique to verify only the memory behavior of loop nests against that of the predictive loop model.

2.3 Trace Modeling as Polyhedral Loops with NLR

The Nested Loop Recognition algorithm known as NLR [65] is an algorithm that takes as input a sequence of integer tuples and constructs as an output sequences of loop nests that produce the same original trace when executed.

There are many applications of the NLR algorithm including:

1. program behavior modeling for any measured quantity e.g., memory accesses, etc.
2. execution trace compressing
3. value prediction, i.e, extrapolating loops under construction (while reading input) to predict incoming values.

To illustrate how NLR works, let us reconsider the iterative matrix-vector product code in Listing 2.14 with statements printing the memory addresses accessed at run-time. We added a printing statement in the body of the first loop in the code to print the address of `C[i]`, and another one in the body of the second one to print the addresses of `A[i][j]` and `B[j]`. If this code is executed, a sequence of integer tuples will be printed. Then, if this sequence is given as an input to NLR, the output will be a model made up of a sequence of two affine loop nests as displayed in Figure 2.12. These loop nests include affine expressions in terms of the surrounding loop indices. The first affine expression corresponds to the accesses to Array C. The three affine expressions in the innermost loop correspond to the accesses to Arrays A, B and C respectively. Such a model indicates that the corresponding program has a polyhedral memory behavior.

The loops in the matrix-vector product code are already affine, and so the memory addresses are abviously accessed in a polyhedral-compliant way which is also reflected in the corresponding NLR model.

```
for( i = 0 ; i < N ; i++ ) {
  C[i]=0;
  printf("%ld\n",(long)(&C[i]));

  for( j = 0 ; j < N ; j++ ) {
    C[i] += A[i][j]*B[j];
    printf("%ld\n%ld\n%ld\n",(long)(&A[i][j]),(long)(&B[j]),(long)(&C[i]));
  }
}
```

Listing 2.14 – Matrix-Vector Product Printing Memory Addresses Accessed

```
for i0 = 0 to 9
  val 140729968340864 + 4*i0
  for i1 = 0 to 9
    val 140729968340912 + 40*i0 + 4*i1
    ,140729968340816 + 4*i1
    , 140729968340864 + 4*i0
```

Figure 2.12 – NLR model Example

Nevertheless, as we mentioned before, not all codes initially include affine loops, and there are codes that do not involve any loops at all. So, making use of the NLR algorithm in order to detect an affine memory behavior in such codes helps enabling advanced optimizations for them.

In this thesis, in Rec2Poly, we make use of this algorithm which is explained later in Chapter 4. We use it to model the behavior of recursive codes and discover the possibility to transform them to affine loops and optimize them accordingly using the polyhedral model.

Moreover, in this thesis, we present an extended version of the NLR algorithm that captures *partially* affine behaviors which is also explained in Chapter 4.

To sum up, in this chapter, we have presented the approaches and techniques that we have relied on to implement the recursion optimizer and parallelizer, the Rec2Poly framework. In the next chapter, we explain all about recursions and review the state-of-the-art of existing optimization and parallelization techniques dedicated to recursive codes.

Chapter 3

State of the Art

"To iterate is human, to recurse divine."
— L. Peter Deutsch

"Recursion is the root of computation since it trades description for time."
— Alan J. Perlis

In the previous chapter, we discussed advanced automatic optimization techniques dedicated to iterative codes of which we take advantage in this thesis to optimize recursive codes. This is interesting since, unlike recursive structures, iterative structures have always benefited from powerful optimizations. Recursion and iteration are both fundamental concepts in computer science that involve performing computations repeatedly. They are so similar meaning that it is promising to apply efficient optimization techniques initially dedicated to iterative loop structures on recursive structures; yet, they are so different making this process complicated and not as straightforward as it may seem. Although they can be alternatively used in imperative programming languages, a recursive approach is usually so descriptive and sophisticated, but an iterative approach is usually more efficient even without further optimizations.

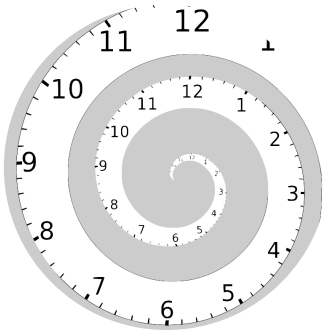


Figure 3.1 – Recursion Example: Clock Spiral Droste Effect

In this chapter, we introduce the target of our work, recursions and recursive programs, and we present the state of the art of this thesis. In the first section, we discuss some generalities on recursions and recursive functions. We show how a recursive algorithm is designed based on induction, how a recursive program is implemented and executed. Also, we present the main types of recursion, and, then, its computational complexity. On the one hand, in the second section, we address the latest works that optimize recursions without manipulating the recursive structures. These optimizations are mainly achieved through task parallelism or through polyhedral modeling. On the other hand, in the last section, we discuss another approach for handling recursive functions which involves transforming them into loops.

3.1 Generality on Recursions

Figure 3.2 – Trees

Recursion is no strange phenomenon for us. A recursion occurs when a thing or entity is defined in terms of smaller instances of itself. Even though we may not recognize it, recursive entities also known as fractals are part and parcel of our environment and nature, e.g., trees, snail shells, rivers, broccoli, fire, lungs, blood vessels, etc. For instance, let us take a closer look at trees (see Figure 3.2). A tree, as a whole, can be seen as a stem of which comes out branches. Similarly, each of these branches can be seen as a smaller stem of which grow out smaller branches, and so on; this recurrence ends by flowers blossoming or leaves blooming at the end of the tiniest branches.

Moreover, as recursion is part of our nature, it can be, correspondingly, found in many disciplines.

In art, droste effect shown in Figure 3.1 and Matryoshka doll, for instance, are considered recursive. In literature, there exists linguistic recursion; as an example, consider the recursive sentence below retrieved from the Jeeves comic novel, "Thank you, Jeeves" [146]:

"He can take a letter from you to her and then one from her to you and then one from you to her and then one from her to you and then one from you to her and then one..."

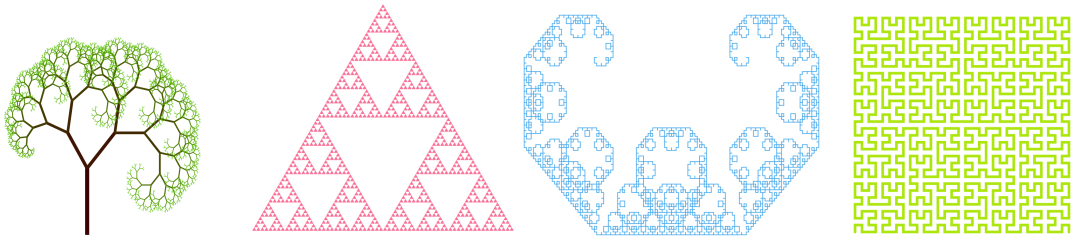
In architecture and engineering, some structures may actually be recursive, e.g., staircases and roads; a highway road usually branches to smaller roads, arterial roads, which also branch to smaller collector roads that fork to narrower local streets leading to buildings, parks, etc.

Additionally and most importantly, recursion is a well-known concept in mathematics and sciences, especially in computer science which is our main emphasis in this chapter. The concept of recursion in mathematics is strongly related to that in computer science.

In mathematics, recursion is witnessed in fractal geometry where geometric fractal patterns can be found in nature, and they are defined as rough or fragmented geometric shapes that can be decomposed into parts such that (almost) each of these parts is a reduced-size copy of the whole" [82]. Fractals can be generated by solving simple equations repeatedly.

Also, similar to fractals, there are the finite subdivision rules which involve dividing recursively a two-dimensional object, particularly a polygon, into smaller pieces. They can be seen as a generalization of fractals because instead of repeating exactly the same pattern over and over again, they allow slight variations in each subdivision. In comparison with fractals, subdivision rules make possible generating a more complex, more real structure while maintaining an elegant fractal style [27].

Visual images or models of fractals and subdivision rules, that may even model nat-



Fractal Canopy Tree | Sierpiński Triangle | Levy Curve | Hilbert Maze

Figure 3.3 – Fractals

ural entities, are of a significant use in biology, physics, art, etc. and they can be generated using computer graphics applications that perform the computations either iteratively (using loops) or recursively (using recursive codes). Some examples of fractals are the canopy tree, Sierpiński triangle, Levy curve and Hilbert maze displayed in Figure 3.3. These fractals, were automatically generated and drawn using Online Fractal Tools [96].

Furthermore, recursion appears in more abstract concepts, in mathematical formulas and definitions. Objects that can be recursively defined include factorials, Fibonacci sequence, cantor ternary set, etc. A recursive definition of an element in such sequences is expressed in terms of other, probably previous, elements in these sequences. Also, a recursive definition of a function defines values of this function for some input n in terms of the values of the same function for smaller inputs. For instance, let us consider the factorial function originally defined as:

$$Factorial(n) = n! = 1 \times 2 \times ... \times (n-1) \times n \qquad \forall n \geq 0$$

Similarly, if $n > 0$, the factorial for $n-1$ is defined as:

$$Factorial(n-1) = (n-1)! = 1 \times 2 \times ... \times (n-1)$$

We observe that the factorial of the natural number n can be expressed as the product of n and the value of the factorial given the smaller argument $n-1$.

When a function can be expressed in terms of itself for smaller input/argument, then it can be redefined using a recursive definition. A recursive definition constitutes two types of expressions, the recursive case and the base case. The recursive case includes all recursive expressions. In the factorial example, there is one recursive case which is:

$$Factorial(n) = (n-1)! \times n = factorial(n-1) \times n$$

As for the base case, it is the case when the output of the function can be trivially obtained without the need of other output values of the function. In case of the factorial function, the most trivial case is when the input is zero, as $Factorial(0) = 0! = 1$. Accordingly, the factorial function can be redefined recursively as follows:

$$Factorial(n) = \begin{cases} 1 & \text{if } \ n = 0, \\ Factorial(n-1) \times n & \text{if } \ n > 0 \end{cases}$$

Another example is the Fibonacci sequence in which the n^{th} term s_n is defined using the two preceding terms s_{n-1} and s_{n-2} such that $s_0 = 0$ and $s_1 = s_2 = 1$. This sequence can be defined using a recursive function of n that returns its n^{th} term as follows:

$$Fibonacci(n) = \begin{cases} 0 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ Fibonacci(n-1) + Fibonacci(n-2) & \text{if } n > 2 \end{cases}$$

Such recursive definitions even provide algorithms, recursive algorithms, allowing computers to perform such computations in a recursive fashion.

In computer science and programming, there are two main programming paradigms: declarative/functional programming and imperative programming. On the one hand, the declarative paradigm focuses on what the code actually achieves by defining the program's logic without explicitly describing the control flow. Examples of declarative languages include Haskell, Scala, Swift, etc. Executing repetitive computations are only possible in declarative programming through recursions.

On the other hand, the imperative paradigm, focuses on how an algorithm or a program works such that the code explicitly describes the control flow and the instructions that change the program's state. Imperative programming languages include C, C++, Java, etc. Executing instructions repeatedly in such languages is accomplished by two approaches: iteration and recursion. Note that this thesis is within the compass of the imperative programming paradigm, and it is particularly dedicated to C/C++ programs.

Iteration involves executing a sequence of instructions over and over again within a looping structure (for loop, while loop, do loop, etc.). In contrast, recursion involves successively decomposing complex problems into smaller and simpler subproblems that are easier to express, compute, code and solve. Accordingly, the solutions of such problems rely on smaller and simpler instances of their own. Recursive programs are implemented using recursive structures (functions). A recursive approach is straightforward to adopt and implement when:

- the initial problem is decomposable into subproblems
- as subproblems are successively broken down into smaller ones, they must eventually become so simple, so they can be solved without further subdivision
- the solutions of the subproblems must be combinable to produce at the end the solution of the original problem.

Although such kind of problems can be usually solved by various, even iterative, algorithms, using a recursive approach whenever possible improves and beautifies the description and eases the expression of computations especially of the problems that are generic to parameters like search depths or problem dimensions.

However, recursive programs, like iterative programs, are compute-intensive because they often involve performing a lot of computations repeatedly, and they may scan huge data structures such as graphs, trees and matrices; this is why recursive structures and iterative structures are interesting targets for optimization.

This section is mainly based on readings from the books "Thinking Recursively" [115] by Eric S. Roberts and "Introduction to Recursive Programming" [116] by Maneul Rubio-Sanchez.

In the rest of this section, our emphasis is on recursions and recursive algorithms and codes. We start by explaining briefly how a recursive algorithm can be designed and proved. Then, we present recursive functions structures in programs and their execution.

Also, we present the different types of recursions. Finally, we discuss run-time analysis of programs, recursive programs in particular.

3.1.1 Recursive Algorithms Design

A computational problem is defined by its inputs, outputs/solutions and the statements describing the relationship between them; an instance is a specific set of inputs values enabling computing the solution to this problem. However, an algorithm describes how to solve a problem by a finite set of instructions and computations given some input. Usually, there exist various algorithms capable of solving a certain problem; yet, a recursive algorithm can be more descriptive, more elegant.

Designing and implementing a recursive algorithm and program for a problem requires that we understand the latter and provide our own definition for it specifying: the base cases and the recursive cases (similar to what is seen in the recursive mathematical definitions). In this regard, one must determine the problem's size, define the base cases, decompose the problem and define the recursive cases accordingly based on induction.

Problem's Size Determination

The problem's size can be understood as a mathematical expression involving the input parameters or other factors that define the problem's complexity in terms of the number of operations needed by the algorithm to be designed to solve this problem. The size may depend on one or more input parameters either directly, e.g., the size of the factorial computation problem given n is n, or indirectly as a function of these parameters, e.g., the size of the problem of computing the sum of the digits composing an input n is the number of these digits, or the size of a problem handling multidimensional input parameters (arrays, lists, etc.) may be the length/size of these parameters, etc.

However, the size is a property of the problem and not the algorithm, so it does not necessarily determine the exact number of operations to perform to solve the problem. On the other hand, the problem's size actually helps understanding when the base case is reached and how to reduce the size of the problem and decompose it. For instance, adding the elements of a matrix can be implemented recursively. Given a square $n \times n$ matrix containing n^2 elements, the solution requires $n^2 - 1$ (function of n) additions in total. One may think that the size of this problem is n^2, but it is only n; for example, it is sufficient to decrease the size of the problem by decreasing n only. On the other hand, given an $n \times m$ dimensional matrix, the problem depends on two parameters n and m; its size can be decreased by either decreasing n or m, so the size in this case is nm.

Also, the size may be defined in several ways such that the algorithm may differ depending on the size considered. When the size is determined, the base cases of the problem can be defined and the problem can be decomposed.

Base Cases Definition

As explained earlier, base cases are instances of the problems that can be solved without using recursion, i.e., without making use of further instances of this problem. In base cases, results can be obtained trivially, sometimes even without performing any compu-

tations. There must be at least one base case for the problem, else we will have an infinite non-terminating recursion that may lead to a bug.

Problem Decomposition

Then, whenever possible, the problem is decomposed into self-similar subproblems. This can be achieved by successively simplifying the problem or reducing its size by considering smaller and simpler instances closer to the base cases. Not only does the problem decomposition result in self-similar subproblems, but also it may bring about additional different (non-self-similar) subproblems. The solutions of these subproblems are then used to establish the recursive cases. A problem may even be decomposed in various ways, e.g., by either decrementing the size by one or dividing it by two, etc.

Recursive Cases Definition

Based on the problem decomposition chosen and assuming that the solutions of the subproblems are readily available by relying on induction, recursive cases can be defined to obtain the full solution of the original program. Recursive cases are derived by determining how we can modify, combine or extend the sub-solutions to arrive at the complete solution to the original complex problem.

In what follows, we briefly revise the concept of mathematical proofs by induction which is considered as the parallel of recursive thinking in mathematics and fundamental to the design and the proof of correctness of recursive algorithms.

Induction

It is a proof technique in mathematical logic used to show that a particular statement is true. In connection with recursion, there are mathematical and structural inductions depending on the recursive problem and the data structures that we are dealing with. Mathematical induction is a simpler proof method handling formulas in terms of some natural number n. It proves that this statement holds for all possible values of n. A proof by such an induction technique involves the following steps:

1. Base case or basis: It proves that the statement is true for the smallest value of n.

2. Inductive step: In this step, the inductive hypothesis is developed which is the assumption that the formula is true for a general value of n. Then, relying on this assumption, prove that the formula also holds for $n + 1$.

This method can be extended to, structural induction, to prove statements about more general and complex structures, e.g., trees, lists.

Accordingly, in both inductive and recursive techniques one must find: a set of simple/base cases for which the proof or the computation can be easily handled, and an appropriate rule/pattern that can be applied repeatedly til a complete solution is acquired. On the one hand, the inductive process starts with the base case, and then considers the more general larger cases based on an assumption a.k.a the inductive hypothesis. On the other hand, a recursive technique processes in the opposite direction; the recursive approach begins with the complex cases and the rule successively reduces the complexity

of the problem until only simple cases are left. Similarly, as there is the induction hypothesis in induction, there is the notion called "leap of faith" in recursive programming.

The recursive "leap of faith" refers to the assumption that the recursive code to implement will also work correctly for the subproblems even though the details of the implementation are not actually seen. Therefore, one can construct recursive cases correctly based on the "leap of faith". Also, while we need to solve the different problems resulting from the decomposition, we will not need to solve the self-similar subproblems since we assume that their solutions are already available. A recursive algorithm is completed by these recursive cases in addition to the base cases which are also correct and this is how a correct recursive algorithm can be constructed.

Finally, based on these steps and concepts, a recursive algorithm can be developed and implemented in a recursive program using recursive structures a.k.a recursive functions; then, of course, it can be executed and tested.

As an example for designing a recursive algorithm, besides the recursive algorithm that can be obviously retrieved from the recursive definition for the factorial and the Fibonacci functions already shown, we show next how a recursive algorithm can be developed to solve the matrix-matrix multiplication problem; for simplicity, we eliminate a dimension from this problem, and we choose to discuss a special case of the matrix multiplication problem, the matrix-vector multiplication ($A.\vec{v}$) of matrix A of size $n \times m$ and vector $\vec{v}$ of size m.

$$A.\vec{v} = \begin{bmatrix} A_{1,1} & \dots & \dots & A_{1,m} \\ \vdots & \ddots & \iddots & \vdots \\ \vdots & \iddots & \ddots & \vdots \\ A_{n,1} & \dots & \dots & A_{n,m} \end{bmatrix} . \begin{bmatrix} v_1 \\ \vdots \\ \vdots \\ v_m \end{bmatrix} = \begin{bmatrix} A_{1,1}.v_1 + \dots + A_{1,m}.v_m \\ \vdots \\ \vdots \\ A_{n,1}.v_1 + \dots + A_{2,m}.v_m \end{bmatrix}$$

The solution of this product is a vector of size n such that the entry on its i^{th} row ($i \leq n$) is obtained by: $\sum_{j=1}^{m} A_{i,j}.v_j$

The size of the matrix-vector multiplication problem depends on the two dimensions: m and n.

There exist many options for defining the base cases and decomposing this problem, and, so, there exist many algorithms to solve it.

In some algorithms, the trivial multiplication occurs between a 1×1 matrix and a vector of size 1 (or scalars multiplication), i.e., the base case is when the dimensions $n = m = 1$. Others may also take into consideration the cases when there are empty matrices and vectors, i.e., $n = m = 0$.

In such algorithms for instance, the solution can be obtained by dividing the matrices and vectors dimensions by two. The row dimension of the matrix is divided by two decomposing the matrix into two block matrices if the number of its rows is greater than that of its columns, and the column dimension of the vector and the matrix is divided by two, i.e., decomposing both the matrix and the vector into two parts otherwise. The decomposition of the matrix-vector product can be illustrated as follows if we suppose that $\frac{n}{2} < m \leq n$, so that the rows are partitioned firstly and the columns secondly.

$$A.\vec{v} = \left[\begin{array}{ccc|c} A_{1,1} & \dots & \dots & A_{1,m} \\ \vdots & \ddots & \iddots & \vdots \\ \hline \vdots & \iddots & \ddots & \vdots \\ A_{n,1} & \dots & \dots & A_{n,m} \end{array}\right] . \begin{bmatrix} v_1 \\ \vdots \\ \vdots \\ v_m \end{bmatrix}$$

$$A.\vec{v} = \left[\begin{array}{ccc|c} A_{1,1} & \dots & \dots & A_{1,m} \\ \vdots & \ddots & \iddots & \vdots \\ \hline \vdots & \iddots & \ddots & \vdots \\ A_{n,1} & \dots & \dots & A_{n,m} \end{array}\right] . \left[\begin{array}{c} v_1 \\ \vdots \\ \hline \vdots \\ v_m \end{array}\right]$$

This decomposition is not performed once only; instead, in every subproblem, the sub-matrices and sub-vectors are decomposed again and again til the base case is reached. If we consider the base cases condition to be $n = m = 1$, so the smallest sub-problem to be handled is the multiplication between one element of the matrix and one element of the vector; their product is just a part of the solution of an element in the resulting vector. Accordingly, the solution involves separately computing simpler products of the corresponding smaller matrices and vectors and then adding the entries of output matrices of the sub-solutions to build the full output matrix/vector. This algorithm may involve other inputs than the dimensions of the matrices and vectors that help keeping track of the decomposition.

Another algorithm is also possible by thinking of the base cases the other way around and decomposing the problem accordingly. It involves as inputs, in addition to the matrix and vector dimensions n and m, indices i and j corresponding to the matrix A's rows and columns respectively. A base case occurs when the multiplication is computed for all the n rows and the m columns of A (or the m rows of $\vec{v}$), i.e., $i = n$ and $j = m$. In this approach, the decomposition is achieved by considering smaller parts of A by successively ignoring one more row at a time. Then, the i^{th} sub-problem handles the block of A starting from the i^{th} row til the n^{th} row included. The base case is reached when the matrix has no rows left to multiply, i.e., $i = n$.

$$A.\vec{v} = \left[\begin{array}{c|ccc} A_{1,1} & \dots & \dots & A_{1,m} \\ \hline \vdots & \ddots & \iddots & \vdots \\ \vdots & \iddots & \ddots & \vdots \\ A_{n,1} & \dots & \dots & A_{n,m} \end{array}\right] . \begin{bmatrix} v_1 \\ \vdots \\ \vdots \\ v_m \end{bmatrix}$$

Furthermore, each "submatrix" and the vector, in the i^{th} subproblem, can be broken successively into smaller parts by considering one less column of A and one less row of the vector $\vec{v}$ at a time. In the j^{th} subsubproblem, the part of the matrix A considered includes all the rows from the i^{th} til the n^{th} row and all the columns from the j^{th} til the m^{th} column, and the part of the vector $\vec{v}$ handled starts from its j^{th} row. Hence, the second base case is reached when all columns entries of the "submatrix" are multiplied with those of the vector, i.e., $j = m$.

$$A.\vec{v} = \left[\begin{array}{c|ccc} A_{1,1} & \dots & \dots & A_{1,m} \\ \hline \vdots & \ddots & \iddots & \vdots \\ \vdots & \iddots & \ddots & \vdots \\ A_{n,1} & \dots & \dots & A_{n,m} \end{array}\right] . \left[\begin{array}{c} v_1 \\ \hline \vdots \\ \vdots \\ v_m \end{array}\right]$$

The recursive cases involve performing the multiplication only between the entry at the first row and the first column of the submatrices of A and the first entry of the subvector of $\vec{v}$ in a subproblem. In other words, in the recursive case, the product of $A_{i,j}$ and v_j is computed and the result is added to the corresponding entry in the output vector before any further decomposition. Note that, before a decomposition, i or j must be incremented accordingly. This approach is interesting since it mimics the traditional matrix-vector iterative algorithm using two nested loops.

Similar recursive approaches can be implemented for the general matrix-matrix multiplication including one more base or recursive cases for the extra matrix dimension.

3.1.2 Recursive Codes: Implementation and Execution

Once the recursive algorithm is designed, it can be translated into a recursive code. A recursive code consists of recursive functions. A recursive function is made up of blocks of instructions for performing computations, control structures, particularly conditional "if" statements, and function calls. In a recursive function, the recursive and base cases are distinguished using conditional statements. The corresponding recursive definitions are implemented using recursive function calls and other instructions required to develop the complete solution. In contrast, the base cases blocks, executed when the stop condition holds, do not involve recursive function calls at all; even, they may not involve any computation at all. In imperative languages, recursive functions can also encompass loop control structures and vice versa which is not an option in other programming paradigms.

The analogy between the recursive definition and the recursive functions in programming languages is evident and straightforward.

For instance, consider the C recursive functions in Listings 3.1, 3.2, 3.3 and 3.4 implementing the recursive factorial function, the recursive Fibonacci function and the recursive matrix-vector multiplication (both first and second approaches discussed earlier) respectively. Listing 3.5 is another implementation of the second recursive approach solving matrix vector product inspired by the matrix multiplication implementation in [86].

```
int Factorial( int n )
{
    if (n == 0) //base case
        return 1;
    //recursive case
    return n * Factorial(n - 1);
}
```

Listing 3.1 – Recursive Factorial C Function

```
int Fibonacci( int n )
{
   if (n <= 1)
      return n;

   return Fibonacci(n-1) + Fibonacci(n-2);
}
```

Listing 3.2 – Recursive Fibonacci C Function

```
void MatrixVectorProduct( int ** A , int * B , int * C , int n , int m
    , int base_r ,int base_c )
{
    if ( m<=1 && n<=1 ) //base case
        C[base_r]+=A[base_r][base_c]*B[base_c];
    //recursive cases
    else if ( n>m ) { //split rows
      MatrixVectorProduct(A, B, C, n/2, m, base_r, base_c);
      MatrixVectorProduct(A, B, C, n - n/2, m, base_r + n/2, base_c);
    }
    else { //split columns
      MatrixVectorProduct(A, B, C, n, m/2, base_r, base_c);
      MatrixVectorProduct(A, B, C, n, m-m/2, base_r, base_c + m/2);
    }
}
```

Listing 3.3 – Recursive Matrix-Vector Product C Function : First Version

```
void MatrixVectorProduct ( int **A , int *B , int *C , int i, int j )
{
    if ( i >= ROWS ) // all rows are handled
        return;
    if ( j < COLUMNS ) {// not all columns are handled
      C[i]+=A[i][j]*B[j];
      MatrixVectorProduct(A,B,C,i,j+1);
    }
    else // all columns at some row are handled
      MatrixVectorProduct(A,B,C,i+1,0);
}
```

Listing 3.4 – Recursive Matrix-Vector Product C Function : Second Version

```
void MatrixVectorProduct ( int **A , int *B , int *C )
{
    static int i=0, j=0;
    if ( i >= ROWS )
        return;
    if ( j < COLUMNS ){
        C[i]+=A[i][j]*B[j];
        j++;
        MatrixVectorProduct( A , B , C );
    }
    i++;    j=0;
    MatrixVectorProduct( A , B , C );
 }
```

Listing 3.5 – Recursive Matrix-Vector Product C Function : Third Version

A recursion in programming takes place when a recursive function reaches itself through a sequence of function calls, recursive calls. A function may even invoke itself in its own body using different function parameters, i.e., it invokes simpler instances of itself that require less recursive calls, less operations until reaching the base case.

Recursions Control Flow

The control flow of recursions majorly depends on the parameters values and the flow control instructions enveloped in the involved recursive functions including conditionals (if, if-else), loops (for, while, etc.), (recursive and non recursive) function calls and returns.

Normally, flow control instructions allow the program control to jump to an instruction that may not textually be the next instruction in the program. For instance, loop structures may execute at least once or not at all. If the looping condition does not hold, the instructions of the loop will be skipped and there will be a jump to the first instruction outside the loop body in the parent function; else, the instructions of the loop are executed, and at the last instruction within the loop body, there will be a jump back to the loop condition.

On the other hand, a function call pauses the execution of the current function, leads to a jump to the first instruction in the function called; the function called executes, and when it terminates and returns, the control jumps back to the calling function body whose execution is resumed starting from the next instruction right after the returned function invocation.

So, in recursion, when a function calls itself, the control will jump back to the function's first instruction, and it re-executes all over again given different arguments. As long as the recursive call is reached, this scenario is repeated over and over again. In the last called function, when the stop condition is satisfied, i.e., the base case is reached, the function terminates and the control jumps back to the calling function and so on until the last active recursive function terminates. Yet, it is not just about "jumps" in function calls; function calls, including recursive calls, involve accessing "the stack".

Recursions Stack Up

Usually, during a program's execution, with every function call, a stack frame composed of the information about the activated or invoked function (e.g., function parameters, local variables information, return address, etc.) is pushed to the top of the so-called the program/call/control stack or "the stack" for short; the stack is a data structure that is used for saving records about active functions in a program and for keeping track of their execution order (last called function, first returns). When a function terminates its execution, its associated stack frame is popped and the memory is de-allocated to be used for other function calls. Then, if the stack frame is not empty, the function whose stack frame is now on the top of the stack, which is the calling function, continues its execution.

The stack is useful for implementing sequences of nested function calls and, of course, recursions.

A recursion is implemented through the program's stack. As long as the base case is not reached, a recursive function repeatedly calls itself and adds as many stack frames as it takes to the control stack. When a base case is reached in a recursive function call, the corresponding function instance terminates and returns to the calling function instance, the associated stack frame is pulled from the stack and the memory is freed. The process does not terminate here; it continues until the last function terminates.

Accordingly, with every function return, a solution for a small instance of the original problem is obtained and utilized as a part of the solution of a bigger sub-problem. The

solution can only be complete when all calls return and the last sub-problem, the last piece of the puzzle is slotted into place.

A recursion will always require an amount of memory storage depending on the stack frames in the stack defined as the computational space complexity (explained later).

Nonetheless, the control stack may have a limited allocated amount of address space; the number of stack frames that can be allocated at the same time is limited, so if a program attempts to perform more recursive calls than the limit, especially in the case of infinite recursions, it will crash because of a "stack overflow" error. Yet, it may still be possible to allocate extra stack space to avoid this error for relatively big recursions.

Finally, it is worth mentioning that as recursions are implemented implicitly through a stack, a recursive approach is considered a powerful and elegant solution to tackle problems that require explicitly managing and manipulating a stack or other similar data structures, e.g., arrays, trees.

Recursion and Activation Tree

Recursive calls can be visualized as a recursion tree where every node corresponds to a function call with specific arguments. A recursive function call is represented as an edge from the node of the calling function instance to that of the callee; the children of a node correspond to the function calls initiated by their parent node function call. The nodes appear top-down starting from the root node til the leaf nodes which correspond to the base cases. Besides, the values appearing in the nodes indicate the input values (parameters) associated with the call. It can be extended to an activation tree that represents the returns from functions after their completion and display the returned values (computed bottom-up) beside the input/parameter values in the corresponding function node. In Figure 3.4, we show an illustration of the general recursion tree for the recursion factorial function when given an input n. To its right, in Figure 3.5, we show the activation tree for the factorial function for a specific value of $n = 4$.

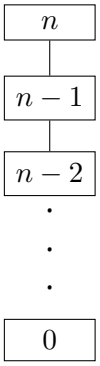


Figure 3.4 – $Factorial(n)$ Recursion Tree

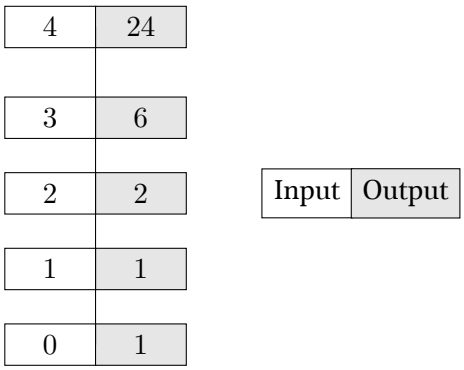


Figure 3.5 – $Factorial(4)$ Activation Tree

As an additional example, we show the recursion tree corresponding to the recursive Fibonacci function with an input $n = 5$ in Figure 3.6.

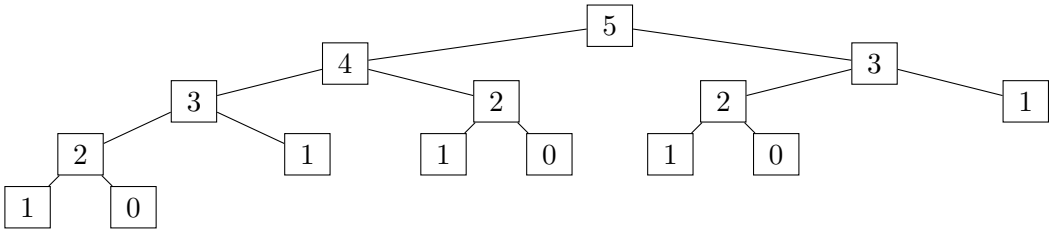


Figure 3.6 – $Fibonacci(5)$ Recursion Tree

We notice that the factorial recursion tree has a more linear shape while the Fibonacci tree is more tree-like as it has branches. This is because every time the factorial function executes, it calls itself only once from its own body; on the other hand, the Fibonacci function calls itself twice in every new call execution. So, every node has two child nodes corresponding to the two recursive function calls initiated. The structure of the recursion tree depends on the type of the recursion which is explained next. Note that these tree representations can be useful for debugging and may give an insight about the time and space computational complexity of the corresponding recursions. Run-time analysis and the computational complexity of recursions are discussed later in this section.

3.1.3 Types of Recursion

There are many types of recursions depending on how a function invokes itself, how many recursive function calls exist and where they are placed in the body of the recursive function. Accordingly, recursions can be direct or indirect (mutual), multiple, nested, linear or tail.

Direct vs. Indirect Recursion

A direct recursion occurs when a function directly calls itself from its own body code. The recursive examples shown in this manuscript till now, the factorial and matrix-vector product, are direct recursions. However, a recursion can still occur even if a function does not call itself right away. Indirect or Mutual recursions occur when a set of functions call each other in a cyclical order. For instance, if there are two functions such that one calls another and then the latter calls back the prior, then these functions are involved in an indirect recursion as illustrated in the Listing 3.6.

```
void F_1() {
    //code
    F_2();
    //code
}
void F_2() {
    //code
    F_1();
    //code
}
```

Listing 3.6 – Indirect Recursion Example

Multiple or Tree Recursion

Multiple recursion takes place when a function calls itself multiple times. If the function invokes itself twice then it can be also described as a "binary recursion". This type of recursion is well-known and particularly used to implement "divide and conquer" algorithms like recursive sorting algorithms, e.g., merge sort (Listing 3.7), quicksort, and many others, e.g., Strassen's matrix-multiplication (based on dividing the matrices dimensions in halves) and Fibonacci function (Listing 3.2), etc.

```
void MergeSort(int * array, int left, int right)
{
    if (left < right)
    {
        int middle = left+(right-left)/2;
        MergeSort(array, left, middle);
        MergeSort(array, middle+1, right);

        Merge(array, left, middle, right);
    }
}
```

Listing 3.7 – Multiple Recursive Merge Sort C Function

Nested Recursion

Nested recursion is a rare type of recursions that occurs when an argument of a recursive function is defined through another recursive call. As an example of this type of recursion, let us consider an alternative version of the recursive Fibonacci function shown in Listing 3.8.

```
int Fibonacci(int n , int s)
{
   if (n <= 2)
      return 1+s;
   return Fibonacci( n-1 , Fibonacci(n-2,0) );
}
```

Listing 3.8 – Nested Recursive Fibonacci C Function

Linear Recursion

Linear recursion occurs when a recursive function has only one recursive function call in its body, i.e., when a recursive function calls itself once only. The recursive factorial function (Listing 3.1) is an example of such a type of recursions.

Tail Recursion

A tail recursion is when the recursive call is the last instruction to be executed in the recursive function. For instance, the factorial example (Listing 3.1) may seem tail recursive,

but it is not so. This is because the recursive call of factorial with the argument $n-1$ is not the last instruction executed; the value returned by this recursive call is then multiplied by n in the calling function. Yet, the factorial function can be rewritten with a tail recursion instead as shown in the following Listing 3.9

```
int Factorial( int n , int i )
{
    if (n == 0)
        return i;
    return Factorial( n-1 , n*i );
}
```

Listing 3.9 – Tail Recursive Factorial C Function

Finally, there is also a **head recursion** where the recursive call is the first instruction in the function.

3.1.4 Runtime Analysis

Algorithms run-time analysis or study of complexity enables determining the resources, e.g., space and time, required by the algorithm in order to solve a problem. It can be expressed as a function of the input size. This is an important measure since it helps comparing the efficiency of different recursive or non-recursive algorithms. Computing this measure precisely is usually difficult because its execution depends on other factors than the input like the hardware, the programming language, the compiler, etc., so the efficiency of algorithms is commonly studied by contemplating how the complexity behaves when the size of the problem is very large, i.e., when the input size tends to infinity; this is known as the asymptotic behavior of the complexity. Thus, the complexity is usually expressed as an asymptotic notation that involves discarding the lower order-terms and multiplicative constants when dealing with randomly large inputs; assuming that the algorithm has only one size factor/input n, there exist the following notations:

- Big-O notation defining the set:
  $\mathcal{O}(g(n)) = \{f(n) : \exists c > 0 \ \wedge \ n_0 > 0 \mid 0 \leq f(n) \leq c.g(n), \forall n > n_0\}$ such that:
  If $f(n) \in \mathcal{O}(g(n))$, then $g(n)$ is an asymptotic upper bound for $f(n)$. It specifies the maximum amount of resources the algorithm needs in the worst case scenario. Algorithms are commonly compared according to their efficiency in the worst case corresponding to a problem instance of the same size requiring more resources.

- Big-Omega notation defining asymptotic lower bounds:
  $\Omega(g(n)) = \{f(n) : \exists c > 0 \ \wedge \ n_0 > 0 \mid 0 \leq c.g(n) \leq f(n), \forall n > n_0\}$
  It specifies the lower bounds on the required resources.

- Big-Theta notation defining asymptotic tight bounds:
  $\Theta(g(n)) = \{f(n) : \exists c_1 > 0, c_2 > 0 \wedge n_0 > 0 \mid 0 \leq c_1.g(n) \leq f(n) \leq c_2.g(n), \forall n > n_0\}$
  If the running time $f(n)$ of an algorithm $\in \Theta(g(n))$, then $f(n) \in \mathcal{O}(g(n))$ and $f(n) \in \Omega(g(n))$, $f(n)$ and $g(n)$ share the same order of growth, and the algorithm will always require, in the best and worst cases, an order of $g(n)$ operations.

The common computational complexity orders of growth are sorted considering large values of n as follows:

1 (constant) $< log\, n$ (logarithmic) $< n$ (linear) $< n\, log\, n < n^2$ (quadratic) $< n^3$ (cubic) $< 2^n$ (exponential) $< n!$ (factorial)

Computational Time Complexity

The computational time complexity is a mathematical measure that describes, in particular, the amount of time needed and the number of operations performed by the algorithm in order to solve a problem. It is more straightforward to compute the complexity of the iterative algorithms than that of the recursive algorithms. As for a recursive algorithm, the number of operations carried out is specified through a recurrence relation.

Recurrence Relation A recurrence relation is a recursive mathematical function describing its computational cost. For instance, let us reconsider the recursive definition of the factorial function:

$$factorial(n) = \begin{cases} 1 & \text{if } \ n = 0, \\ factorial(n-1) \times n & \text{if } \ n > 0 \end{cases}$$

In the base case ($factorial(0)$), the computation requires a finite number of operations, let it be a. if $n \geq 1$, the function will perform a fixed number of operations (multiplication...) b in addition to performing again the factorial computation for input $n-1$. Let the function T given input n denotes the computational complexity of the factorial function and it is defined as:

$$T(n) = \begin{cases} a & \text{if } \ n = 0, \\ T(n-1) + b & \text{if } \ n > 0 \end{cases}$$

The computational cost is computed by solving this recurrence. As follows, we briefly discuss the main methods for solving common recurrence relations.

Substitution Method This method consists of the two steps:

1. guessing the solution
2. proving that this solution is valid using mathematical induction

For example, for the recurrence of the factorial function, let us guess that the complexity is linear w.r.t. input n with constants c and d to be determined.
Our guess is: $T(n) \leq cn + d$. Induction proof is as follows:

- **Base Case:** for $n = 0$, we have:
  $T(0) = a$ from the recurrence base case, and
  $T(0) \leq d$ from our guess, which is true for all $d \leq a$.
- **Inductive Step:** we suppose that the solution $T(n) \leq cn + d$ is correct for some $n > 0$; we need to prove that this is also correct for $n + 1$:
  $T(n+1) = T(n) + b$ is obtained from the recurrence; we need to substitute $cn + d$ for $T(n)$ from our hypothesis:
  $T(n+1) \leq cn + d + b = cn + c - c + d + b = c(n+1) + b - c + d \leq c(n+1) + d$ which is true for $b \leq c$.

So, $T(n) \in \mathcal{O}(n)$. Furthermore, we can go further than getting the asymptotic upper bound only, and we can similarly show, in this example, that $T(n) \in \Omega(n)$. Therefore, we determine that in the best and worst cases, the factorial algorithm takes an order of n operations, i.e., $T(n) \in \Theta(n)$ and it is said that $T(n)$ grows linearly w.r.t. n.

Backward Substitution, Iteration or Expansion Method In this method, we take the recursive case of the recurrence relation, we expand the recursive terms on its right-hand side several times (recursive steps) til we guess or detect a general pattern for the i^{th} step. Then, we find the value of i with which the base case is reached and substitute this value for i in the pattern we have previously got. For example, this method is more straightforward to use to solve the factorial recurrence. Starting from the recursive case $T(n) = T(n-1) + b$; we expand it by replacing $T(n-1)$ by $T(n-2) + b$, then $T(n-2)$ by $T(n-3) + b$, so we get:

$$T(n) = T(n-1) + b = [T(n-2) + b] + b = T(n-3) + 3b$$

We notice that the general pattern is:

$$T(n) = T(n-i) + ib$$

For some value of i, the base case defined for $T(0)$ will be reached; $T(n-i)$ reaches the base case when $i = n$. We eliminate the variable i by substitution as follows:

$$T(n) = T(n-i) + ib = T(n-n) + nb = T(0) + bn = 1 + bn \ \in \ \Theta(n)$$

Recursion Tree Method Usually, the recursion tree (explained earlier) provides an idea about the computational complexity of a recursion. If the tree is linear like that of the factorial function, then the cost of a subproblem/call (disregarding the recursive calls cost), which is $\Theta(1)$ in case of the factorial example, multiplied by the height of the tree (n) determines the complexity $(\Theta(n))$. However, if it is not, then the complexity depends on the number of the tree nodes. For instance, if the recursion tree is a full binary tree of height n, then the number of nodes will be 2^n. Accordingly, we can hypothize about the run-time of recursive algorithms whose recursion trees has a similar structure even if it is not balanced and contains less nodes like that of the Fibonacci algorithm (Figure 3.6); we can conclude that their run-time upper bound is $\mathcal{O}(n)$, but, this solution is neither precise nor a tight bound.

Alternatively, the recursion tree method involves visualizing the recurrence as a recursion tree where each node represents a subproblem or a recursive call and its value represents the cost of the operations carried out by the corresponding call. The cost at each node does not include the operations performed by further recursive call. The computational time complexity can be determined by summing the nodes costs within each tree level and then summing all the levels costs.

For example, let us take the merge sort function (Listing 3.7) that involves dividing its problem size by two with every recursive call until the base case is reached where there is only one element left in the array to sort; the array elements left to sort is defined as n in the recurrence function T of the merge sort algorithm. The work done at the base case is $\Theta(1)$; the recursive case, in addition to the fixed number of operations performed through the non-recursive call to the function Merge that takes $\Theta(n)$, it also requires the

operations performed by the recursive calls. Its recurrence relation is:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

The initial input of this recurrence is n, the recursive part involves two recursive calls each with half the input n, and the cost of the non-recursive part is $\Theta(n)$. Accordingly, the value of the root node will be n, and this node will have two children corresponding to recursive calls each of size $\frac{n}{2}$. Then, the values of the children nodes will be $\frac{n}{2}$, and they will have, in turn, children corresponding to $T((n/2)/2)$ in the next level, and so on. The tree is illustrated in Figure 3.7.

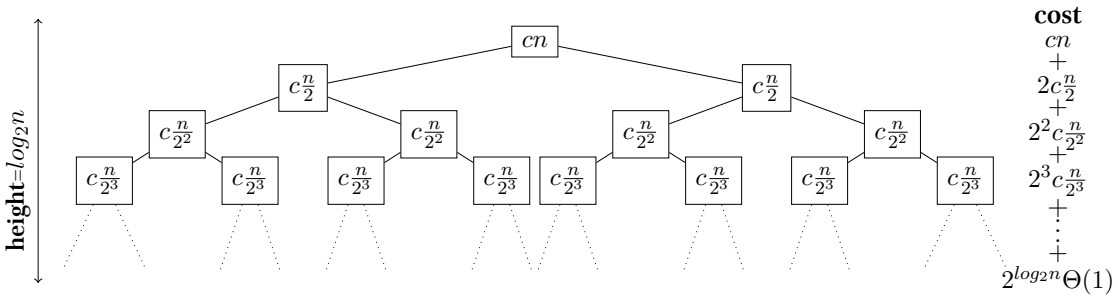


Figure 3.7 – Recurrence Tree

The expansion of the tree will stop, i.e., the leaves of the tree are reached when the base case is met ($n = 1$); at this level the cost of work at each leaf is $\Theta(1)$, and the depth of the tree is $log_2 n$. By observing the tree expansion, the cost of a node at a level i (starting at level 0) is a constant multiplied by $\frac{n}{2^i}$, and the number of nodes is 2^i. Then the cost is: $\Theta(\sum_{i=1}^{logn} 2^i \times c \times \frac{n}{2^i}) = \Theta(nlogn)$. However, the tree method does not necessarily gives an accurate solution for the recurrence; it rather gives an insight or a good guess about the time complexity without providing a proof.

Master Theorem This theorem serves as a quick method for computing the computational complexity of recursive algorithms based on divide and conquer that solve a subproblems whose size equals that of the original problem divided by b. It is exclusively used to solve recurrence relation of the form:

$$T(n) = \begin{cases} c & \text{if } n = 1, \\ aT(n/b) + f(n) & \text{if } n > 1 \end{cases}$$

such that $a \geq 1$, $b \geq 1$, $c \geq 0$, and f is an asymptotically positive function. The asymptotic tight bound of T can be generally determined by the master theorem based on the terms of the recursive case of its recurrence, $aT(n/b)$ and $f(n)$, according to these following three cases:

1. $f(n) = \mathcal{O}(n^{log_b a - \epsilon})$ with constant $\epsilon > 0 \implies T(n) \in \Theta(n^{log_b a})$
2. $f(n) = \Theta(n^{log_b a}(logn)^k)$ with $k \geq 0 \implies T(n) \in \Theta(n^{log_b a}(logn)^{k+1})$
3. $f(n) = \Omega(n^{log_b a + \epsilon})$ with $\epsilon > 0$ and $af(n/b) \leq df(n)$ for some constant $d < 1$ and a sufficiently large $n \implies T(n) \in \Theta(f(n))$

If $f(n)$ is a polynomial of degree k, the following simpler definition of the master theorem can be applied:

$$T(n) = \begin{cases} \Theta(n^k) & \text{if } \frac{a}{b^k} < 1 \\ \Theta(n^k log n) & \text{if } \frac{a}{b^k} = 1 \\ \Theta(n^{log_b a}) & \text{if } \frac{a}{b^k} > 1 \end{cases}$$

For example, let us consider again the merge sort function (Listing 3.7). We can also solve its recurrence by applying the general master theorem (its first definition) such that:
$a = 2$, $b = 2$ and $f(n) = \Theta(n)$, so
$f(n) = \Theta(n) = \Theta(n^{log_b\, a}\, (log\, n)^k) = \Theta(n^{log_2\, 2}\, (log\, n)^0)$ for $k = 0$.
Then, it falls in the second case where $T(n) \in \Theta(n^{log_2\, 1}\, (log\, n)^{0+1})$, i.e., $T(n) \in \Theta(n\, log\, n)$.
Moreover, in this example since $f(n)$ is a polynomial of degree $k = 1$, we can make use of the simplified master theorem; since $\frac{a}{b^k} = \frac{2}{2^1} = 1$, then we can directly apply the second case which indicates that $T(n) \in \Theta(n^1\, log\, n)$, i.e., $T(n) \in \Theta(n\, log\, n)$.

The general Method The methods discussed earlier are effective for resolving recurrence relations particularly those where the recursive function appears once only in the recursive case. However, there may exist recurrences that call themselves multiple times in the following particular form:

$$T(n) = -a_1 T(n-1) - \cdots - a_k T(n-k) + P_1^{d_1}(n) b_1^n + \cdots + P_s^{d_s}(n) b_s^n$$

where a_i and b_i are constants, and $P_i^{d_i}(n)$ are polynomials of n of degree d_i. The definition is composed of the terms involving T called "T difference" terms and may involve polynomial terms multiplied by an exponent of the input n (exponential terms). Here, we only discuss the simplest forms of such recurrences a.k.a homogeneous recurrences, and we show how they are solved using this method using a basic example, the recurrence of the classic recursive Fibonacci algorithm (whose code is in Listing 3.2). These recurrences only consist of the "T difference" terms as:

$$T(n) = -a_1 T(n-1) - \cdots - a_k T(n-k)$$

Such types of recurrences can be solved by the following steps:

1. passing all the "T difference" terms on the right-hand side to the left hand-side:
$$T(n) + a_1 T(n-1) + \cdots + a_k T(n-k) = 0$$
It is a homogeneous recursion since what is left on the right hand side is 0, so we can proceed with the following steps.

2. defining the associated characteristic polynomial by substituting x^{k-z} for $T(n-z)$ for $z = 0, ..., k$:
$$x^k + a_1 x^{k-1} + \cdots + a_{k-1} x + a_k = 0$$

3. the k roots of this polynomial. Assuming that r_i is its i^{th} root, then it can be factorized as follows:
$$(x - r_1)(x - r_2) \ldots (x - r_k)$$

4. redefining T using the following non-recursive expression if all the roots are different from each other:
$$T(n) = C_1 r_1^n + \cdots + C_k r_k^n$$
where $C_1, ..., C_k$ are constants whose values will depend on the base cases of the recurrence.

5. Determining these constants by solving a system of k linear equations with k variables C_i, we need to find k initial values of T (T base cases) in order to construct the k equations.

The recurrence relation of the recursive Fibonacci function is:

$$T(n) = \begin{cases} 0 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ T(n-1) + T(n-2) & \text{if } n > 1 \end{cases}$$

This recurrence relation can be solved by applying the explained steps as follows:

1. $T(n) = T(n-1) + T(n-2) \rightarrow T(n) - T(n-1) - T(n-2) = 0$.
2. The corresponding characteristic polynomial is: $x^2 - x - 1$.
3. Factorized characteristic polynomial is: $(x - \frac{1+\sqrt{5}}{2})(x - \frac{1-\sqrt{5}}{2})$.
   The square roots are: $r_1 = \frac{1+\sqrt{5}}{2}$ and $r_2 = \frac{1-\sqrt{5}}{2}$.
4. Non-recursive T expression is: $T(n) = C_1(\frac{1+\sqrt{5}}{2})^n + C_2(\frac{1-\sqrt{5}}{2})^n$.
5. The linear equation to solve is: $\begin{cases} C_1 + C_2 = T(0) = 0 \\ \frac{1+\sqrt{5}}{2}C_1 + \frac{1-\sqrt{5}}{2}C_2 = T(1) = 1 \end{cases}$
   Its solutions are: $C_1 = \frac{1}{\sqrt{5}}$ and $C_2 = -\frac{1}{\sqrt{5}}$.

The solution of the Fibonacci recurrence is:

$$T(n) = 1\sqrt{5}(\tfrac{1+\sqrt{5}}{2})^n - 1\sqrt{5}(\tfrac{1-\sqrt{5}}{2})^n$$

The growth of this exponential function depends mainly on $r_1^n = (\frac{1+\sqrt{5}}{2})^n$ because: $r_1 > r_2$ and, as n goes to infinity, $r_2^n = (\frac{1-\sqrt{5}}{2})^n$ approaches 0 since $|r_2| < 1$. Also, in asymptotic notations, we can ignore the constants, so $T(n) \in \Theta((\frac{1+\sqrt{5}}{2})^n) \approx \Theta(1.618^n)$. This is compliant with what we know about the complexity of the Fibonacci using the recursion tree earlier, yet accurate.

More details about this method (concerning non-homogeneous recurrences, etc.) and more information about recursions in general are found in the book [116].

Computational Space Complexity

The computational space complexity describes the memory storage needed by an algorithm. As for a recursive algorithm, since it is implemented through an implicit stack, a.k.a the program stack, its space complexity depends on the largest number h of stack frames placed on this stack (and their sizes) at any moment during execution. So, a recursion will always require h units of memory ($\in \Omega(h)$) minimum. It can be also understood as the height or depth of the recursion tree h multiplied by the amount of memory M required by each recursive call (the size of a stack frame). For example, the recursive tree of the factorial example (Figure 3.4) has a depth of n and each stack frame requires only a constant storage space ($\Theta(1)$) for storing arguments, etc., so the space complexity is $\Theta(n)$. This recursion is linear and it has a linear number of nodes, a linear depth and, accordingly, a linear space growth. As another example, let us consider the recursive tree of the Fibonacci recursive algorithm (Figure 3.6). Although it has an exponential number

of nodes, its depth is n for input n which is linear which determines the memory storage requirements. This is because not all recursive calls are executed at the same time; a base case ($n = 0$ or $n = 1$) is always reached after n recursive calls maximum, so there will not be more than n stack frames in the program stack at a time. Then, the space complexity of the Fibonacci function is also linear w.r.t n, i.e., $\in \Theta(n)$.

3.2 Optimizing Recursive Programs "as They Are"

Recursive functions are time, compute and space intensive structures. For this reason various studies exist for the purpose of optimizing, improving data locality and parallelizing them. Nevertheless, the proposed techniques are mainly static, not as aggressive as loop dedicated optimization techniques, or dedicated to special forms of recursions. We categorize these works into two categories: (1) based on the classical task parallelization and (2) based on polyhedral modeling.

3.2.1 Task Parallelism

Task parallelism is the decomposition of a task into smaller tasks that are distributed over multiple processors and executed simultaneously. In recursive codes, task parallelism involves concurrently executing recursive calls that are meant to solve sub-problems constituting the original problem.

Automatic Task-based Parallelization of Recursions

Interesting works enabling automatic task parallelism techniques are presented as follows.

Rugina and Rinard [117] present a static compiler that parallelizes multiple recursions, divide-and-conquer recursive functions in particular. In order to do so, it uncovers independent recursive calls based on pointer analysis and symbolic analysis, and, accordingly, generates a code that executes these independent calls concurrently. Also, for divide-and-conquer implementations, Gupta *et al.* [53] propose a compile-time framework that makes use of inter-procedural symbolic array section analysis to capture the independent multiple recursive calls and parallelize them automatically. It also enables a speculative run-time parallelization technique when static analysis is not sufficient. Besides, there is also an implemented tool called Huckleberry [34] that automatically parallelizes recursive divide-and-conquer codes for multi-core platforms.

Another technique proposed by Morihata and Matsuzaki [94] enables automatic parallelization for recursive functions using quantifier elimination such that the input structure is decomposed into blocks that are run in parallel. In addition, Mizutani et al. [91] present a different approach to parallelize recursive functions where programmers decide themselves whether to use simple or dynamic load balancing depending on the recursive functions workload. Accordingly, each invocation is executed in parallel until a threshold that is also specified by the programmer. Saougkos et al. [122] propose an automatic fine-grained parallelism extraction method for recursive functions involving integer variables that are updated in a systematic fashion.

Gupta *et al.* propose DECAF in a more recent work [54] which is a technique to optimize recursive task parallel programs by reducing the task creation and termination overheads. Moreover, there is Adriadne [85] which is a compiler that extracts directive-based parallelism from recursive function calls. It is wider than the scope of this category as it extracts three forms of parallelism and a transformation for each of them: (1) recursion elimination: recursion-to-iteration conversion, (2) parallel-reduction: recursion elimination and workload distribution into independent tasks, (3) thread-safe parallelizion of recursive functions containing independent recursive calls. This work is revisited in the next section.

Limitations

This optimization technique is still classical, unrefined and not as robust as the optimizations and parallelization opportunities that the polyhedral model offers for loops for instance. Probably, rescheduling a recursive code may reveal significant more aggressive optimizations.

3.2.2 Polyhedral Modeling of Recursive Invocations

As already seen with the polyhedral model, analysis and transformation are usually performed statically at the level of the statements dynamic instances. Polyhedral analysis and transformation have been exclusively dedicated to sequences of affine loop nests. Since this model has been so powerful for such programs, interest in applying it to analyze and transform recursive programs to optimize them and improve their data locality has risen.

One of the main works in this area is that of Amiranoff *et al.* [4] based on [31, 43] that generates context-free language representations of general recursive programs providing dependence analysis among the instances and parallelizes them accordingly. Yet, their optimizations are still simple and cannot handle nested recursions for instance. Besides, there exist many recent works and frameworks that support other transformations and optimizations of recursive codes, e.g., nested loops and recursions interchange and blocking [59, 60, 144], multiple recursive traversals fusion [108, 118, 100] and transformation of multiple nested recursive functions [135]. However, these works are “ad hoc” according to Sundararajah *et al.* [133] since they do not provide general ways of reasoning about combinations of recursions and loops and transformation correctness. An interesting framework called PolyRec proposed by Sundararajah *et al.* [133, 134], combining the approaches mentioned above, represents recursive function dynamic instances and their dependences as polyhedra, and applies polyhedral scheduling transformations (e.g., interchange and code motion) that improve data locality and enables parallelism. However, their approach is exclusively committed to particular forms or structures of non-mutual recursions such that recursive invocations are nested and data is organized in two trees, the inner and outer trees.

To sum up, there exist different approaches to optimize recursive calls. Although the majority of the works fall under the category of the classic task parallelism, many interesting works have got recently inspired by the polyhedral model and the powerful analysis and transformations that it provides in the realm of iterative programs. However, all of

these approaches are still static, depend on the information only known at compile time, dedicated to special recursive structures (mainly trees), or with restricted optimizations.

In our work, Rec2poly, does not need to know all the information at compile time and does not rely on the type of the recursion it is handling as long as its behavior is polyhedral-compliant. This is the main feature that distinguishes our approach, Rec2Poly.

3.3 Transforming Recursive Programs as Loops

In this section we discuss the relation between recursions and iterations and the reasons behind the interest in recursion optimization mainly through recursion-iteration transformation a.k.a recursion removal/elimination. There have been conducted intensive studies on this subject especially between the 1970s and early 2000s. However, the majority of the transformation techniques proposed back then were non-automatable or not so powerful. Here, we focus on the options that we can rely on, today in modern compilers in particular, to automatically optimize recursive codes in imperative languages.

3.3.1 Recursion and Iteration: Two Sides of the Same Coin

A recursion and an iteration can be alternatively used to perform repetitive computations to solve a problem. Recursion and iteration are implemented using different control structures, recursive functions and loops respectively; recursive calls involve accessing an implicit stack, a.k.a the program stack, while loops do not.

There are probably various approaches, recursive and non-recursive algorithms to solve a certain problem. Whether one chooses to solve the problem using recursion or iteration, there is always a way to convert the prior to the latter and vice versa. It is well-known that a recursion can be simulated and rewritten using iterative control structures, probably conditional loops, e.g., while loops, in addition to an explicit data structure that mimics the program stack to save and keep track of the parameters and local variables. This conversion is mainly achieved by putting, within the loop, the calculations to be performed by the recursive functions, and replacing each recursive function call and return by a push to and a pop from the stack respectively inside the loop. Tail recursions, in particular, can be transformed into loops without the need of a stack; this is because a tail recursive call is the very last instruction in a recursive function, and it actually does not need or use stored information on the stack; so, this recursion-to-loop transformation can be done without an explicit stack. Reversely, iterations can be implemented as recursions, as tail recursions in particular, using recursive functions instead of loops.

Usually, a recursive approach is effective, yet more elegant, to adopt than an iterative approach when the latter involves manipulating a data structure, e.g., stack, queue or array, etc; we can skip writing all the data structure access instructions inside a loop in a program by simply implementing a recursion instead. Nonetheless, when there is no need for such expensive data structures to solve a problem, even if an iterative approach is less elegant, it is more efficient.

3.3.2 Recursion Versus Iteration: Elegance/Efficiency Trade-off

If a recursive algorithm has the same computational complexity as an iterative one, the recursive code usually stays less efficient. For example, let us compare both of the classical recursive and iterative factorials computation algorithms, in Listings 3.1 and 3.10 respectively; they both require an order of n computations. However, we consider the

```
int Factorial (int n)
{
    int factorial=1;
    while(n>0)
    {
            factorial = factorial*n;
            n--;
    }
    return factorial;
}
```

Listing 3.10 – Iterative Factorial C Function

iterative algorithm to be more efficient, mainly, because of the fact that recursions are implemented through the program stack while loops are not; the drawbacks are that:

- recursive functions require more memory (in the form of stack frames) than the loop structures
- stacking/de-stacking operations in recursive calls induce computational overhead
- so many recursive calls may cause stack overflow and program crashes

Moreover, a recursive approach, particularly when it involves a multiple recursion, may be considerably slower than its alternative iterative approach since it may solve overlapping subproblems and perform redundant computations. For instance, let us consider the classic recursive Fibonacci numbers algorithm (code in Listing 3.2). It requires an exponential number of computations and recursive calls w.r.t. n in order to compute the n^{th} Fibonacci number. Its recursive definition is:

$$Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2)$$

Computing Fibonacci of n requires that the function is called again for both inputs $n-1$ and $n-2$, and calling the Fibonacci function for $n-1$ initiates another recursive call for $n-2$ and so on. So, Fibonacci for $n-2$ is computed one more extra time, so is $Fibonacci(n-3)$, etc. Other computations may be repeated more than just twice. For example, let us go back to the Fibonacci recursion tree (in Figure 3.6); when $Fibonacci(5)$ executes, it performs the same recursive calls and computations twice for $n = 3$ and thrice for $n = 2$.

3.3.3 Recursion Optimization: Loop at the End of the Tunnel

Recursions with the redundant overhead due to overlapping recursive calls and computations can be optimized using dynamic programming [16] which speeds them up by storing the results of the executed function calls and returning the cached values when the

same inputs are encountered again in new executions; so, $Fibonacci(k)$ for $1 < k < n$ is only computed once. Dynamic programming techniques majorly include:

- Memoization [88]: Top-down recursive approach; for instance, memoization in Fibonacci is achieved by saving computed numbers starting from $Fibonacci(0)$, so $Fibonacci(n)$ is on the top of computations (Listing 3.11).
- Tabulation [19]: Bottom-up iterative approach, example is in Listing 3.12.

Both of these optimized versions require a linear time or number of computations w.r.t. the input n. Yet, the recursive approach is still less efficient due to the stack accesses. Another space-optimized iterative version is in Listing 3.13, since we only need the last two computed values, we can use some scalar constants replacing the array "mem" of size $n + 1$. Furthermore, it is also possible to develop a faster iterative algorithm for the Fibonacci numbers that requires a logarithmic time w.r.t. the input only.

```
int Fibonacci(int n) {
    if ( n<=1 && mem[n]==0 )
        mem[n] =n;
    else if (mem[n]==0)
        mem[n] =Fibonacci(n-1) +Fibonacci(n-2);
    return mem[n];
}
```

Listing 3.11 – Recursive Fibonacci C Function - Memoization

```
int Fibonacci(int n) {
    int mem[n+1];
    int i;
    mem[0] = 0;
    mem[1] = 1;

    for (i = 2; i <= n; i++)
        mem[i] = mem[i-1] + mem[i-2];

    return mem[n];
}
```

Listing 3.12 – Iterative Fibonacci C Function - Tabulation

```
int Fibonacci(int n) {
    int first = 0, second = 1, temp;
    if (n == 0)
        return first;
    for (int i = 1; i <= n; i++) {
        temp = first + second;
        first = second;
        second = temp;
    }
    return second;
}
```

Listing 3.13 – Iterative Fibonacci Function - Space Optimized

Recursive programs can be memoized by hand. On the other hand, there have been implemented many automatic techniques to apply dynamic programming techniques to recursions based on memoization as in [95, 87, 46, 1] initially implemented for functional languages. As for imperative languages, there exist many libraries and tools that have been developed to efficiently automatize memoization for recursive functions such as the functools functions cache and lru_cache decorators built-in in Python [47] and C-Memo function memoization library for C programs [25]. Nevertheless, to our knowledge, applying dynamic programming automatically has been always limited by caching and straightforward recursive-to-recursive transformations. The iterative version, which

is expected to be even more efficient, is not as straightforward to derive and it requires more analysis and a deeper understanding of the computations execution order.

In general, in addition to these dynamic programming techniques, there exist many studies concerned with recursion removal or recursion-to-iteration transformation [6, 24, 143, 56]. They present static approaches to transform recursions to tail recursions or loops with or without using a stack or eliminating it whenever possible. Many of these works are non-automatable, difficult to implement or may not guarantee performance, storage or efficiency improvements. An interesting automatable recursion-to-iteration transformation algorithm based on incrementalization is proposed in [77]. It also makes use of an earlier work of a non-traditional dynamic programming technique also based on incrementalization [76] to optimize non-linear recursions automated and implemented in the prototype called CACHET [75]. There is also Adriadne [85], mentioned in the previous section, that provides recursion-to-loop transformation options.

In modern advanced compilers for imperative languages particularly C/C++ our main target in this study (e.g., GCC [48], LLVM/Clang [30], etc.), the recursion-to-iteration transformation mainly implemented and automated is exclusively dedicated to tail recursions, the most straightforward transformation replacing the tail call by a jump back to the entry of the function without the need to access the control stack or replace it with an alternative data structure; this is because, as mentioned before, the information stored due to this call in the stack is of no use later after its return. This optimization helps reduce computation (stack accesses) overhead and wasted storage space.

```
define void @MatrixVectorProduct(i32** %A, i32* %B, i32* %C) #0 {
entry:
  ...
  %cmp = ...
  br i1 %cmp, label %if.then, label %if.end

if.then:                                          ; preds = %entry
  br label %return

if.end:                                           ; preds = %entry
  ...
  %cmp1 = ...
  br i1 %cmp1, label %if.then2, label %if.end9

if.then2:                                         ; preds = %if.end
  ...
  call void @MatrixVectorProduct(...)
  br label %if.end9

if.end9:                                          ; preds = %if.then2, %if.end
  ...
  call void @MatrixVectorProduct(...)
  br label %return

return:                                           ; preds = %if.end9, %if.then
  ret void
}
```

Listing 3.14 – MatrixVectorProduct LLVM IR Function

```
define void @MatrixVectorProduct(i32** %A, i32* %B, i32* %C) #0 {
entry:
  ...
  br label %tailrecurse

tailrecurse:                                      ; preds = %if.end9, %entry
  ...
  %cmp = ...
  br i1 %cmp, label %if.then, label %if.end

if.then:                                          ; preds = %tailrecurse
  br label %return

if.end:                                           ; preds = %tailrecurse
  ...
  %cmp1 = ...
  br i1 %cmp1, label %if.then2, label %if.end9

if.then2:                                         ; preds = %if.end
  ...
  tail call void @MatrixVectorProduct(...)
  br label %if.end9

if.end9:                                          ; preds = %if.then2, %if.end
  ...
  br label %tailrecurse

return:                                           ; preds = %if.then
  ret void
}
```

Listing 3.15 – MatrixVectorProduct LLVM IR Function with Tail Call Elimination

In LLVM-Clang, there is an optimization pass for tail call elimination (-tailcallelim) [80] that checks if the callee is eligible for tail call elimination (i.e., does not need stack frame info.) and transforms the corresponding (self) call followed by a return instruction with a branch to the entry block or a basic block (labeled as "tailrecurse") right after the entry of the function, thus creating a loop. Trivial instructions between the call and the return do not prevent this transformation. As an example, we show in Listing 3.14 the LLVM/Clang human readable intermediate representation (IR) and notation of the matrix vector product code (in Listing 3.5). The function encompasses two self recursive calls of which the second one is tail and can be replaced by a loop. In Listing 3.15 , we show the optimized version after applying tail call elimination as we previously described. Besides, LLVM/-Clang can compile some recursive implementations (e.g., typical factorial or Fibonacci), that are not tail recursive due to an associative expression, into efficient code; it transforms the recursive functions to use an accumulator variable.

Additionally, besides recursion elimination through transformation into loops, recursion elimination may be achieved by recursion flattening or recursive functions inlining [136, 127]. Modern compilers apply inlining (inline expansion) by replacing the call instruction of a function by the code from its definition directly in the body of the calling function; so, the inlined function instructions are executed without the need to create,

add and remove stack frames. However, in case of recursive functions, a complete inline expansion may not be a good option especially for those involving too many recursive calls because it may lead to a code bloat; so, compilers either do not inline recursive calls at all or inline them up to a certain depth [8].

3.3.4 Limitations

Recursion elimination has never been used to enable optimizing and parallelizing loop transformations. Moreover, although a lot of works investigated the possibility to automatically apply recursive-to-iterative transformations, such transformations are still not sophisticated enough and they may not tackle complex recursive codes. For example, Adriadne [85] performs recursion elimination, but it only supports recursive functions whose parameters remain constant among recursive calls, except for one integer parameter, the index variable participating in all the conditions and the termination of the recursive function. In addition, even powerful compilers are not capable of converting recursions other than tail or simple linear recursions.

Nevertheless, generic equivalent iterative programs generated statically and automatically from recursive programs may have either the call stack simulated using an expensive dedicated data structure, or conditional loops with dynamic termination conditions, or complex loop structures with several conditional branches to mimic the original recursive cases particularly if there are multiple recursive cases. In such cases, recursion elimination may not lead to performance improvements and the resulting loops may not be eligible candidates for further powerful loop-dedicated optimizations (e.g., polyhedral optimizations). Even in the most trivial case of all, in the case of tail recursions, tail calls elimination in compilers does not allow applying further aggressive loop-dedicated transformations and optimizations to the generated loops because they do not necessarily fit in the polyhedral model. Moreover, it is possible to have a recursive function with multiple recursive calls, one of which is a tail call. If the tail call is eliminated by the compiler, the function remains recursive with a loop encompassing the other recursive calls. As an example, we reconsider the matrix vector product LLVM IR function optimized by tail call elimination by LLVM/Clang compiler in Listing 3.15; although the second call is eliminated, there is still a recursive call in the core of this function. Such kind of loops does not fit in the polyhedral model and cannot be captured by the currently available advanced polyhedral (even speculative) optimizers dedicated to loops.

3.3.5 Beyond the Limits

All of these currently available automatic recursion transformations and optimizations that are activated and applied at compile time based on a static analysis obviously lead to more efficient executions. But, these optimizations are relatively limited in comparison to the optimization options that actually exist for loop structures. Furthermore, in case these optimizations involve transforming recursive functions into loops, the resulting loops structures may not necessarily be a good fit for powerful polyhedral optimizations. Yet, this does not mean that these loops cannot behave as polyhedral ones, and, in such situation at compile time, polyhedral optimization opportunities that can be exploited may be left undiscovered.

In our work, we go beyond the classical limits that are set for recursion removal techniques. Our transformation proposal is not solely dedicated to recursions with a special structure, it is rather dedicated to recursions with a special behavior, a polyhedral one. Accordingly, not only we rely on a static analysis, but also on a dynamic analysis to guide our recursion-to-iteration transformation. So, we acquire an iterative code that is guaranteed to take advantage of powerful polyhedral optimizations. Therefore, we unveil sophisticated optimization opportunities for recursions and we seize them.

Chapter 4

Dynamic Speculative Rewriting

"We often miss opportunity because it's dressed in overalls and looks like work."
— Thomas Edison

"If opportunity doesn't knock, build a door."
— Milton Berle

Although various optimization techniques dedicated to recursive codes exist, they are classic, static or solely based on compile-time analysis. There may be a huge gap between the statements outlined in a program source code and the instructions that are actually performed by a given processor architecture. Due to this gap, static analysis alone may only provide a hint about a program execution. Accordingly, significant optimization opportunities, that may considerably enhance a program's performance, may be missed just because they cannot be discovered or the required information cannot be all known at compile-time. Yet, it is still possible to reveal these opportunities and apply efficient optimizations as soon as the actual run-time execution behavior of the program is discovered. Techniques that seize these opportunities are said to be speculative; they are based on a dynamic analysis technique and code rewriting and require dynamic verification. Such optimization techniques have been dedicated to specific control structures and memory access patterns, loops in particular. For instance, Apollo [132, 84] has made possible applying advanced polyhedral optimizations that were only applicable to statically affine loops to non-affine loop structures, in particular to those that exhibit an affine memory behavior at run-time. On the other hand, recursive structures, which may even be considerably more time-consuming than loops, have not benefited of such techniques. For this purpose, in this chapter, we present our unique solution to optimize recursions, the Rec2Poly framework based on dynamic speculative code rewriting.

We previously introduced the proof of concept of Rec2Poly's analysis, profiling and recursion-to-optimized-loops transformation phases in our work [66]. Then, in a later work [67], we presented the extension of these phases in addition to introducing the verification feature based on the inspector-executor paradigm. Here, we revisit, discuss and extend these studies and the inspector-related parts in particular.

This chapter is organized as follows. In the first section, Section 4.1, we present an overview of the Rec2Poly framework. Then, we develop in details each phase of Rec2Poly in a separate section. In Section 4.2, we introduce the static code analysis and prepara-

tion phase of recursive programs. In Section 4.3, we present our dynamic analysis profiling phase to detect an affine recursive behavior. Then, in Section 4.4, we start with the code generation phase by presenting how the code part required for the verification is generated, and, in the final section, we show how the recursion-equivalent loops are generated and optimized.

4.1 Overview of the Rec2Poly Framework

Rec2Poly is a speculative dynamic recursion optimizer based on the mainstream compiler LLVM/Clang [137]. It discovers, through an offline profiling technique, an affine run-time behavior of recursive codes that manipulate large data structures, e.g., arrays, etc. When successful, it builds a semantically equivalent code where all the execution flow related to recursive functions is replaced with affine loops that enables aggressive polyhedral loop optimizations and parallelizations. Rec2poly also generates a run-time verification mechanism following an inspector-executor scheme [15, 112] to ensure the validity of the generated iterative code for different input data. Accordingly, in order to achieve that, Rec2Poly is composed of three major phases:

1. Code static analysis and preparation phase
2. Offline profiling phase
3. Inspector-Executor code generation phase

Analyses, transformations and optimizations in these phases are mainly implemented as LLVM passes [80] processing the LLVM intermediate representation (IR) of the code [78].

The phases of Rec2Poly and their main constituents are depicted in Figure 4.1.

At the beginning, Rec2Poly performs static analysis. It deeply analyzes the target recursive code in order to identify the recursive functions. It also identifies the functions that may be invoked by or that may invoke, directly or indirectly, the recursive functions. We call these identified functions *impacting functions*. Then, Rec2Poly builds the so-called Backward Static Slice (BSS) corresponding to every memory store (write) instruction in the impacting functions, and collects the identifiers of all Basic Blocks that contain at least one instruction involved in the computation of the target memory address or the stored value, i.e., in the BSS. We call such basic blocks *impacting basic blocks*. Then, Rec2Poly prepares the code for the next phases and performs local variable globalization. It is achieved by inserting, at the beginning of an impacting function in the LLVM IR, an invocation counter and by transforming each local data structure or a scalar variable into a global data structure indexed using this counter. This helps keeping track of the initially local variables in the functions of our concern and removing dependences among their different invocations. Then, this expanded globalized code is re-analyzed to obtain the necessary recursion-related information taking into consideration the modifications applied to the original recursive code.

Using the so-collected analysis information and the expanded version of the target recursive code, Rec2Poly starts with the second phase, the run-time profiling phase, by generating an instrumented version of the target code. The instrumented code is obtained by augmenting every impacting function with instructions for generating the output trace.

The generated trace describes the control and memory behavior of the program execution; it is composed of impacting basic blocks identifiers, invocation counters values and the memory addresses referenced, in the impacting basic blocks, through load (memory read) and store (write) instructions.

After the instrumented code is executed, the generated trace is given as an input to an extended version of the Nested Loop Recognition software tool NLR [65]. NLR generates, whenever possible, a representation of the whole trace made of loops, even affine loops computing affine expressions.

The code analysis information and the profiling result obtained, i.e., the so-generated loop model, can then be used by Rec2Poly to perform the code generation and optimiza-

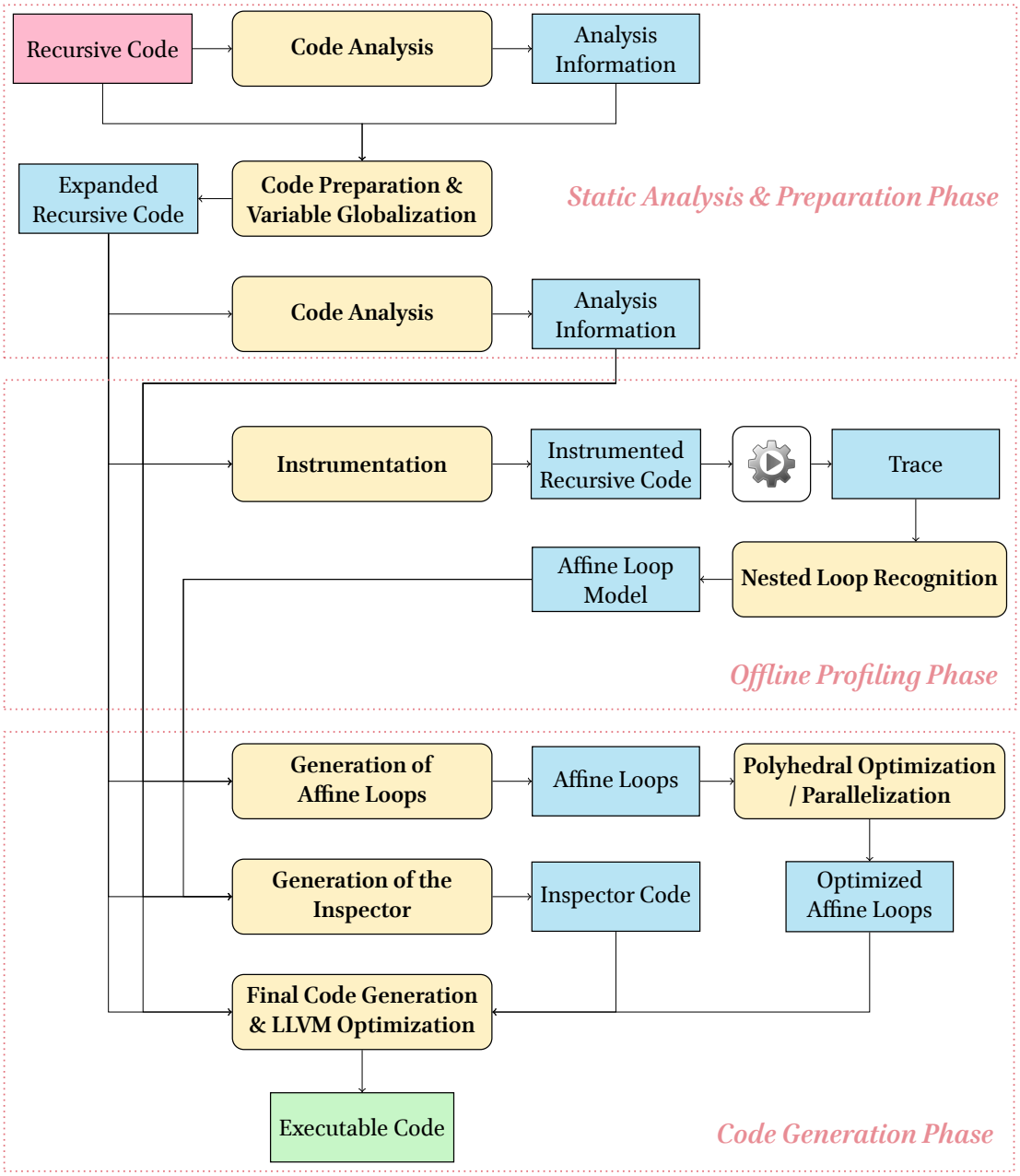


Figure 4.1 – Rec2Poly

tion phase and build an optimized version given the expanded recursive code. For that purpose, Rec2Poly generates an iterative code composed of sequences of optimizable loop nests replacing the impacting functions existing in the original recursive code.

Since the replacing loops can be fully affine loops, they can benefit from polyhedral optimizing and parallelizing transformations. For this sake, Rec2Poly uses the polyhedral compiler Pluto [101] mentioned in Chapter 2. Otherwise, a dedicated dependence analysis can still be performed and the generated loops can still be powerfully parallelized using OpenMP.

The affine loop model, obtained from the offline profiling based on one execution of the target recursive code, can also be useful for later code optimizations and executions even for different inputs of the same size, i.e., same problem size. Yet, since the further executions are speculative, their validity must still be ensured at run-time.

The generated optimized program is customizable and verifiable at run-time. This is thanks to the fast parallel inspector code generated by Rec2Poly whose role is to collect lacking information and verify, at run-time, that the generated affine loops executing in parallel are still behaving in compliance with the original recursive code.

Finally, the final code is generated based on the inspector-executor scheme made of:

- the inspector code
- the executor code containing the optimized loops
- the original recursive code

This code can be additionally optimized using the available LLVM/Clang optimizations.

During the generated program execution, the original recursive code is executed simultaneously with the inspector using POSIX Threads. This is because, in case a misspeculation is detected at run-time, the whole inspector-executor code must be cancelled and a correct state of the program's execution must be recovered at the lowest time-overhead. The thread executing the recursive code proceeds its execution as long as the inspector is still in the verification process; when a correct execution is ensured, the thread is aborted and the executor launches its parallel optimized loops.

The main phases of the Rec2Poly framework are presented and discussed in details in the following sections.

4.2 Code Static Analysis and Preparation Phase

This is the very first phase of Rec2Poly. The Clang/LLVM part of the Rec2Poly tool, takes as input the target source code and transforms it into its LLVM intermediate representation (IR). Rec2Poly performs static analysis, at the level of this generated IR, to detect the existence of recursions and collect all the necessary information required for the dynamic analysis and code transformations in the later phases. Also, at this phase, Rec2Poly modifies and prepares the code for these later phases. Note that code preparation requires information about the existing recursions and their involved functions. So, a static analysis must be performed to get this information before applying any slight modification on the original code. Afterwards, the modified expanded recursive code is deeply reanalyzed to get all the recursion-related information (impacting functions, basic blocks,

etc.), as explained later in this section, required for profiling and code generation taking into consideration the modifications made.

It would be possible to activate some available LLVM optimization passes that refine the code for the analysis, e.g., dead code elimination, promote memory to register, etc. Also, as mentioned in the previous chapter (Chapter 3), LLVM provides an optimization pass that eliminates tail recursive calls and transforms them into loops and another pass that performs an automatic recursion inlining. However, we do not enable any of these optimizations at the beginning, especially the tail call elimination LLVM pass for two reasons:

1. the way the target recursive function is transformed with tail recursion elimination may not result in an affine optimizable loop, and

2. if there are several nested recursive calls in the target code, only one tail call may be eliminated.

Accordingly, keeping recursions at this level as they are means that they will be taken into account in the analyses and then aggressively transformed and optimized using Rec2Poly whenever possible. Yet, LLVM optimizations, including recursion-related optimizations, can still be applied later after the target code is analyzed and processed as it is first.

In what follows, we describe in details how Rec2Poly performs static analysis and code preparation.

4.2.1 Static Analysis

Rec2Poly's general analysis dedicated to code preparation, run-time profiling and the generation of the inspector-executor code involves the following steps. Note that the inspector-executor generation phase requires further specific analyses than what is mentioned in this section; they will be presented later and discussed in the corresponding sections, Sections 4.4 and 4.5.

Recursion Detection

As an entry step in the analysis, Rec2Poly checks if the code is recursive and, if so, identifies the recursions and the participating recursive functions.

In order to detect recursions, Rec2Poly uses the call graph extracted from the LLVM IR of the program. The call graph is a directed graph that represents the relationships between functions in a program, where a node represents a function and an edge from one node to another represents a call such that the source node constitutes the caller function and the destination is the callee.

Figure 4.2 shows an example of a call graph of an arbitrary program composed of the following functions: $main$, A, B, C, D, E, F, G and H. In this program we have:

- function $main$ calls A
- then, A invokes B from within a loop
- B calls C

- C calls itself, E and D
- E calls back C in addition to F and G;
- finally, G calls H

This program is recursive since function C exhibits a direct recursion with itself and an indirect recursion involving function E.

Each of the function calls is represented in the corresponding call graph figure as a directed path from the caller node towards the callee. Accordingly, the recursions can be visualized in the call graph through a cycle or a loop.

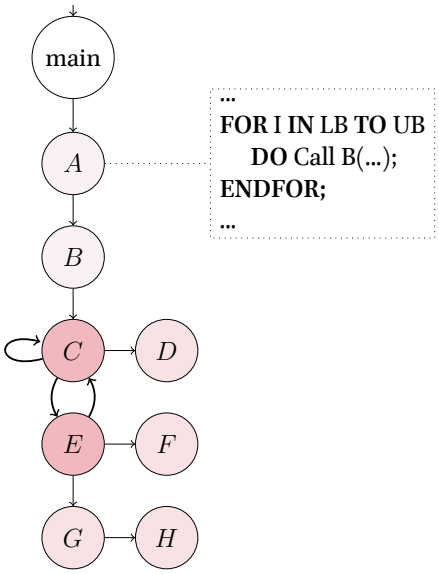


Figure 4.2 – Example of a Call Graph of an Arbitrary Recursive Program

From the call graph, in order to discover recursions, Rec2Poly seeks strongly connected components (SCC), that are sub-graphs in which every node is reachable from every other node. In this context, a cycle in a SCC indicates that there is a recursion among the functions associated to the nodes involved in this cycle. If the cycle involves only one node, *i.e.,* it is a loop, then it refers to a direct recursion. Otherwise, it is indirect. As we know, a recursion can be categorized into more different types, e.g., linear, tail or multiple. However, Rec2Poly does not distinguish between the different types of recursions; all detected recursions, whatever their types are, are similarly taken into consideration for further analysis.

For the example in Figure 4.2, there is only one SCC: The nodes of C and E are reachable from each other. Yet, there are two recursions observed in this SCC as there are:

- one loop over C showing a direct recursion, and
- a cycle from C to E and E back to C, showing an indirect/mutual recursion involving C and E.

Accordingly, Rec2Poly can identify, for instance, functions C and E as recursive functions.

Recursion Reachability Recognition

We are interested in tracking impacting basic blocks and memory accesses, whether they are executed directly or indirectly by the recursive functions.

This is necessary because recursive functions may or may not include significant expensive computations or memory access instructions in their own bodies, yet they still call or reach functions that do so. Some of these reachable functions may even include costly sequences of affine loop nests that intensively access memory; such loops would be interesting for (polyhedral) optimization. But, the following scenarios and recursion-iteration interactions may occur:

- the invoking recursion may act as a loop and can be replaced by a one;
- these functions with the beautiful loop nests may be called within a loop in the recursive functions;
- the existing recursion and function calls may distort the existence of a bigger or deeper affine loop nest and scatter its parts and iterations over different functions and invocations.

Such code parts can be neither detected nor optimized using classical or even advanced polyhedral optimizers. It would be interesting to see the whole picture and investigate the possibility of compressing such code parts and reconstructing them as a whole loop nest (without recursions) compliant for aggressive loop-dedicated optimizations and parallelization.

For this reason, besides identifying the recursive functions themselves, Rec2Poly needs to determine their reachability in the program; this constitutes its second analysis step. By reachability we mean, the set of all the functions that can be reached by a sequence of calls initiated by the recursive functions themselves.

In the call graph example in Figure 4.2, the reachability of the recursive functions C and E is the set consisting of: function D which is directly called by C, both functions F and G which are directly called by E and, finally, H that is indirectly called by E through G.

Recursion Source Recognition

Not only do we track a recursion, its involved functions and its reachability in a program, but also the source of this recursion. There may be a situation where there is a function calling from within a loop, one of the recursive functions either directly or indirectly through a sequence of function calls. Then, the source of the invoked recursion is a set of functions composed of:

- The first non-recursive function to initiate this sequence of calls from the looping structure a.k.a. the initial source function.
- All the other non-recursive functions participating in this sequence of calls reaching the recursion.

For example, the set of source functions of the recursions found in the program whose call graph is in Figure 4.2 includes functions A and B; as illustrated in the figure, A calls B from the body of a for-loop, and B, in its turn, calls the recursive function C.

We assume that, in the recursive program, the existing recursion is launched by invoking only one of its recursive functions, a.k.a. the recursive source function. Also, we assume that a recursive source function is reachable at most from only one unique initiating source function. These assumptions serve implementation purposes without invalidating the general approach.

Taking source functions into account in Rec2poly's analysis is necessary to ensure that the profiling phase is complete and its result is correct. For instance, if we ignore the existing source functions and do not track their behavior during the run-time analysis, then a detected looping behavior will be associated to the recursion itself which would be wrong. The detected loop behavior might correspond to an actual loop existing in the program and reaching this recursion somehow. Depending on such results and proceeding with code generation might lead to semantically wrong optimized iterative code. Accordingly, recursion source analysis helps understanding how a recursion behaves relatively to its encompassing structures and, obviously, building loop structures which are semantically equivalent to the whole target code part in the original program.

Furthermore, in general, loop structures calling any function and, especially, reaching recursive functions (e.g., the one in function A in Figure 4.2) cannot benefit from sophisticated loop-dedicated polyhedral optimizations since they are not considered as valid SCoPs as we know from Chapter 2. It is interesting to discover a chance and a way to blend and integrate recursive code parts with a polyhedral compliant environment. Then, if it is possible to reconstruct such loops as valid affine loop structures without recursions distorting them anymore, they may eventually be able to take advantage of advanced optimizations.

In this study, all of the recursive functions and the corresponding source and reachability functions are known as *impacting functions* in a recursive code.

Impacting Basic Blocks Identification

Rec2Poly needs to discover the control and memory execution behaviors of the target recursive code later in the profiling phase that will guide the code transformation and generation. The control behavior is defined by the execution order of the *impacting basic blocks* of the impacting functions.

The *impacting basic blocks* are the basic blocks that are crucial to trace in the profiling phase and to re-use and re-construct in the code generation phase. Within their parent functions, they contain all the necessary instructions required to access the correct memory addresses and save in them the correct values. The *impacting basic blocks* are identified in the following way.

Firstly, at the level of the LLVM IR of the program, Rec2Poly marks all the memory write, a.k.a. store, instructions to the *main data structures*. The *main data structures* are defined as being the final output data structures of the recursions and their corresponding impacting functions. A read or a load from the memory is not as significant as the write in the analysis; loads are only taken into account if they are used to perform memory writes.

Secondly, for every one of these store instructions, Rec2Poly also collects all the other instructions and computations leading and contributing to it, i.e., all the instructions that form its *backward static slice*. *A backward static slice* (BSS) is the set of instructions existing in the code of a program that may affect a value. In our case, the BSS of a store instruction corresponds to both the value that the instruction stores in a data structure and the memory address that it accesses.

The so-collected instructions, i.e., all the significant stores and their BSS, are referred to as the *impacting instructions*.

Thirdly, the parent basic blocks which contain at least one of the impacting instructions are identified. Then, these basic blocks constitute the set of the impacting basic blocks.

Intra-Function and Inter-Function Memory Access Analysis

As previously mentioned, Rec2Poly needs to analyze the memory execution behavior of the target recursive code in addition to the control behavior; we look for a potential affine looping memory and control behavior of the recursion. The memory behavior is defined by the sequence of the memory addresses touched by the memory instructions (load or store) inside the impacting basic blocks of the impacting functions.

However, Rec2Poly should not perform the run-time analysis (profiling) based on the actual addresses touched. It cannot proceed with a valid code transformation and generation relying on a model, obtained from an offline profiling, consisting of the actual memory addresses accessed in a previous execution. The reason is that among different execution instances of the same target recursive program, even if given exactly the same input data and the same hardware platform, the memory addresses accessed do not obviously remain the same. This is due to the fact that data structures are not always allocated at the same place in the memory. However, the memory behavior relative to the base addresses of the data structures remains unchanged among the different execution instances. Therefore, we are interested in obtaining loop models where the *relative* memory behavior can be expressed as affine functions of surrounding loop indices.

Accordingly, Rec2poly needs to instrument the memory offsets that are relative to the base addresses accessed instead of the actual addresses. For this purpose, Rec2Poly performs the following memory analysis to map every memory address to be accessed to its base address.

When handling data structures that are local to the impacting functions, this analysis phase can be carried out in the two steps described below:

Intra-Function Analysis Each memory access is associated to its corresponding base address accessible in the scope of the current function, i.e., the parameters of the function are the farthest analysis point reached.

Inter-Function Analysis In the case where the accessed data structures are function parameters, intra-function analysis would not be enough. Memory analysis propagates further outside the function to trace the corresponding arguments fed to this function in its invocation. Inter-function analysis associates each access to its actual base address in the whole program.

Yet, this memory analysis is much simpler when handling global data structures; the accessed memory addresses can be directly associated to their base addresses.

4.2.2 Code Preparation

Rec2Poly prepares the target code as required for the later phases. Code preparation mainly involves preserving a temporarily idle copy of the original functions in the module to be reused whenever needed and performing local variables globalization. These steps are presented as follows.

Original Code Preservation

Before Rec2Poly applies modifications to the code, it is preferable to preserve an untouched original copy of the code, i.e., all the existing functions in the module except for the main function. So, Rec2Poly must clone these functions and, then, in each of the new clones, it must replace all the referenced functions in the call instructions by their proper clones. Besides, it substitutes the global variables and their uses for new similar copies of these global variables. Also, the functions called in the main function must be replaced by their corresponding clones. This way, the cloned recursive code is the one actually launched by the main, and the original recursive code is preserved yet not invoked. Then, this dead code part will be utilized, in the code generation phase, to create the backup thread invoking the original recursive code.

For now, we suppose that the main function does not perform a memory allocation and does not pass, as parameters, pointer values. Otherwise, if the original code is launched in a parallel thread until halted due to a successful verification process, it may write to a memory address that is supposed to be accessed later by the optimized loops. Accordingly, the final generated loops may lead to a wrong output because they accessed a corrupted memory address even if the speculation has been proved to be valid. So, memory allocation at the level of the thread, besides automatic creation of global variables copies, is helpful to avoid such memory inconsistencies.

Correspondingly, any analysis performed and other modifications applied to the code, presented in this chapter, are considered to be applied to the cloned recursive code unless otherwise explicitly stated. Therefore, by default, an impacting function refers to the impacting function clone.

Local Variables Globalization

As we know and as previously mentioned in Chapter 3, when a function calls another or itself, its information and its local variables are allocated on the program/call stack until the call returns. Then, in the case of recursive calls and function calls within loop structures, accesses to the data structures local to the functions can never exhibit any affine memory accesses across all the invocations of the function. Moreover, the false data dependences that may arise due to accesses to such variables among different functions invocations must be removed if we intend to replace the impacting functions calls with affine loop structures. Besides, obviously, the local data structures must still be referenced in the constructed loops. For these reasons, the program stack role in this regard must be preserved and represented using an explicit data structure that also serves

Rec2Poly's purpose, i.e., finding an affine loop behavior among impacting functions and replacing them with equivalent loops whenever possible.

Accordingly, Rec2Poly must do the following steps after getting the required analysis information about the code concerning the recursions, the impacting "cloned" functions and their impacting basic blocks.

Function Invocation Counter Insertion Rec2Poly creates, for every significant impacting function, a *function invocation counter* as a global variable of an integer type that is initialized to value -1 in the LLVM module. The *function invocation counter* associated to a function must be incremented with every activation/call of this function. So, Rec2Poly inserts, at the first/entry basic block of the impacting functions, the instructions that enable incrementing the corresponding function invocation counters. Also, the invocation counter variables get specific metadata that helps distinguishing them from other variables which is required for later stages.

Local Variables Globalization The data structures that are local to the impacting functions are transformed into global arrays of their initial type. Then, their uses are replaced by their corresponding new global data structures which are indexed by the parent function invocation counter.

Rec2Poly must exclude all the local scalar variables that are used either as indexes of some data structures or as induction variables; the references (data structures offsets) will actually be instrumented at run-time and the values of the induction variables will be detected and presented anyway in the affine loop model in the profiling phase in case an interesting loop behavior is discovered. For now, we assume that integer scalars are always used as induction variables or arrays indexes in the target recursive code, so only non-integer local scalar variables are globalized.

This way, references to these globalized data will finally exhibit affine behaviors if the related functions are invoked following an affine control flow.

```
float A[N];


void foo () {

    int i;
    float x = 0.5;

    for ( i=0 ; i<N ; i++)
    {
        A[i] = x;
        ...
    }
    ...
}
```

Listing 4.1 – Impacting Function

```
float A[N];
int foo_invocation_counter=-1;
float x_glob[10000];

void foo () {
    foo_invocation_counter++;
    int i;
    x_glob[foo_invocation_counter] = 0.5;

    for ( i=0 ; i<N ; i++)
    {
        A[i] = x_glob[foo_invocation_counter];
        ...
    }
...
}
```

Listing 4.2 – Impacting Function after Globalization

Using Listings 4.1 and 4.2, we illustrate an example about local variables globalization at the level of the C source code instead of the LLVM IR for simplicity. In Listing 4.1, we show an impacting function called `foo` in some recursive code. In this function, we have two local variables `i` and `x` such that `i` is an induction variable. In Listing 4.2, we show the code part of the program including function `foo` after inserting its invocation counter `foo_invocation_counter` and performing local variables globalization. As we see, only `x` is globalized as the float array `x_glob`. The invocation counter is incremented at the beginning of the function and all uses of `x` are replaced by `x_glob[foo_invocation_counter]`.

Note that, only impacting functions that include impacting basic blocks get invocation counters and have their local variables globalized. However, there may exist other impacting functions that do not include significant memory instructions, and will not be instrumented anyway; they usually perform some light computations. Light computations are instructions, probably binary operations, involving integer scalars, that can be performed without touching the memory, e.g., induction variables modification or problem decomposition. Functions, with such light computations, may call the significant memory-touching impacting functions passing new arguments computed or modified in their own bodies. Such functions do neither get invocation counters nor have their local variables globalized. Even though information about such functions may not appear in the model resulting from profiling, their computed parameters are taken care of and extracted in the inspector-executor part; this aspect in explained later in Section 4.4.

After the code is modified and expanded to include the invocation counters and has its local variables of interest globalized, Rec2Poly enables the LLVM optimization pass mem2reg; it promotes memory references to be register references promoting allocation instructions which only have loads and stores as uses. This helps refining the code and omitting unnecessary expensive memory accesses.

Finally, the resulting code is the one to be used for profiling and code generation. It must be re-analyzed and the information about the existing recursions must be obtained again taking into consideration the modifications and the optimization applied.

4.3 Offline Profiling Phase

The first goal of Rec2Poly is to detect an affine looping behavior of the recursions that may allow transforming recursions to optimizable loops. The behavior of the recursions to discover corresponds to both the memory and control behaviors of the impacting functions involved.

In the speculative loop optimizer, Apollo (presented in Chapter 2), analyzing the memory run-time behavior of the loops of concern is enough to guide code transformations. Apollo intends to transform loops to other loops that are affine. So, it converts a control structure to a similar one. On the other hand, Rec2Poly transforms recursive calls to totally different control structures, loops. Accordingly, such a transformation requires understanding the recursive control behavior in addition to the memory behavior.

For this sake, Rec2Poly performs a dedicated run-time analysis using an offline profiling technique. This technique is composed of two steps: instrumentation and nested

loop recognition. These steps are presented in what follows.

4.3.1 Instrumentation

Rec2Poly generates, given the globalized version of the target recursive code, an instrumented version of the code to produce the control and memory execution trace of the impacting functions.

On the one hand, the control behavior is described by the impacting basic blocks executed at run-time. Rec2Poly needs to instrument the identifiers of these basic blocks. Their textual identifiers representation consists of the parent impacting function name concatenated to that of the corresponding basic block executed.

On the other hand, the memory behavior is described by the relative memory addresses accessed or the memory offsets. The offsets are computed for all the existing memory instructions given their associated/mapped base addresses obtained from the memory behavior analysis (presented in Section 4.2). A pointer difference instruction is inserted, to the LLVM IR, for every load or store instruction subtracting the corresponding base address from the actual memory address touched.

Then, Rec2Poly removes all the existing printing instructions from the LLVM IR and appends to it the instructions required for printing the output trace. For each impacting basic block, an instruction is added to print the basic block identifier when visited and the offsets of all the memory addresses accessed in it during the execution of the program.

However, although incrementing the functions invocations counters requires accesses to the memory, rec2Poly handles them differently. Instead of instrumenting their memory offset (obviously zero), it adds instructions to print their actual values at run-time in the entry basic blocks of the impacting functions. Moreover, the values of induction variables of existing loops must also be printed. This is significant to the code generation phase.

4.3.2 Nested Loop Recognition

Rec2Poly instruments the target recursive program in order to generate an execution trace which is composed of tuples. Each tuple is made of:

1. the impacting basic block ID,

2. the relative offsets of the memory addresses accessed by all the memory instructions in the current basic block, and

3. the values of the function invocation counters and the existing loop indices.

After running the instrumented recursive program and the execution trace is generated, this trace is analyzed by the Nested Loop Recognition (NLR) algorithm. The NLR software tool [65] and its applications have already been presented in Chapter 2. One of the main original/standard goals of NLR is to model the behavior of a program for any measured quantity such as memory addresses. In this regard, NLR, as part of the

Rec2Poly framework, is used to model the memory behavior of recursive programs. Besides, in Rec2Poly, the role of NLR is further extended to model the control behavior, i.e., the sequences of basic blocks IDs executed, which is more singular.

Also, in Rec2Poly, we need to model the loops induction variables values by NLR as affine functions of the NLR loop indices. This is because induction variables of existing loops may induce dependences that prevent the polyhedral optimizer used later in the code generation phase to apply some optimizations. Accordingly, their values need to be computed differently to remove the data dependences by using the affine expressions by NLR.

When NLR is given the output trace of the target instrumented recursive program, as an input, it may build, as an output, sequences of affine loop nests including the impacting basic blocks IDs and the memory addresses represented as affine expressions or functions of the surrounding constructed loop indices. Based on such an output, we infer that the recursive program exhibits an affine behavior when fed with this specific input and this input size.

An illustrative example of a possible NLR output modeling the execution of the recursive program (Figure 4.2) is displayed in Figure 4.3.

```
val A::BB1
for i0 = 0 to 99
  val A::BB2
  for i1 = 0 to 49
    val D::BB1, 1*i1, 1*i1
    val E::BB1, 1*i1
    val E::BB2
    val F::BB1, 1*i1, 0
    val H::BB1, 1*i0
    for i2 = 0 to 24
      val H::BB2
      , 4*i0 + 2*i1 + 1*i2
      , 1 + 6*i0 + 2*i1 + 1*i2
      , ...
      val H::BB3
    val E:BB4
  val A::BB3
```

Figure 4.3 – NLR Model for Affine Control and Memory Behavior

As we have implied earlier, the program's impacting functions are all of:

$$A, B, C, D, E, F, G \text{ and } H$$

In order to enrich this example, we choose to make the following assumptions:

- Functions B, C and G execute only light instructions that do not touch the memory; so none of their basic blocks are impacting, and none of them appear neither in the trace nor in the loop models. The impacting basic blocks are:
  - `BB1`, `BB2` and `BB3` inside function A

  - `BB1` inside D
  - `BB1`, `BB2`, `BB4` and `BB5` inside E
  - `BB1` inside F
  - `BB1`, `BB2` and `BB3` inside H

- There is a conditional branch from `BB3` to `BB4` or `BB5` in function E.

- In this execution for instance, the condition holds only to branch to `BB4` the whole time. Accordingly, the basic block `BB5` in function E is not visited at all when the program is run for profiling.

- The entry/first basic blocks of the functions are referred to as `BB1`. So, the first affine function that may appear after the entry basic block is the value of the invocation function counter.

Such generated loops show how the basic blocks are invoked by following an affine or a linear looping behavior, and how memory is referenced through relative addresses that can be modeled as affine functions of the encompassing loop indices.

Also, NLR has been extended with an advanced feature taking into consideration that the affine modeling of a trace may depend on some unknown parameters. NLR discovers these values at run-time and exhibits a memory behavior which is actually not fully affine. Such a behavior is known as a parametrically-affine behavior in which some coefficients in the affine functions representing the memory addresses may be lists of parameters values.

This is useful for Rec2Poly as it helps detecting a linearly looping control behavior of recursions regardless of their memory behavior. Also, it allows recording and capturing the memory-related parameters at run-time and representing them in a way that completes the final model that will guide the final code generation. Moreover, it enables Rec2Poly to perform a useful and precise dependence analysis for the generated loops, that are equivalent to a recursive code with such a behavior, which helps applying valid transformations and parallelizations.

As an example, Figure 4.4 shows an NLR loop model in which the memory behavior is parametrically-affine.

```
val A::BB1
for i0 = 0 to 99
  val A::BB2
  for i1 = 0 to 49
    val D::BB1, 1*i1, 1*i1
    val E::BB1, 1*i1
    val E::BB2
    val F::BB1, 1*i1, 0
    val H::BB1, 1*i0
    for i2 = 0 to 24
      val H::BB2
      , 1*i0
      , [25:3,5,...,1][i0] + 2*i1 + 1*i2
      , [25:7,1,...,6][i0] + 1*i1 + 1*i2
      , ...
      val H::BB3
    val E:BB4
  val A::BB3
```

Figure 4.4 – NLR Model for Linear Control and Parametrically-Affine Memory Behavior

We observe that instead of having constants as coefficients in some expressions, we have lists of coefficients; each list contains `25` integer values which are successively used to compute the referenced memory address. Let us consider the list `[25:3,5,...,1][i0]`; it means that:

$$[25:3,5,...,1][i0] = \begin{cases} 3 & \text{if } i0 = 0 \\ 5 & \text{if } i0 = 1 \\ ... & \\ 1 & \text{if } i0 = 24 \end{cases}$$

Based on Rec2Poly's run-time analysis information and the "beautiful" loop model resulting from NLR in case successful, Rec2Poly is capable of optimizing the next program executions by generating equivalent affine loop nests that are meant to replace the recursive code parts.

One may think that analyzing the code statically will be sufficient to model recursive codes as loops. This is true, however statically recursion-equivalent loops are not guaranteed to be polyhedral model-compliant and most probably they are too complex. The recursive and impacting functions often envelop various branches including conditional branches which will remain in the code after applying recursion elimination at compile-time. Accordingly, such loops cannot be aggressively optimized.

Yet, it is possible that the recursive calls and the branches in the impacting functions may be executed in a uniform manner during the whole execution; i.e., the same basic blocks may be visited again and again for all or a significant number of activations/calls of the impacting functions at run-time.

In such a code, if we forsaw its execution behavior, we would be able to rewrite the code even without if-statements in the first place for instance. However, such recursive program behaviors are not taken into account in earlier studies, and they may only be found out at run-time. In comparison, our work considers the actual dynamic control and data flow. Thanks to Rec2Poly's profiling technique and NLR, we are potentially capable of discovering interesting behaviors of recursive codes; this allows rewriting recursions as actual affine loops apposite for sophisticated optimizations.

4.4 Code Generation Phase: Part Inspector

Given a refined expanded and globalized version of the target recursive code and its static analysis information, and based on the affine loop model generated by NLR in the offline profiling phase, Rec2Poly can now proceed with code transformation and generation.

Since the code generation is based on offline profiling of a previous program execution, the correctness of the final code generated is expected to be verified during the next program executions. For this purpose, Rec2Poly generates the final code based on an inspector-executor mechanism. This mechanism has already been presented in Chapter 2 in Section 2.2. Although it has exclusively been utilized in the realm of iterative codes and for verifying their memory accesses, as it is the case in Apollo, Rec2Poly extends this mechanism and brings it to the realm of recursive codes and control verification.

This section is devoted to the inspector code generation part of the last phase of Rec2Poly, the code generation phase. As follows, we thoroughly present the inspector and its constituting components one by one. We explain their functionalities and discuss in details the transformations that Rec2Poly needs to apply to the LLVM IR of the given recursive program for building the suitable inspector. The executor part is discussed afterwards.

4.4.1 Fast Parallel Inspector

Normally, the inspector is a lighter version of a code that proves the execution correctness and collects necessary information required by the executor which corresponds to a speculatively transformed optimized version of the initial code. Uniquely, the inspector that is generated by Rec2Poly involves light recursive code parts; accordingly, its role is to achieve these two tasks:

1. proving that the affine loops, which are ought to replace the recursive code parts, are still congruent with the current dynamic control and data flow, i.e., when the recursive program is run given a different input; this requires verifying both of the memory accesses and the flow control of these structures;

2. instantiating the necessary specific and basic run-time parameters.

In order for Rec2Poly's inspector to fulfill its role, it is constructed from three major types of components which are:

1. ***Trace generator*:** it is a minimal version of the original impacting functions, committed to producing the same control and memory execution trace as the one that was generated by the instrumented code at the profiling phase;

2. ***Verifier***: it is a function constructed from the NLR affine loop model whose task is to check if a trace generated by a trace generator is still compliant with the loop model;

3. ***parameter saver***: it is a minimal version of the original impacting functions whose mission is to record their basic run-time parameter values used by the impacting instructions.

Obviously, trace generators along with verifier components are utilized to carry out the first task of inspectors whereas parameter savers accomplishes the second one.

The inspector must be significantly faster than the original recursive program, so that the final inspector-executor couple provides interesting speed-ups. Generating one whole trace of complete tuples of values, similar to the one generated at the profiling phase, would be generally too costly. We must also not neglect the cost of verifying the whole trace at once too.

Hence, in order to guarantee a fast trace generation process, Rec2Poly must construct an inspector made of multiple trace generators and verifiers that will be executed in parallel each by a distinct parallel thread. Each of the generators is responsible for generating one sub-part of the trace. For each trace generator, there is a devoted verifier verifying the generated trace. Accordingly, Rec2poly is expected to tackle a load balancing issue among threads by deciding how many and which basic blocks or memory accesses a single trace generator must handle.

Also, Rec2Poly must reserve a thread for the parameter saver. In addition, it must dedicate another one for executing the original recursive code part, preserved since the code preparation step, that should run simultaneously with the trace generators and verifiers. This is important to halt the inspector and proceed with a correct execution without extra time-overhead in case a misspeculation is discovered at run-time.

Note that, all the functions within the inspector are not supposed to write to a shared memory location, so a parallel execution will not cause any conflict or race condition among threads.

In what follows, we present the components of the inspector and how they are generated by Rec2Poly.

4.4.2 Trace Generators

A trace generator is made up of a set of light minimal copies of the impacting functions, *i.e.*, source, recursive and reachable functions. Its role is to produce a trace representing either the actual control flow or the memory behavior of the current execution. For every trace generator to create, Rec2Poly must clone the impacting functions and their basic blocks and associate a new invocation counter for every cloned function. It may also create in case needed, for every trace generator, different copies of the existing global variables and use these copies instead, in the trace generator functions. The (initially local) globalized variables copies in the cloned functions must be referenced using the invocation counter corresponding to that clone.

Besides, a trace generator is expected to output a trace, so the latter can be verified against the NLR affine loop model. For this sake, Rec2Poly creates, in the IR module,

global memory buffers or arrays and global indices to be used by the trace generators for referencing their dedicated buffers.

Inside the cloned functions, the call instructions still call the initial impacting functions; so, Rec2Poly must replace these referenced functions in the call instructions by their proper clones.

Then, Rec2Poly creates a thread function that invokes the trace generator through the initial source function clone. In case the recursive code is not invoked within a loop, *i.e.,* the source is empty, the thread function must initiate its associated trace generator by calling the clone of the initiating recursive function, *i.e.,* the recursive function called by a non-recursive one in the program. As we mentioned earlier, we assume that the recursion handled is only initiated through one specific recursive function or through a unique set of source functions all participating in a sequence of calls reaching the recursion.

For every trace generator, Rec2Poly creates a thread launching the corresponding thread function replacing the initial call to the initial source function or the recursive function. Similarly, all of threads must be joined at the end. Also, Rec2Poly dedicates two semaphores for every trace generator-verifier couple used to control their access to their shared buffers of traces. Semaphores functionality in the inspector is presented later in this section in more details.

Afterwards, Rec2Poly must make the clones, constituting the trace generators, the lightest possible, *i.e.,* by removing expensive instructions that involve touching the memory such as stores and loads that do not affect the execution flow. Then, Rec2Poly modifies these clones further and adds the instructions that permit a trace generator to generate and save its own part of the trace to be verified later. Note that the verification process is interleaved with the trace generation; this is discussed later in this section. Before modifying the code, Rec2Poly needs more analysis information regarding the newly cloned functions.

Extended Code Clone Analysis

When Rec2Poly performs a function clone corresponding to a trace generator, it only knows that this cloned function corresponds to an impacting function. Yet, it does not have any information about the contents of this new function. Does it have impacting basic blocks that must be traced? What are the necessary instructions that must be kept? What is the relative memory address accessed by a store/load? Rec2Poly must know the answers to such questions before deleting/adding any instruction from/to the clone. This is achieved by performing an extended analysis which is described in the following.

Impacting Basic Blocks Clones Determination As Rec2Poly previously determines the impacting basic blocks of the original impacting functions, it, similarly, identifies the impacting basic blocks clones corresponding to every trace generator.

Control-Conserving Instructions Identification Rec2Poly needs to determine all the instructions that are crucial to preserve a correct control flow. The obvious primary control instructions can be branches and function calls. Yet, these instructions may depend on other instructions that must also be considered, e.g., conditions for conditional branches and arguments computations for function calls, and loops-related instructions,

e.g., loops iterators increment instructions. In order to identify all such instructions, Rec2Poly finds the BSS of the branch instructions. Note that, in this study, we make the assumption that conditional branches do not depend on any memory accesses.

Memory-Address-Maintaining Instructions Identification In trace generators, the memory trace will be produced; so, all the instructions that are required to compute a memory address touched by a load or a store, *i.e.,* its BSS, must be identified.

Clones Memory Behavior Analysis As Rec2Poly performs the memory behavior analysis as described in Section 4.2, it performs this analysis again, for the trace generator functions, to associate every memory address with its corresponding base address. As the relative memory offsets are the values instrumented, they must be the values which are verified.

Based on this analysis, Rec2Poly refines the functions clones, at the LLVM-IR level at compile time, to construct the desired and customized trace generators.

Trace Generators: from Copies to Originals

Trace generators are customized based on their dedicated task, i.e., which part of the trace they are intended to generate. In this regard, we distinguish between the two types below of trace generators:

Control Trace Generator (CTG) It is a trace generator which is dedicated to generate the dynamic control flow of the impacting functions, which is mainly described as a sequence of basic blocks IDs executed at run-time. There exists only one CTG in the inspector. The CTG functions must mimic the original functions execution flow. So, Rec2Poly only keeps the control-conserving instructions and the invocation counter increment instructions and removes the rest. In every impacting basic block, it adds instructions that save the ID of that basic block, when executed at run-time, in the trace buffer dedicated to the CTG.

Memory Trace Generator (MTG) It is tailored for generating the touched relative memory addresses at run-time. The inspector usually includes multiple MTGs. Yet, every MTG still generates a different part of the memory trace. The workload assigned to an MTG must be almost the same as that of the CTG. Accordingly, as the CTG accesses the buffer once per basic block to generate its ID, an MTG obviously should generate one memory access per basic block at most. In MTGs, all the control-conserving instructions and the memory-address-maintaining instructions for the memory accesses of interests are preserved. The memory accesses themselves are deleted. Then, in each MTG function, the necessary instructions, needed to save its share of computed addresses into its dedicated buffer, must be inserted by Rec2Poly. Nevertheless, there is an exception for the globalized data structures: they are ignored. This is because their memory offsets equal the values of the invocation counters. So, it is enough to trace the functions invocation counters values at run-time. Accordingly, MTGs need to save the values of the function

invocation counters. In addition, MTGs handle the existing loops induction variables values; they save for each loop, the value of its induction variable per iteration. It is possible to assign a separate trace generator to produce the function counter and loop induction variables values instead of taking care of them along with the memory addresses; such a trace generator is constructed similarly as any other memory trace generator.

Accordingly, each set of the functions clones, i.e., each trace generator, is differently handled and modified.

4.4.3 Verifiers

For every trace generator, Rec2Poly creates a corresponding trace verifier based on the NLR affine loop model in the LLVM IR of the program. As it is the case for trace generators, we similarly differentiate between the following verifier kinds:

- **Control Verifier (CV)** It is a verifier responsible for verifying the control trace generated by the CTG; Rec2Poly generates one CV.
- **Memory Verifier (MV)** It is a verifier associated with an MTG, responsible for verifying the particular memory addresses and values traced by the MTG; there is one MV created for every MTG.

Each verifier is generated as a new thread function that implements the NLR loops using minimal versions of the constituting basic blocks. The basic blocks are created with only the branch instructions and the constructed affine loop increment and condition instructions. Then, in every verifier, Rec2Poly must add the necessary instructions to access and verify the values stored in the buffers that this particular verifier shares with its associated trace generator. The CV verifies the basic block IDs generated by the CTG, and the MVs verify the corresponding memory offsets and the invocation counter and induction variable values saved to the buffers against the associated values captured by NLR.

Then, the trace verifiers are launched in parallel threads with the trace generators during the execution.

However, although the original recursive code is supposed to involve expensive memory accesses, the supposedly light trace generators and verifiers, as described here, may not allow the inspector to be sufficiently fast. In case of having a lot of memory accesses in the program, there may be too many threads, MTGs and MVs, to create; this may induce a significant overhead to the program. Conversely, in case the original recursive program includes only one access at most per basic block in its impacting functions, the memory accesses performed by the inspector to verify the traces will not be any cheaper. Yet, further inspector optimizations are possible and the options that we depend on in Rec2Poly are discussed later in this section.

4.4.4 Parameter Saver

As mentioned earlier, all memory accesses, global variables and local variables that are globalized are instrumented at the profiling phase. The local variables that are not globalized are those that are used to compute the memory addresses, so they are tracked along

with memory accesses. The NLR loop model obtained, whenever successful, includes all this information which guides the recursion-to-loops transformation. Yet, there is still a missing piece for the loops to be successfully generated. The function parameters that are not of a pointer type may not be taken into consideration in the profiling phase. Such parameters may be passed as arguments to the impacting functions and used directly by their instructions through sequences of calls by non-impacting functions in which they are local variables and out of the scope of globalization. Accordingly, Rec2Poly can neither get a prediction or prior information about these parameters nor extract their values at compile time to complete the code generation phase. Thus, in order to fill the blanks in the loops generated, Rec2Poly needs to record the values of such parameters during the program's execution within the inspector before initiating the actual loops.

For this purpose, Rec2Poly creates, in the LLVM IR of the program, the parameter saver which is composed of a minimal version of the impacting functions clones. The parameter saver functions, like those of the trace generators, require having only the control-conserving instructions and the instructions incrementing their corresponding copies of the invocation counters; all the other instructions are deleted. Also, Rec2Poly creates a global buffer for every important function parameter in the parameter saver. Then, Rec2Poly adds the instructions, at the entry basic block of every function in the parameter saver, needed to collect the function's input values and store them in the corresponding buffers/arrays indexed by the particular function's invocation counter.

The parameter saver is also supposed to be executed simultaneously with trace generators and verifiers. So, Rec2Poly, in the LLVM IR, initiates the parameter saver through a thread as it does for the trace generators and the verifiers.

Figure 4.5 shows the call graph of the inspector corresponding to the arbitrary example of the recursive program of Figure 4.2.

The impacting functions for which Rec2Poly creates the light clones to build the inspector are: A, B, C, D, E, F, G and H.

This figure shows that the main function launches $M+1$ parallel threads. The $M+1$ threads needed by the inspector belong to one parameter saver, $M/2$ trace generators, such that one is a CTG and the rest are MTGS, and $M/2$ verifiers. Note that, for every thread that initiates a trace generator, there is a thread that initiates a verifier function.

The parameter saver and the $M/2$ trace generators require $M/2+1$ different minimal light clones of the impacting functions. The i^{th} functions clones are referred to as : A_l^i, B_l^i, C_l^i, D_l^i, E_l^i, F_l^i, G_l^i and H_l^i.

The $M/2$ verifiers require creating $M/2$ light functions, each of which is constructed using light basic blocks based on the NLR model. The i^{th} verifier function is called v^i; v^i is assumed to be constructed from $L+1$ light impacting basic blocks: BB_0^i, BB_1^i,..., BB_L^i. In the verifier functions, the control flow graph composed of nodes representing the basic blocks is visualized; in this graph, the constructed loops appear as cycles.

Each trace generator and its associated verifier have their own set of $N+1$ buffers to process on, and their accesses to the buffers are managed using two semaphores. Accordingly, the inspector requires M semaphores. For instance, Thread 1 and Thread 2 communicate using Semaphores sem 0 and sem 1. Each trace generator produces and saves its own trace part using its dedicated buffers which are verified by the associated verifier.

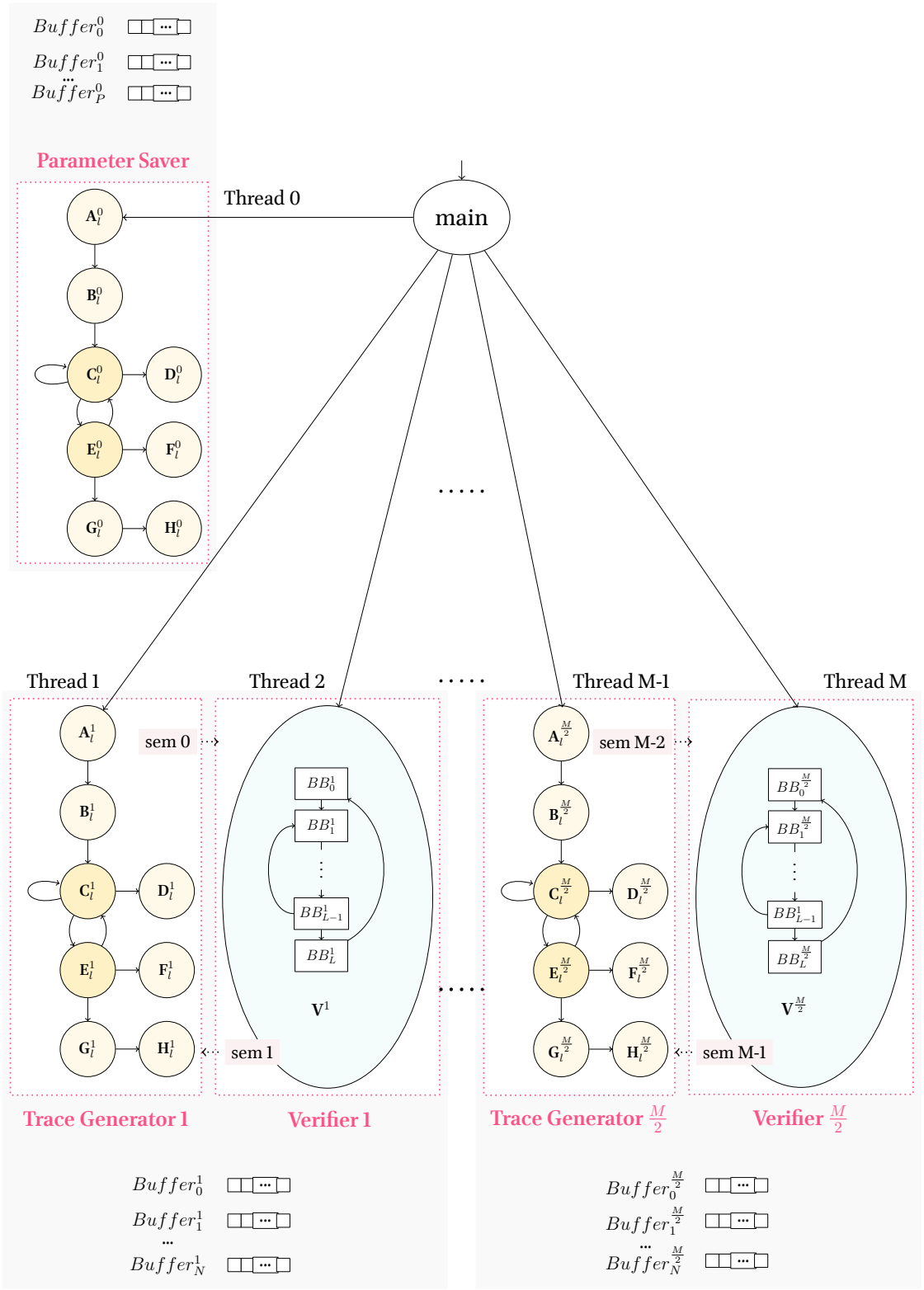


Figure 4.5 – Detailed Inspector Call Graph Example

On the other hand, the parameter saver has its own set of $P+1$ buffers used for saving the parameters values of the impacting functions clones.

4.4.5 Inspector Optimizations

The inspector, especially its constituting trace generators and verifiers, may remain insufficiently fast.

Yet, there are still several interesting optimization opportunities that can be exploited and applied to Rec2Poly. They involve limiting insignificant predictable repetitions of certain control structures and basic blocks and eliminating memory addresses that do not need to be verified besides the globalized-initially local-data structures.

Control Flow Optimizations in the Control Trace Generator and Verifier

The control-related optimizations must reduce the size of the traces corresponding to basic blocks. Since, we are highly interested in optimizing recursive codes that also involve expensive loops, such optimizations are exclusively dedicated to the already existing loop structures, for-loops in particular. In LLVM IR, the exit condition, body and increment of a for-loop are expressed as distinct basic blocks: `for.cond`, `for.body` and `for.inc` respectively as shown in Listing 4.3. Every time a loop iterates, all of its three corresponding basic blocks execute. When the upper bound is met, as checked in `for.cond`, the loop exits to execute the `for.end` basic block. Also, as we know, the precise number of iterations of a for-loop can be pre-computed without the need to actually execute the loop in case the loop bounds and index are not modified inside the loop body. This requires knowing values of the lower and upper bounds and the increment. In this case, even if the loop information and parameters are not all known at compile-time, as soon as they are discovered at run-time, the loop's number of iterations can be computed.

```
...
for.cond:                                         ; preds = %for.inc, %entry
  %i.0 = phi i32 [ 0, %entry ], [ %inc1, %for.inc ]
  ...
  %cmp = icmp slt i32 %i.0, 10
  br i1 %cmp, label %for.body, label %for.end

for.body:                                         ; preds = %for.cond
  ...
  br label %for.inc

for.inc:                                          ; preds = %for.body
  %inc = add nsw i32 %i.0, 1
  br label %for.cond

for.end:                                          ; preds = %for.cond
  ...
...
```

Listing 4.3 – For Loop at the Level of the LLVM IR

Rec2Poly achieves its first trivial optimization, in this context, by dismissing impacting

```
void foo() {
  for (int i=0; i<N; i++)
        printf("foo::BB1\n");
}
```

(a) Loop Example

```
val foo::BB1
val foo::BB2
```

(b) NLR Output for N = 2

```
for i0 = 0 to 2
  val foo::BB1
```

(c) NLR Output for N = 3

Figure 4.6 – For-Loop and its NLR Trace

basic blocks that are always executed with other blocks, the same blocks all the time, and that are almost totally modified or reconstructed for the executor, e.g., the condition basic blocks of a loop in case they are impacting.

Another optimization by Rec2Poly is dedicated to affine for-loops that do not involve any conditional statements or function calls, and whose execution within an active function is predictable, and body instructions do not have any impact on the control flow outside the loops.

As we have presented, the verification of the control flow is managed by the CTG and the corresponding verifier; in the CTG, the ID of the impacting basic blocks must be saved into buffers that must be verified by the verifier. However, some of these basic blocks may be for-loop-related basic blocks. Knowing that such basic blocks are going to execute, as they are, again and again, it seems unnecessary to verify the same basic block ID over and over again within an activated function. However, such basic blocks must be re-verified if the function is called again with different parameters.

Accordingly, executing such loop-related basic blocks at least once, computing the number of loop iterations and verifying this information against the NLR-equivalent loop information, should be ample to ensure that the control flow of this structure is correct. Yet, NLR does not form loops for values that are repeated less than three times; if a loop happens to execute two times at run-time during the offline profiling phase, its basic blocks will appear two times consecutively without a loop in the NLR model (see Figure 4.6). So, if the CTG saves the loop basic blocks IDs once and the number of iterations as two, and, in the verifier, the basic block ID is expected to exist twice; this will signal a misspeculation which is a false positive.

For this purpose, Rec2Poly constructs such loops in the CTG and the corresponding verifier to iterate three times at most; this saves much time-overhead. Instead of verifying the blocks IDs only, the CTG and the verifier are expected to handle the number of loop iterations too. However, a dedicated extended static analysis must be performed by Rec2Poly before changing the upper bounds of the corresponding loops. It must make sure that this modification does not affect the overall control flow; no values changed or instructions executed within these loops are needed by the succeeding branch instructions outside the loops. Whenever such optimizations succeed, the control trace generator will become much lighter and faster.

For example, in the recursive program example in Figure 4.2, the loop existing in function A from which the recursion is indirectly initiated, cannot be optimized. However, if there exists another for-loop in function H (see NLR loops in Figure 4.3), which satisfies all the conditions for this optimization, it will be built in the verifier and the original loop copy will be modified in the CTG to iterate three times instead of twenty five times.

Memory Access Optimizations in the Memory Trace Generators and Verifiers

The correctness of the memory addresses accessed in the final optimized code must be ensured; for this reason, Rec2Poly generates the memory trace generators and verifiers as we have presented. It would have been ideal to verify every single memory access; but, this can be a huge burden on the inspector to achieve robustly and quickly enough to leave time for the executor later to perform its sophisticated optimized loops and shine.

In codes where there are interesting multi n-dimensional arrays that are manipulated through a recursion, every single access to these array elements may yield n memory accesses at the level of the LLVM IR of such codes; each of these n accesses must be handled and verified by memory trace generators and verifiers. This may require creating many more trace generators, verifiers and threads which induces overhead. Yet, if an array is statically allocated, an access to it requires only one memory instruction and the address of its elements can be directly computed at once even if it is n-dimensional. However, if the array is dynamically allocated, and so an access to it yields n LLVM IR memory instructions, although we can assume and hypothesize that the address can be correctly directly computed at once too, this remains an assumption that may lead to false positives and negatives in the run-time verification which should be avoided.

A more reliable optimization is to ignore redundant memory accesses in an impacting basic block and verify only the principal accesses only. If a certain array is touched several times through indices based on the same induction variable, then only one of these accesses is worth the cost of the verification. This may also work among different data structures if they are multi-dimensional in particular. However, in such situations, their first relative addresses, w.r.t. the initial base addresses of the data structures themselves, must be verified even if accessed using identical indices. For example, consider an array `A` accessed three times in this statement `A[i]=A[i-1]+A[i-2]` within a loop. A classic inspector would generate memory traces (offsets) for the three memory accesses, and save each of the offsets: `i`, `i-1`, and `i-2` per iteration. But, verifying them all is pointless and costly; it is adequate to save and verify only the offset of $A[i]$. Another example includes accesses to array elements `A[i][j]`, `B[k][j]` and `C[i][j]`. Instead of verifying both of `i` and `j` for `A`, `k` and `j` for `B` and `i` and `j` for `C`, it would be enough to verify `i` for `A`, `k` for `B`, `i` for `C` and `j` only once, taking into consideration the types of the different data structures.

Another optimization can be activated for the MTGs and the memory verifiers which is similar to that applied to the CTG and its verifier. That is, whenever there is an affine for-loop accessing multidimensional arrays that does not involve instructions that affect the control flow and the computation of other memory addresses outside the loop, the loop can be modified to iterate thrice at most with every new activation of the parent function. As we mentioned earlier, the conditional statements are not supposed to depend on a memory access. Therefore, in the MTGs and their verifiers, within such loops, every access to a data structure indexed by an affine expression of its induction variable, is verified three times at most. If the verification of the first relative memory address accessed in the loop is successful, the predicted address remains correct for the next two iterations, and the loop induction variable is expected to change linearly within a specific verified range. This should be sufficient to ensure that the rest of the memory accesses will still be respected in the transformed code. For a loop nest of depth n, the memory addresses touched in the innermost loop will be verified at most 3^n times. The MTGs and

the verifiers may become much faster and more performant. The number of iterations is handled by the CTG and its verifier.

An example of optimizable loops is shown in Listing 4.4. Modifying the upper bound of the first loop will surely affect the values saved in array `A` in function `foo`, but not the control flow of the second loop and the rest of the function. Also, the second access to array `A` in the second loop does not depend on anything executed within the first loop. Hence, it is possible to optimize the first loop in the trace generator and its equivalent-loops in the verifier. Similarly, the second loop can be optimized too. Other examples are shown in Listings 4.5 and 4.6. In Listing 4.5, the first loop changes the value of `j` on which depend the memory access to `A` in the second loop. So, the first loop cannot be optimized, but the second loop can be. In Listing 4.6, the existing loop must not be optimized because it includes an instruction that affects the if-condition after the loop exits.

```
void foo() {
  ...
  for (int i = 0 ; i < N ; i++)
        A[i] = x;
  for (int i = 0 ; i < M ; i++)
        A[i] = x;
}
```

Listing 4.4 – Example of For-Loops Optimizable in the Inspector

```
void foo() {
  ...
  int j = 0;
  for (int i = 0 ; i < n ; i++) {
        A[i] = x;
        j += 2;
  }
  for (int i = 0 ; i < m ; i++)
        A[i+j] = x;
}
```

Listing 4.5 – Non-Optimizable Loop

```
void foo() {
  ...
  int j = 0;
  for (int i = 0 ; i < n ; i++)
  {     A[i] = x;
        j += 2;
  }
  if ( j > x )
        ...
}
```

Listing 4.6 – Non-Optimizable Loop

4.4.6 Inspector In Action: Verification Process

After exhibiting how an inspector code is generated at compile time, we present now how it works at run-time, for its verification technique in particular. As mentioned before, all trace generators, verifiers and the parameter saver are run in parallel such that every trace generator interacts with its corresponding verifier to handle a particular part of the trace. In order to understand the whole verification process, one must understand how the verification is performed at the level of every generator-verifier couple.

The run-time interaction between a trace generator and its associated verifier is illustrated in Figure 4.7, and it is explained in what follows.

The verification process within the inspector is performed in a pipelined fashion be-

tween a trace generator and its corresponding parallel verifier. A trace generator fills sequentially several buffer arrays with the values to be verified. A verifier verifies the buffers contents in parallel with the generator.

Trace Generator in the Lead

Although both a particular trace generator and its verifier are launched simultaneously, the prior must fill at least a buffer, so the latter can take action and start verifying the saved values in the buffer.

The generator must not generate the whole trace before the verifier performs its task. Such a scenario does not allow parallel execution between the generator and the verifier. For that reason, the trace generator must only generate a partial trace that is, then, handled by the verifier while the generator is working, in parallel, on generating the following part of the trace. The size of these partial traces can be adjusted to get the best trace-generator verifier performance.

We note that there are principal points or basic blocks in both the generator and the verifier in which their interaction and synchronization are controlled. They are known as trace initiating or trace terminating basic blocks which are identified according to the NLR loop model. There may be more than one of each kind of such basic blocks in each of the trace generator and the verifier on condition that they are in corresponding positions in both of them. The basic blocks with which the NLR model begins are considered first trace initiating, the ones with which the model is supposed to end are trace terminating. At least one of the initiating and the terminating basic blocks must determine the scope of the major loop in the NLR model. For instance, if the NLR model shows a loop nest such that the outer loop already exists in the code, then the first basic block of this loop clone in the corresponding trace generator function and that of the constructed equivalent loop in the verifier function are initiating basic blocks. If the NLR outer loop nest does not exist in the code, so it may correspond to a repetitive function call; then, the basic block in which the corresponding function, i.e., the looping code part, is invoked is the initiating basic block. On the contrary, the last basic block in the loop and that with which the NLR model ends are terminating basic blocks. For example, if we reconsider the NLR model of Figure 4.3, the initiating basic blocks are, `BB1` and `BB2` of A, and the terminating basic block is `BB3` of A too.

The generated trace part cannot be verified until a terminating basic block is reached by the trace generator. The verifier waits in the initiating basic block for the dedicated shared trace buffer to be filled before it performs any verification.

Also, note that the trace generator includes all the basic blocks clones of the initial recursive code, so it has both impacting and non-impacting basic blocks. However, the generators must only trace the impacting ones. As for the verifier function, it mainly includes the set of the impacting basic blocks.

Accordingly, at the beginning, when the trace generator and verifier are launched, the first basic blocks visited in both are the very first initiating basic blocks. At this point, the generator accesses the first buffer dedicated to it to start its trace production. On the other hand, although the verifier knows that it needs to process on the first buffer shared with the generator, it must wait to make sure that the latter finishes saving the first part of the trace. As we already mentioned, the couple threads communicate using semaphores, so the verifier waits for a semaphore to be posted by the generator when saving the trace

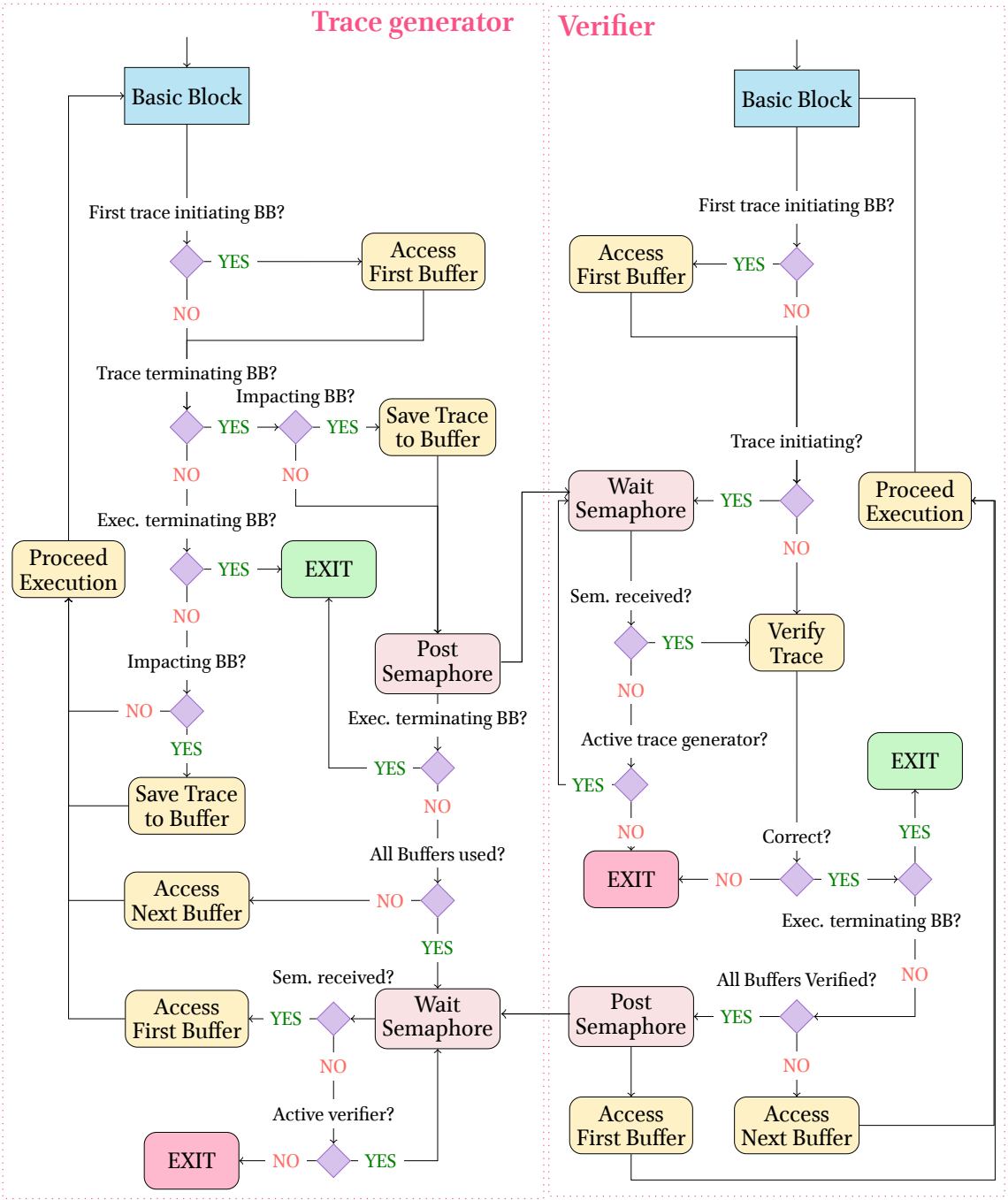


Figure 4.7 – Trace Generator-Verifier in Action

part into the current buffer is successfully done. With every initiating basic block visited, the verifier must wait a semaphore from the generator. In the meantime, in the trace generator, as long as its execution does not terminate, if the basic block visited is not trace terminating but impacting, then the desired trace corresponding to this basic block (ID or memory address) is saved to the current buffer. If the basic block is not impacting, then it is simply ignored, and the trace generator proceeds to execute the next basic block. If the basic block is trace terminating, and if it is impacting, then the generator must also save it into the buffer; if it is not, it is ignored. Then, the generator posts the semaphore to the verifier so the latter can start verifying the saved trace values.

Trace Generator-Verifier Embracing Parallelism

When the verifier receives its first semaphore, it starts verifying the generated values in the first buffer one by one with every basic block visited in it. As long as the trace is correct, the verifier proceeds its execution as long as its execution is not supposed to normally terminate.

In parallel, the trace generator continues its work and fills more buffers as long as there are available dedicated buffers, and it notifies the verifier again and again with every terminating basic block, i.e., when a buffer is filled with trace values. When all buffers are full, the generator must wait for a semaphore back from the verifier. This is just to prevent the trace generator from corrupting the buffers by refilling them again with new values before the old ones are completely verified.

The verifier keeps receiving the semaphore at the level of every initiating basic block executed as long as the trace generator is active, and verifying the values in the next buffers until there are no more buffers left to be verified. Then, the verifier needs to post a semaphore back to the trace generator, so the latter can continue its work.

No Misspeculation Missed

If the trace generator finishes its work and terminates its execution in a normal state, and the verifier does not detect any wrong trace and terminates right after the trace generator, then, this particular part of the trace handled by this couple is proved to be valid. When all trace generators and verifiers exit normally, then the whole speculation is approved, the original recursive code running in parallel is stopped and the reliable executor can be launched. However, if at least one couple terminates abnormally due to a defect detected in the predictive model w.r.t. the current execution by the verifier, thus there is a misspeculation.

A misspeculation at the generator-verifier couple level is detected and handled as follows.

It is possible that the trace generator finishes its execution naturally while the verifier remains active even after it finishes verifying all the filled buffers. This means that the predictive NLR model, on which the verifier is based, involves more iterations which must be faulty w.r.t. the actual execution. Consequently, the verifier may be blocked waiting forever for a semaphore that will never be posted by a terminated generator. In such a situation, the verifier must be stopped flagging a misspeculation.

On the other hand, such a situation may occur in the opposite direction. The verifier may detect a wrong trace value and eventually exit, or it may terminate because it must

terminate according to the prediction model. Yet, even if the trace generated till then is valid, if the trace generator is still active, this implies a misspeculation. When the verifier terminates, the generator must exit instead of waiting infinitely for a semaphore from the terminated verifier.

When the inspector disproves the predictive model, the executor is prevented from executing; instead, the original recursive code carries on with its execution on which the final output of the program will depend.

Beyond the Verification Process: a Race to Win

Even if the inspector does not detect a misspeculation, if the original code terminates its execution before the inspector-executor, then the latter must be aborted. In some contexts, the inspector, in particular, may induce much overhead and remain slow in comparison to the original recursive code; this may yield a costly inspector-executor code. For this reason, as the inspector-executor and the original code backup are launched in parallel during the optimized program execution, they are in a race. The first one to terminate, is the one whose output must be taken into account; the other one is ignored and aborted.

4.5 Code Generation Phase: Part Executor

Whenever, at run-time, the inspection is successfully completed and all the inspector threads safely terminate, the original recursive code that has been running simultaneously in the background is halted and the execution of the parallel optimized loop, a.k.a. the executor, is launched.

In this section, we explain how this executor code can be generated by Rec2Poly at compile-time. We first show how the loops are constructed and then how possible optimizations can be applied.

4.5.1 Loops: from Design to Construction

Rec2Poly modifies the globalized and expanded LLVM IR of the recursive program and builds into it the executor. The executor is first built as a function made up of sequences of loop nests and their basic blocks. The NLR affine loop model is the design that guides automatically, or even manually, the construction of this iterative code part. The resulting code is considered semantically equivalent to its initial recursive counterpart.

Loop Model Information Extraction

Rec2Poly must take the NLR loop model as an input and extract all the information about all NLR loops, traces and the affine memory functions. For every loop, it must obtain the following necessary information:

- the loop ID within the model
- the depth

- the upper bound; the lower bound and the increment step of the NLR loops are always zero and one respectively
- the parent loop ID
- the loops and traces IDs enclosed within this loop in order of occurrence

As for the control traces, it gets the information below:

- the trace ID within this model
- the ID/name of the impacting function that initially produced this particular trace in the offline profiling
- the ID of the impacting basic block corresponding to this trace
- the invocation counter expression if the trace corresponds to a function's entry
- the list of all relative memory addresses affine expressions accessed, in order of occurrence, within the original basic block
- the affine expressions corresponding to the values of the inductions variables of the loops already existing in the original code.

As for the affine functions of the memory offsets, Rec2Poly must parse these functions and identify the loops whose induction variables are part of this expression in addition to the associated coefficients. If the loop model is parametrically affine, then Rec2Poly needs to extract all the list of the possible coefficient values associated with a variable.

When all this information is extracted, code rewriting begins.

Create, Clone and Make a Difference

Rec2Poly creates an executor function and inserts a call instruction to it right after the inspector threads are all joined in the LLVM IR.

Then, the control flow graph of this function is constructed; each node, each constructed basic block corresponds to either a NLR for-loop or a NLR trace line; a trace line is the NLR tuple including the basic block ID and the list of affine expressions corresponding to memory accesses. First of all, a function entry basic block must be constructed.

A loop is represented as a sub-graph composed of a maximal subset of the control flow graph nodes, corresponding to basic blocks, forming a strongly connected component (SCC). Within such a SCC, there exists an entry node dominating all the other loop-related nodes [79]. If the NLR model starts with a loop, then the function entry block is assigned to be the loop preheader. In case the loop is preceded by a trace line, then the basic block corresponding to this trace is the loop preheader. The loop preheader must always branch to the loop entry node. Also, a loop iterator variable is allocated dedicated to this loop. In the loop entry, the loop iterator is checked if it is still less than or equal to the NLR upper bound. If so, it branches to a basic block representing the loop body. Else, it branches to a loop exit basic block which, in turn, branches to the succeeding NLR loops or traces corresponding basic blocks.

The loop body basic block either branches to inner NLR sequences of loops or traces basic blocks; the last of these inner basic blocks must branch to a basic block in which the loop iterator is incremented by one. The basic block responsible for the loop increment must perform a jump back (backedge) towards the entry loop block.

As for the traces basic blocks, Rec2Poly maps every NLR control trace value/ID to its corresponding basic block in the LLVM IR. Rec2poly creates a corresponding basic block in which it clones all the corresponding enveloped impacting instructions. For every occurrence of a basic block ID appearing in the NLR trace, a basic block must be constructed. Accordingly, the impacting instructions must be cloned again as many as required. However, the branch instructions must not be cloned. New branch instructions are created and added to the basic blocks in the executor functions to branch to the succeeding newly constructed basic blocks according to the NLR model. The operands of the instructions must be modified in order to fit in the new function.

In case we arrive as far in code generation, this means that the recursive code is affine behaving; thus, the values of the inserted functions invocation counters are also expressed as affine functions of the surrounding loops. These affine expressions must be translated as instructions using the created loops iterator variables in the executor function. Accordingly, the uses of the non-pointer parameters in the function must be replaced by the values recorded by the parameter saver in the dedicated buffers indexed using the counters affine expressions. Similarly, Rec2Poly manages the accesses to the globalized variables.

As for the memory access instructions per created basic block, they must be replaced by accesses to the original associated base addresses using the NLR affine expression in terms of the surrounding loops iterators. In addition, the uses of the induction variables of initially existing loops must be replaced by the corresponding NLR affine expressions in terms of the NLR loops.

The control flow graph of the executor function generated for the recursive program whose behavior is modelled by NLR as shown in Figure 4.3 (or 4.4) is illustrated in Figure 4.8.

The executor must be optimized by the classical LLVM passes that, for example, simplify the control flow, perform dead code elimination and promote memory to register references. Finally, further powerful loop optimizations can be applied to the beautifully constructed loops in the executor function.

As mentioned in Section 4.3.2, when successful, NLR may produce two types of interesting loop models:

1. a model exhibiting a fully affine control and memory behavior
2. a model that is parametrically affine

Rec2Poly has to handle and optimize each of these models differently. A different optimization approach must be adopted and applied depending on the type of loops obtained. This is discussed in what follows.

4.5.2 Fully Affine Loop Model Optimization

This is the most favorable case in which Rec2Poly generates perfect affine loop nests. The resulting loops are ready to be optimized using an automatic polyhedral optimizer.

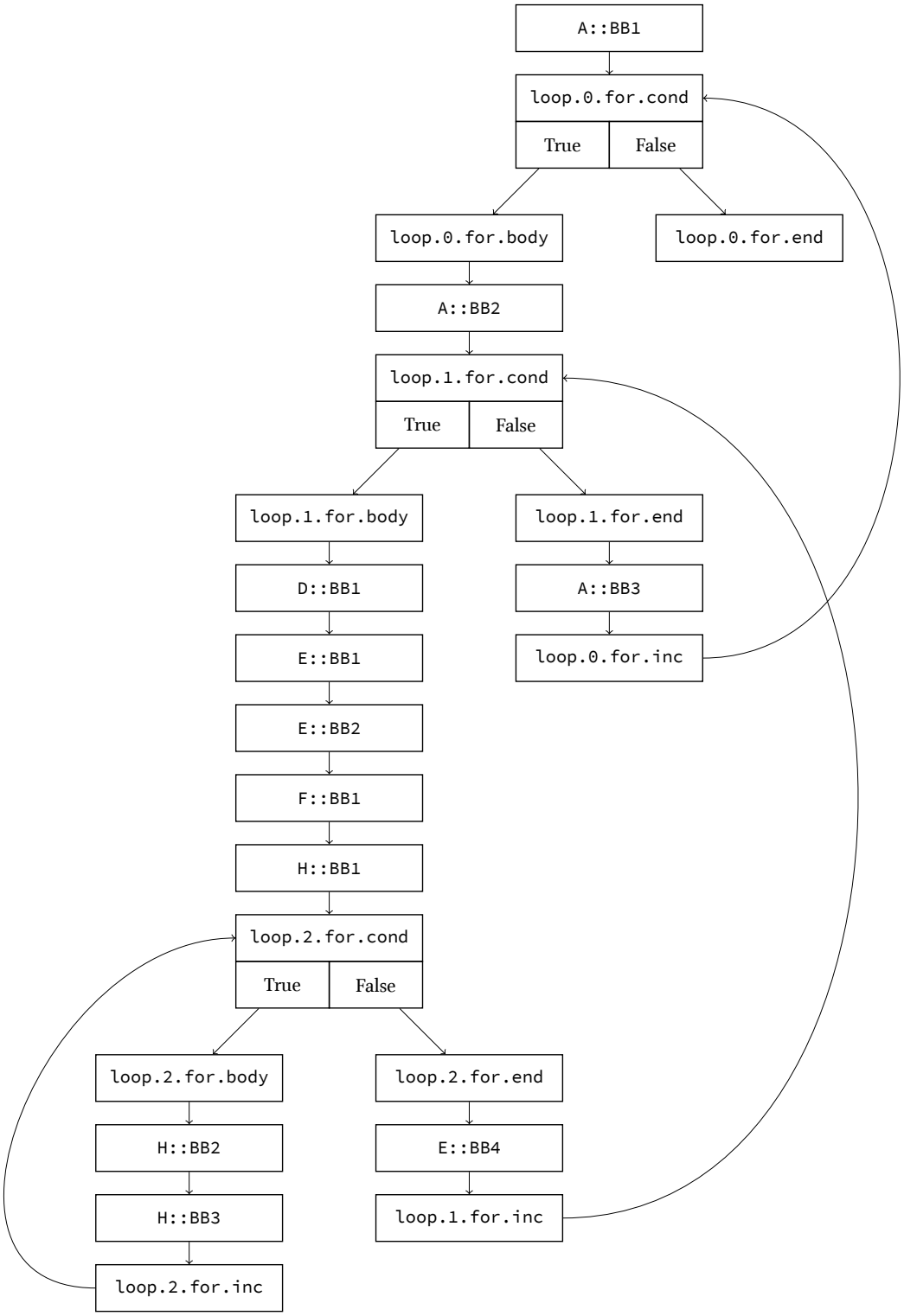


Figure 4.8 – Executor Control Flow Graph Example

However, although the Polly optimizer would have been an obvious pick in this regard, the loops generated by Rec2Poly may encounter the same challenges as those encountered by Apollo's loops with Polly [83]. Our choice in this regard is Pluto. However, since we generate and transform codes at the level of the LLVM IR, and Pluto is not capable of processing at that low level, Rec2Poly needs to feed Pluto with an OpenScop representation [12] of the generated affine loops. Data dependences can automatically be taken care of by Pluto.

An interesting option would be to integrate Rec2Poly functionalities in Apollo which is also based on Pluto. Yet, the run-time verification performed by Apollo at run-time would be costly for the so-generated loops since they do not need to be verified any further.

4.5.3 Loops with Parametrically-Affine Memory Behavior

In this case, unfortunately, polyhedral automatic optimizers cannot be applied because the memory accesses do not fit the polyhedral model.

Nevertheless, efficient loop parallelization can be still achieved which requires a dedicated dependence analysis process. This process consists of computing the ranges of touched memory addresses by store and load instructions at each iteration. This can be easily resolved at compile time, because thanks to NLR, Rec2Poly can pre-compute all the memory addresses for all iterations using the list of coefficients obtained. These ranges are then used to build independent sets/lists of iterations.

Accordingly, this helps breaking the outer loop into two nested loops such that:

- the outer loop iterates over the lists of loop indices values
- the inner one iterates over the indices values inside each list

Finally, the outer loop can be parallelized into parallel threads using OpenMP.

As an illustrative example, consider the parametrically-affine NLR model in Figure 4.9 of the behavior of some instrumented recursive program. The NLR model shows a loop nest in which the outermost loop corresponds to a recursion and not a loop already existing in the code. All the inner loops and the basic blocks, whose IDs appear in the model, correspond to an impacting function that is called "`foo`" whose code is shown in Listing 4.7. We assume that Function "`foo`" already has its invocation counter inserted to the code, which is `foo_invocation_counter`, and has its initially local variable `c` globalized to `c_glob`. Based on the NLR model, we can rewrite the corresponding recursive code as an optimized iterative code using the impacting basic blocks and instructions of Function "`foo`" only.

```
for i0 = 0 to 7
  val foo::entry , 1*i0
  for i1 = 0 to 31
    val foo::for.cond
    for i2 = 0 to 31
      val foo::for.cond1
      val foo::for.body3 , 1*i0
      for i3 = 0 to 31
        val foo::for.cond4
        val foo::for.body6
          , [8:64,96,2112,2144,64,96,2112,2144][i0] + 64*i2 + 1*i3
          , 1*i0
          , 1*i0
        val foo::for.inc , 1*i3
      val foo::for.cond4
      val foo::for.end
        , 1*i0
        , [8:0,0,32,32,2048,2048,2080,2080][i0] + 64*i1 + 1*i2
        , [8:0,0,32,32,2048,2048,2080,2080][i0] + 64*i1 + 1*i2
      val foo::for.inc84 , 1*i2
    val foo::for.cond1
    val foo::for.cond87
    val foo::for.inc120 , 1*i1
  val foo::for.cond
```

Figure 4.9 – Parametrically Affine NLR Model

```
int foo_invocation_counter=-1;

void foo(double * A, double * B)
{
  foo_invocation_counter++;
  for ( int i = 0 ; i < p ; i++ )
    for ( int j = 0 ; j < p ; j++ ) {
      c_glob[foo_invocation_counter] = 0;

      for ( int k = 0 ; k < p ; k++ )
        c_glob[foo_invocation_counter] += A[s*j+k];

      B[s*i+j] = c_glob[foo_invocation_counter]*B[s*i+j];
    }
}
```

Listing 4.7 – Impacting Function of a Recursive Code After Code Preparation

The equivalent iterative code, at the level of the C source code, is shown in Listing 4.8. The parameters values appearing in the NLR model, are defined in Arrays/Lists `A_param` and `B_param` in the iterative code. In order to access the elements of Arrays A and B

```
void executor_non_opt () {
  int A_param [8]={64,96,2112,2144,64,96,2112,2144};
  int B_param [8]={0,0,32,32,2048,2048,2080,2080};

  for ( int i0 =0 ; i0 <= 7 ; i0++ )
    for ( int i1 =0 ; i1 <= 31 ; i1++ )
      for ( int i2 = 0 ; i2 <= 31 ; i2++ ) {
        c_glob[i0]=0;//S1

        for ( int i3 =0 ; i3 <= 31 ; i3++ )
            c_glob[i0]+=A_base[A_param[i0]+64*i2+i3];//S2

        B_base[B_param[i0]+64*i1+i2]=
            c_glob[i0]*B_base[B_param[i0]+64*i1+i2];//S3
      }
}
```

Listing 4.8 – Recursion-Equivalent Iterative Code

in the iterative version, we need to use their original base addresses in which they are initially allocated, i.e., `A_base` and `B_base` respectively, and the parametrically-affine expressions in terms of the surrounding loops indices as array indices.

Since polyhedral optimizations cannot be applied to such loops, one would think of parallelizing the outermost loop, but the existing data dependences must be analyzed and removed first. Since we already know all parameters values and all loops information thanks to the NLR model, we can perform a precise data dependence analysis and find dependence-free parallel regions in this loop nest.

Initially, disregarding read-after-read dependences, Statement $S2$ is dependent on itself and $S1$, but these dependences are not carried by the outer loop that we would like to parallelize. However, Statement $S3$ is dependent on $S1$, $S2$ and itself. There is a risk to corrupt the memory location to which $S3$ writes to in case the outermost loop with index `i0` is parallelized. This is because, for instance, in the loop iterations where `i0=0` and `i0=1`, the parameter value obtained from `B_param` is `0`, and so the same memory location is accessed by $S3$, i.e., `B_base[64*i1+i2]`; accordingly, these iterations cannot be run in parallel. Making use of the NLR model information, we can create lists of iterations that can be run safely in parallel. In this example, we need to be careful about the accesses and writes to Array `B_base` by $S3$, so for every iteration of the outermost loop, we compute the range of addresses that are supposed to be accessed by $S1$. In case two iterations happen to have mutual exclusive ranges, then they can run in parallel safely such that their iterations belong to distinct parallel lists. Here, for the first iteration where `i0=0`, the offset of Array `B_base` accessed by $S3$ ranges from 0 for `i1=0` and `i2=0` to $2,015$ for `i1=31` and `i2=31` (31 is the NLR loops upper bound); same for the second iteration where `i0=1`. As for the third and fourth iterations, i.e., `i2=2` and `i2=3` respectively, the range of array offsets is between 32 and 2047 which overlaps with those of the previous iterations. This means that the iterations between `i0=0` and `i0=3` must be executed serially. Similarly, the iterations between `i0=4` and `i0=7` cannot be executed in parallel w.r.t. each other. Yet, we can still execute the iterations $0-3$ in parallel with the iterations $4-7$, i.e., we have two parallel lists of iterations in this example.

```
void executor_opt_parallel () {
  //parallel lists
  int B_p_lists[2][4]={{0,0,32,32},{2048,2048,2080,2080}};
  int A_p_lists[2][4]={{64,96,2112,2144},{64,96,2112,2144}};
  int idx_lists[2][4]={{0,1,2,3},{4,5,6,7}};

  int i_p, i0, i1, i2, i3;
  #pragma omp parallel
  #pragma omp for private(i0,i1,i2,i3)
  for ( i_p = 0 ; i_p < 2 ; i_p++ )
    for ( i0 = 0 ; i0 < 4 ; i0++ )
      for ( i1 = 0 ; i1 <= 31 ; i1++ )
        for ( i2 = 0 ; i2 <= 31 ; i2++ ) {
          c_glob[idx_p_lists[i_p][i0]]=0;

          for ( int i3 =0 ; i3 <= 31 ; i3++ )
            c_glob[idx_p_lists[i_p][i0]]+= A_base[A_p_lists[i_p][i0]+64*i2+i3];

          B_base[B_lists[i_p][i0]+64*i1+i2]=
              c_glob[idx_p_lists[i_p][i0]]*B_base[B_p_lists[i_p][i0]+64*i1+i2];
        }
}
```

Listing 4.9 – Parallelized Iterative Code

Finally, the outermost loop is broken into two loops, such that the outer one iterates over the parallel lists which are only two in this example and it can be parallelized. The inner one iterates over the elements of each list. The corresponding parallelized C code is shown in Listing 4.9.

At the end of this chapter, we have thoroughly presented the Rec2Poly framework, its phases, and the optimizations which it is capable of applying. In the next chapter, we present some interesting experiments performed and the interesting results obtained using Rec2Poly.

Chapter 5

Benchmarks

"There are three principal means of acquiring knowledge... observation of nature, reflection, and experimentation. Observation collects facts; reflection combines them; experimentation verifies the result of that combination."
— Denis Diderot

After having introduced the Rec2Poly framework and described its phases in depth, we dedicate this chapter to show some benchmark experiments performed to assess the performance of Rec2Poly. Also, we discuss, based on these experiments, some aspects of Rec2Poly that must be optimized further in addition to interesting functionalities that can be embedded in Rec2Poly to broaden its scope.

The chapter is organized as follows. In the first section, Section 5.1, we present recursive programs with polyhedral dynamic behavior. Also, we display the analysis results and the polyhedral models generated by Rec2Poly for these programs. In the second section, Section 5.2, we describe the inspectors that Rec2Poly generates when different inspector optimizations are activated and evaluate the inspector-executor codes and the optimizations that Rec2Poly applies for the presented programs. Then, in the the third section, Section 5.3, we point out challenges that may be faced while trying to analyze and optimize some recursive programs, and we discuss how Rec2Poly can be extended in the future to handle them.

5.1 Recursive Programs with Polyhedral Behaviors

In this section, we show Rec2Poly's polyhedral modeling of interesting recursive implementations of Matrix Multiplication and Heat.

5.1.1 Matrix Multiplication

We present two different recursive algorithms for the matrix multiplication problem. We show and discuss the run-time analysis results obtained by Rec2Poly for each one separately as follows.

Recursive Implementation: One Dimension Down per Call

We have already shown, in chapter 3, a recursive implementation of matrix-vector multiplication, which can be considered as a simplified version or a special case of matrix multiplication, in which the matrix/vector dimension diminishes by one with every recursive call. Here, we tackle a similarly designed recursive algorithm and implementation for the general recursive matrix multiplication problem.

The experimented matrix multiplication program includes the recursive Function "MatrixMultiplication" that computes the product of the matrices passed to it as parameters. The C code of this function is displayed in Listing 5.1. The function involves three recursive calls and one terminating condition. It accesses the memory multiple times in a unique basic block to perform the computation `C[i][j]+=A[i][k]*B[k][j]` when the second if-condition in the function is valid. It does not include any loops; there is no evidence for a loop behavior.

```
void MatrixMultiplication(double **A , double **B , double **C , int i ,
    int j , int k){

    if (i>=ROW1) //all rows of A are handled
        return;

    if(k<COLUMN1 && j<COLUMN2) //not all columns of A & rows of B handled
    {
        C[i][j] += A[i][k]*B[k][j];
        MatrixMultiplication( A , B , C , i , j , k+1 );
    }

    else if (j<COLUMN2) //not all columns of B are handled
        MatrixMultiplication( A , B , C , i , j+1 , 0 );

    else //not all rows of A are handled
        MatrixMultiplication( A , B , C , i+1 , 0 , 0 );
}
```

Listing 5.1 – Matrix Multiplication Recursive Function C Code

We use Rec2Poly to analyze this program at run-time. Rec2Poly generates for it an instrumented version to produce the run-time control and memory trace of the recursive function. After the instrumented matrix multiplication program runs for matrices of size 1000×1000 and generates the trace, the trace is then fed to NLR. The profiling output shows a perfect affine loop nest composed of three loops with iterators i0, i1 and i2. The model is displayed in Figure 5.1.

```
for i0 = 0 to 999
  for i1 = 0 to 999
    for i2 = 0 to 999
      val MatrixMultiplication::if.then3
        , 1*i0 , 1*i2 , 1*i2 , 1*i1 , 1*i0 , 1*i1 , 1*i1
```

Figure 5.1 – Matrix Multiplication Control and Memory Behavior NLR Model

As we can see, the loops encompass the only one existing impacting basic block including seven affine expressions. Every affine expression corresponds to an LLVM IR memory instruction in the basic block with ID "`if.then3`", i.e., a read/load or a write/store instruction required for performing the computation `C[i][j]+=A[i][k]*B[k][j]`. In Listing 5.2, we show the content of the corresponding basic block at the level of the LLVM IR. This basic block includes seven stores and loads which are highlighted in the listing.

So, the seven affine expressions, in the NLR model, define, in terms of the surrounding NLR loops indices, the relative memory addresses accessed by these LLVM IR memory instructions, which in this example happen to be equivalent to the accessed arrays indices: `i` and `k` for the load/read from `A[i][k]`, `k` and `j` for the read from `B[k][j]`, `i` and `j` for the read from `C[i][j]` and `j` for the store/write to `C[i][j]`.

```
if.then3:                                         ; preds = %land.lhs.true
  %idxprom = sext i32 %i to i64
  %arrayidx = getelementptr inbounds double*, double** %A, i64 %idxprom
  %0 = load double*, double** %arrayidx, align 8
  %idxprom4 = sext i32 %k to i64
  %arrayidx5 = getelementptr inbounds double, double* %0, i64 %idxprom4
  %1 = load double, double* %arrayidx5, align 8
  %idxprom6 = sext i32 %k to i64
  %arrayidx7 = getelementptr inbounds double*, double** %B, i64 %idxprom6
  %2 = load double*, double** %arrayidx7, align 8
  %idxprom8 = sext i32 %j to i64
  %arrayidx9 = getelementptr inbounds double, double* %2, i64 %idxprom8
  %3 = load double, double* %arrayidx9, align 8
  %mul = fmul double %1, %3
  %idxprom10 = sext i32 %i to i64
  %arrayidx11 = getelementptr inbounds double*, double** %C, i64 %idxprom10
  %4 = load double*, double** %arrayidx11, align 8
  %idxprom12 = sext i32 %j to i64
  %arrayidx13 = getelementptr inbounds double, double* %4, i64 %idxprom12
  %5 = load double, double* %arrayidx13, align 8
  %add = fadd double %5, %mul
  store double %add, double* %arrayidx13, align 8
  %add14 = add nsw i32 %k, 1
  call void @MatrixMultiplication(double** %A, double** %B, double** %C,
    i32 %i, i32 $j, i32 %add14)
  br label %if.end24
```

Listing 5.2 – LLVM Basic Block "`if.then3`" Content

Therefore, this recursion behaves as an affine loop nest and the code can be remodeled and rewritten as affine loops compliant with the polyhedral model. The control flow graph of the iterative code equivalent to the MatrixMultiplication function that can be generated based on this profiling result is illustrated in Figure 5.2.

General Matrix Multiplication: High Performance Recursive Linear Algebra Library

We also conducted other experiments on another matrix multiplication implementation, a recursive C implementation of the general matrix-matrix multiplication (GEMM) from

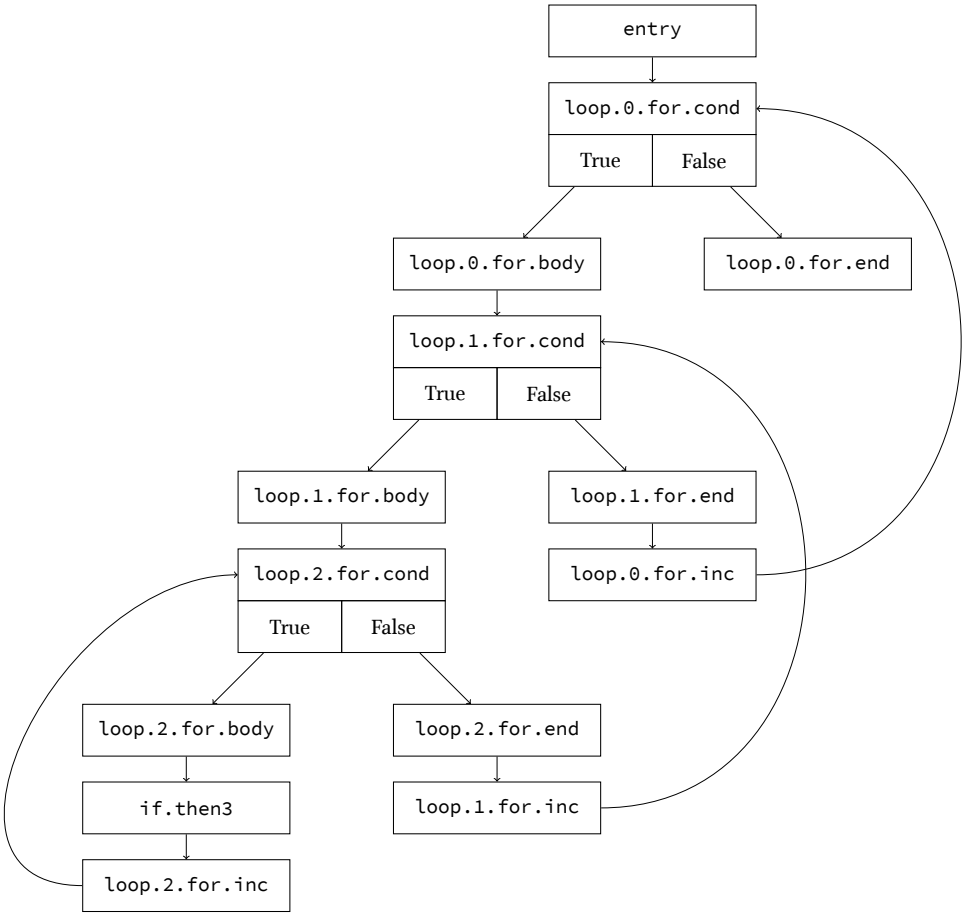


Figure 5.2 – Control Flow Graph Example

the High Performance Recursive Linear Algebra Library [124]. This version handles sub-matrices by successive dichotomy and bipartitioning (division of the matrices dimensions by two) until a specific threshold.

This program involves mainly seven functions: Functions main, gemm, sgemm_trans_r, sgemm_i_trans, sgemm_j_trans, sgemm_k_trans and sgemm_trans2.
There is an indirect recursion among the four functions: sgemm_trans_r, sgemm_i_trans, sgemm_j_trans and sgemm_k_trans.

A simplified call graph of GEMM is depicted in Figure 5.3. It clearly shows the indirect recursions and the interactions among the functions. Each arrow in this graph from one node to another represents a call to the function of the latter by that of the prior.

Function main calls Function “gemm” that creates the matrices as arrays, fills them and then initiates a recursion by calling the recursive Function “sgemm_trans_r” passing to it as parameters the arrays pointers, etc. Function “sgemm_trans_r” is the initial recursive function; it may call “sgemm_i_trans”, “sgemm_j_trans” and “sgemm_k_trans”, each of which calls it back through two calls. The recursion among them proceeds to decompose the matrices until the threshold is reached. Then, Function “sgemm_trans_r” calls a non-recursive function which is “sgemm_trans2” to perform the multiplication iteratively between the final sub-matrices.

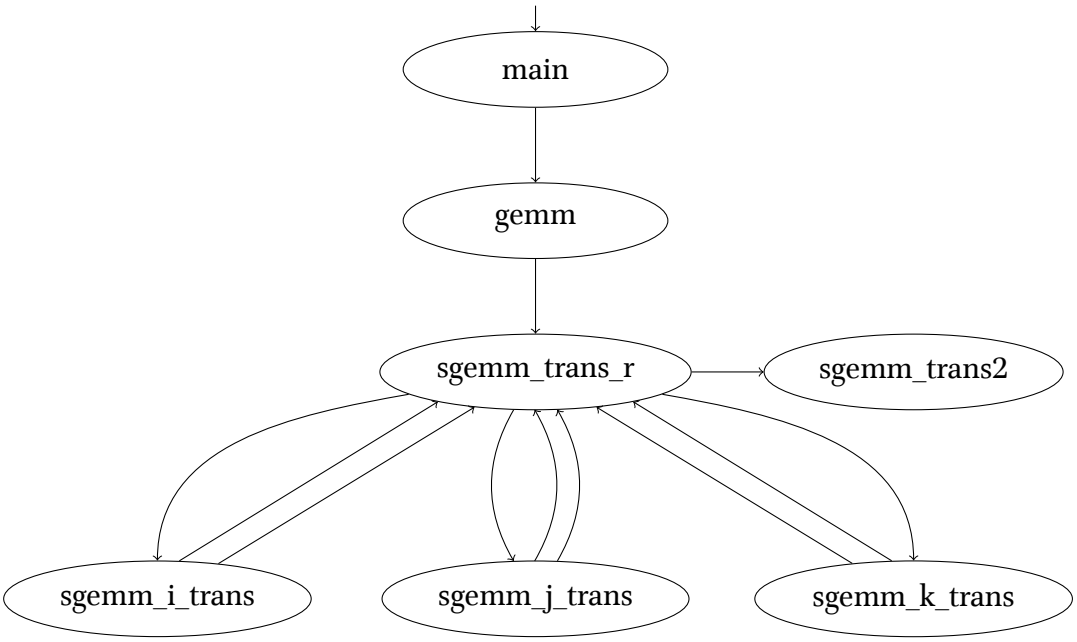


Figure 5.3 – GEMM Program Call Graph

```
void sgemm_trans2(float *a,float *b,float *c,long i,long j,long k,long
    s_a,long s_b,long s_c,const float alpha,const float beta)
{
    float s1,s2,s3,s4,s,x;
    long i2,j2,k2;
    for ( i2 = 0 ; i2 < i ; i2 ++ ) {
    for ( j2 = 0 ; j2 < (j - 3) ; j2 += 4 ) {
        s1 = s2 = s3 = s4 = 0.0;
        for ( k2 = 0 ; k2 < k ; k2 ++ ) {
            x = a[i2 * s_a + k2];
            s1 += x * b[k2 + j2 * s_b];
            s2 += x * b[k2 + ( j2 + 1 ) * s_b];
            s3 += x * b[k2 + ( j2 + 2 ) * s_b];
            s4 += x * b[k2 + ( j2 + 3 ) * s_b];
        }
      c[i2 * s_c + j2] = alpha * s1 + beta * c[i2 * s_c + j2];
      c[i2 * s_c + j2 + 1] = alpha * s2 + beta * c[i2 * s_c + j2 + 1];
      c[i2 * s_c + j2 + 2] = alpha * s3 + beta * c[i2 * s_c + j2 + 2];
      c[i2 * s_c + j2 + 3] = alpha * s4 + beta * c[i2 * s_c + j2 + 3];
    }

    for ( ; j2 < j ; j2 ++ ) {
        s = 0.0;
        for ( k2 = 0 ; k2 < k ; k2 ++ ) {
            s += a[i2 * s_a +k2] * b[k2 + j2 * s_b];
        }
        c[i2 * s_c + j2] = alpha * s + beta * c[i2 * s_c + j2];
    }
  }
}
```

Listing 5.3 – GEMM sgemm_trans2 Function C Code

```
for i0 = 0 to 63
  val sgemm_trans2::entry , 1*i0
  for i1 = 0 to 49
    val sgemm_trans2::for.cond
    for i2 = 0 to 11
      val sgemm_trans2::for.cond1
      val sgemm_trans2::for.body3 , 1*i0 , 1*i0 , 1*i0 , 1*i0
      for i3 = 0 to 49
        val sgemm_trans2::for.cond4
        val sgemm_trans2::for.body6
          , [64:0,50,0,...][i0] + 200*i1 + 1*i3
          , 1*i0 , 1*i0
          , [64:0,50,10000,...][i0] + 800*i2 + 1*i3
          , 1*i0 , 1*i0 , 1*i0
          , [64:200,250,10200,...][i0] + 800*i2 + 1*i3
          , 1*i0 , 1*i0 , 1*i0
          , [64:400,450,10400,...][i0] + 800*i2 + 1*i3
          , 1*i0 , 1*i0 , 1*i0
          , [64:600,650,10600,...][i0] + 800*i2 + 1*i3
          , 1*i0 , 1*i0
        val sgemm_trans2::for.inc , 1*i3
      val sgemm_trans2::for.cond4
      val sgemm_trans2::for.end , 1*i0 , [64:0,...][i0] + 200*i1 + 4*i2 , ...
      val sgemm_trans2::for.inc72 , 4*i2
    val sgemm_trans2::for.cond1
    val sgemm_trans2::for.cond75
    val sgemm_trans2::for.body77 , 1*i0
    for i2 = 0 to 49
      val sgemm_trans2::for.cond78
      val sgemm_trans2::for.body80 , [64:0,...][i0] + 200*i1 + 1*i2 , ...
      val sgemm_trans2::for.inc89 , 1*i2
    val sgemm_trans2::for.cond78
    val sgemm_trans2::for.end91 , 1*i0 , [64:48,...][i0] + 200*i1 , ...
    val sgemm_trans2::for.inc101 , 48
    val sgemm_trans2::for.cond75
    val sgemm_trans2::for.body77 , 1*i0
    for i2 = 0 to 49
      val sgemm_trans2::for.cond78
      val sgemm_trans2::for.body80 , [64:0,...][i0] + 200*i1 + 1*i2 , ...
      val sgemm_trans2::for.inc89 , 1*i2
    val sgemm_trans2::for.cond78
    val sgemm_trans2::for.end91 , 1*i0 , [64:49,...][i0] + 200*i1 , ...
    val sgemm_trans2::for.inc101 , 49
    val sgemm_trans2::for.cond75
    val sgemm_trans2::for.inc104 , 1*i1
  val sgemm_trans2::for.cond
```

Figure 5.4 – NLR Model for the Control and Memory Behavior of GEMM

The recursive functions do not involve significant computations; they are only responsible for the matrices decomposition. Yet, "sgemm_trans2" envelops expensive memory accesses. In addition, the memory instructions in "sgemm_trans2" are executed within interesting loop nests. The C code of this function is shown in Listing 5.3.

So, not only may the existing recursion hide an optimizable loop behavior, but also it already disturbs existing loop structures which makes this program interesting for our study. Eliminating the recursion and rewriting the code and the loops as sequences of affine loop nests will open the opportunity for interesting optimizations.

When Rec2Poly analyzes and prepares this program for the offline profiling, it will take into account and clone the recursive functions and their only reachable function "sgemm_trans2". Since "sgemm_trans2" and its clone include non-induction local variables (e.g., `s1`, `s2`, etc.), Rec2Poly globalizes them besides injecting an invocation counter dedicated to this function's clone. Then, Rec2Poly generates the instrumented version of the code by adding the printing instructions exclusively to the reachable function since it is the only one performing impacting memory accesses. When the instrumented code for matrices of sizes 200×200 and threshold 80 is executed, and the corresponding run-time control and memory trace is generated, the trace is then fed to NLR to discover potential loops in it. As a result, NLR produces the model displayed in Figure 5.4.

As we can see, the NLR model is a parametrically-affine loop nest made of six loops. So, the control flow of the recursive part of the GEMM program can be fitted within a loop nest. Besides, the relative memory addresses touched at run-time can be expressed as semi-affine expressions such that some coefficients are not constant throughout all loops iterations, yet their values can be predicted and known given the NLR model information. Hence, the recursive code part of this program can be rewritten accordingly as an iterative code, and, although it may not fit in the polyhedral model, the data dependences can be unveiled and a dedicated transformation and significant optimization can still be applied.

Note that, in general, the NLR loops bounded by the basic blocks with IDs "for.cond" and "for.end" correspond to loops that already exist in the code. In this model, there is only the outermost NLR loop that does not exist in the source code; this loop is resulting from the recursion itself. The rest of the NLR loops correspond to the loops existing in Function "sgemm_trans2".

5.1.2 Heat

Another experiment was conducted on a recursive C implementation of Heat from the REAPAR benchmark suite [114]. Heat is a program that performs stencil computations. The Heat program involves mainly six functions: the main function, heat, divide, allcgrid, initgrid and compstripe. Its call graph is depicted in Figure 5.5. The main function initiates the computation by calling the heat function. The "heat" function calls the function divide three times. Function "divide" is a recursive function that involves two non-tail direct recursive calls. Then, divide may call "allcgrid", "initgrid" or "compstripe". Note that there are two invocations of "compstripe" in Function "divide".

First, Function "heat", when activated, allocates memory for two arrays of pointers. When heat executes its first call to "divide", and after "divide" has completed with the division and decomposition, "divide" calls "allcgrid" to allocate memory for the array point-

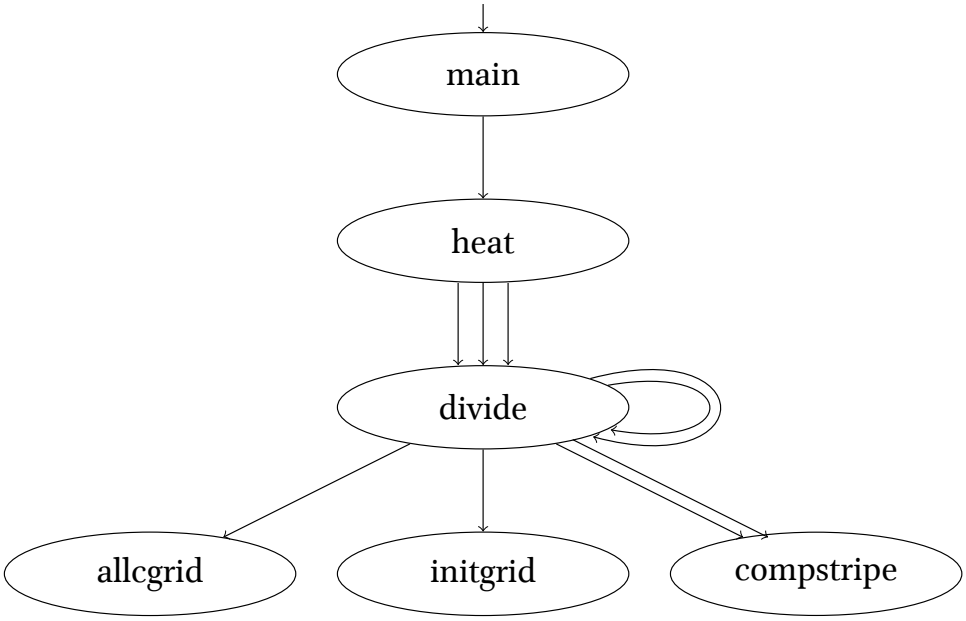


Figure 5.5 – Heat Program Call Graph

ers. When “heat” calls “divide” for the second time, “divide” will eventually call “initgrid” responsible for initializing the arrays values. Then, through the third “divide” call inside “heat”, “compstripe” is reached to perform the arrays computations in a recursive fashion too. Yet, the third call to Function “divide” inside “heat” is placed within a for-loop which invokes “divide” several more times. In Figure 5.6, the control flow graph of Function “heat” is illustrated. The first basic block is the entry block in which the two calls to “divide” are executed. The third call exists in the basic block called “for.body”, involved in a cycle with the Basic Blocks “for.cond” and “for.inc” reachable from “entry”, corresponding to the for-loop existing in the code.

Accordingly, Function “divide” must be analyzed by Rec2Poly since it is recursive. Functions “allcgrid”, “initgrid” and “compstripe” are reachable from “divide”, so they must also be analyzed. Since “heat” initiates the recursion from a loop body at least once, it is also analyzed for being the source of the recursion.

Also, we shall mention that Function “divide” performs mainly the decomposition of the program without accessing memory or performing any significant computation. So, it neither gets to have an invocation counter nor appears in the loop model generated by the offline profiling. The arguments it passes to its reachable functions will be saved by the parameter saver, part of the inspector generated by Rec2Poly, and used to build the executor in case needed.

On the other hand, the reachable functions, especially “compstripe”, involve several expensive interesting loops. The C code of the “compstripe” function is shown in Listing 5.4.

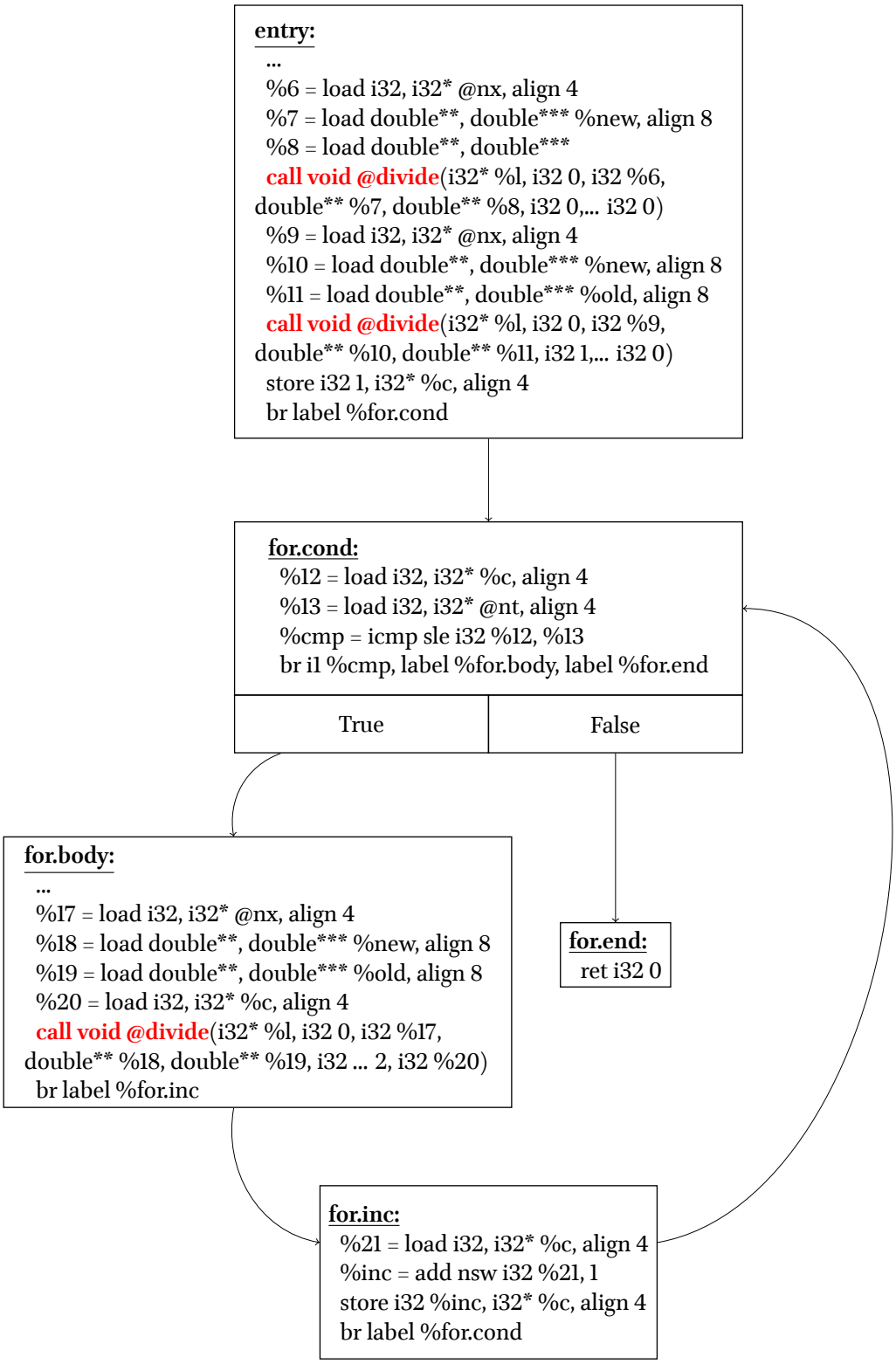


Figure 5.6 – Heat Function Control Flow Graph

```
void compstripe( double **new, double **old, int lb, int ub)
{
  int a, b, llb, lub;
  llb = (lb == 0) ? 1 : lb;
  lub = (ub == nx) ? nx - 1 : ub;

  for (a=llb; a < lub; a++) {
    for (b=1; b < ny-1; b++) {
      new[a][b] = dtdxsq * (old[a+1][b] - 2 * old[a][b] + old[a-1][b]) +
      dtdysq * (old[a][b+1] - 2 * old[a][b] + old[a][b-1]) + old[a][b];
    }
  }

  for (a=llb, b=ny-1; a < lub; a++)
    new[a][b] = randb(xu + a * dx, t);

  for (a=llb, b=0; a < lub; a++)
    new[a][b] = randa(xu + a * dx, t);

  if (lb == 0) {
    for (a=0, b=0; b < ny; b++)
      new[a][b] = randc(yu + b * dy, t);
  }

  if (ub == nx) {
    for (a=nx-1, b=0; b < ny; b++)
      new[a][b] = randd(yu + b * dy, t);
  }
}
```

Listing 5.4 – Heat's Compstripe Function C Code

This recursive program is highly interesting for our study because it includes many loops that are probably optimizable besides the target recursive function. In Heat, there is a loop initiating a recursion numerously such that the recursion tends to execute indirectly several affine loop nests each time. In other words, the existing recursion distorts the existence of these loops. The existence of the recursion among loops may deprive them of advanced optimization and parallelization opportunities.

Therefore, it is intriguing to investigate the possibility of rewriting the code as an equivalent sequence of affine loop nests without the recursion. Whenever possible, the time performance gain obtained from such code reconstruction and optimization may be considerable.

The Rec2Poly framework succeeds in modeling the control and memory behavior of the Heat program as a sequence of fully-affine loop-nests. Part of the NLR model generated, as the whole is too long to fit on one page, is shown in Figure 5.7.

The model corresponds to the trace produced at run-time by the instrumented version of the Heat program, generated by Rec2Poly, when processing two dimensional arrays of size 1024×512 such that the third call to Function "divide" in function heat is executed $200,000$ times.

```
...
for i0 = 0 to 99999
  val heat::for.cond
  val heat::for.body , 0 , 0 , 0
  val compstripe::entry , 128*i0
  for i1 = 0 to 14
    val compstripe::for.cond
    for i2 = 0 to 509
      val compstripe::for.cond7
      val compstripe::for.body10
        , 0 , 2 + 1*i1 , 1 + 1*i2 , 1 + 1*i1 , 1 + 1*i2 , 1*i1 , 1 + 1*i2
        , 0 , 1 + 1*i1 , 2 + 1*i2 , 1 + 1*i1 , 1 + 1*i2 , 1 + 1*i1 , 1*i2
        , 1 + 1*i1 , 1 + 1*i2 , 1 + 1*i1 , 1 + 1*i2
    val compstripe::for.cond7
    val compstripe::for.inc53 , 1 + 1*i1
  val compstripe::for.cond
  val compstripe::for.end55 , 0
  for i1 = 0 to 14
    val compstripe::for.cond57
    val compstripe::for.body59 , 0 , 0 , 0 , 1 + 1*i1 , 511
  val compstripe::for.cond57
  for i1 = 0 to 14
    val compstripe::for.cond72
    val compstripe::for.body75 , 1 + 1*i1 , 0
  val compstripe::for.cond72
  for i1 = 0 to 511
    val compstripe::for.cond85
    val compstripe::for.body88 , 0 , 1*i1
  val compstripe::for.cond85
  for i1 = 0 to 61
    val compstripe::entry , 1 + 128*i0 + 1*i1
    for i2 = 0 to 15
      val compstripe::for.cond
      for i3 = 0 to 509
        val compstripe::for.cond7
        val compstripe::for.body10
          , 0 , 17 + 16*i1 + 1*i2 , 1 + 1*i3 , ...
      val compstripe::for.cond7
      val compstripe::for.inc53 , 16 + 16*i1 + 1*i2
    val compstripe::for.cond
    val compstripe::for.end55 , 0
    for i2 = 0 to 15
      val compstripe::for.cond57
      val compstripe::for.body59 , 0 , 0 , 0 , 16 + 16*i1 + 1*i2 , 511
    val compstripe::for.cond57
    ...
  ...
```

Figure 5.7 – NLR Model for the Control and Memory Behavior of Heat

The shown part of the model is the one that is exclusively produced by the for-loop structure in Function “heat” in which the third call to the recursive Function “divide” is executed, thus reaching “compstripe”. This part is the most interesting one for code rewriting because it involves the largest, deepest and most expensive loop nest, and it is responsible for the major time overhead of the whole program. Note that the first skipped part of the model involves sequences of lighter loops involving Functions “allcgrid” and “initgrid”. Furthermore, the remaining part of the outer NLR loop body includes repetitions of the same displayed control behavior but with different memory offsets; this is caused by reaching the terminating condition, in which “compstripe” is called, many times by the recursion.

Given this NLR model, we conclude that the control behavior is linear/affine as the IDs of the functions’ basic blocks visited at run-time are encompassed by the loops. In addition, we notice that the memory relative offsets are represented as affine functions of the surrounding loops indices which also indicates that the memory behavior is affine. Hence, the whole behavior is affine and the code can be substituted by equivalent polyhedral loop structures.

5.2 Inspector-Executor

In this section, we discuss the inspector-executor codes generated by Rec2Poly for each of the presented programs. As mentioned in the previous chapter, there are many optimizations available for the inspector part. So, we also describe the generated inspector code when enabling different optimizations and the required number of POSIX threads. Furthermore, we show the execution speedup and compare the performance of the different inspectors and the inspector-executor codes corresponding to each of the recursive programs.

The programs and the generated codes were compiled with Clang version 6.0 and the optimization flags `-O3 -march=native`, and run on two Intel Xeon CPU E5-2650 v3 @ 2.30GHz of ten cores each. The optimizations and parallelizations for the executors with affine loops were simulated with Pluto and those for the executor with the parametrically-affine loops were simulated based on the approach introduced in the previous chapter and parallelized using OpenMP. The Parallel executor codes were executed using forty threads on the twenty hyperthreaded cores of the hardware platform.

Note that the default inspector codes are already subject to default optimizations, the execution of obvious basic blocks like the loop-condition basic blocks do not need to be verified by the control trace generator-verifier couple. Also, in case there are affine loop nests, they are optimized as explained in the previous chapter exclusively in the control trace generators and verifiers. Optimizing the control-related threads by default is crucial because they are initially the costliest of all in the inspectors as verifying the control must be carried out serially, and so the corresponding thread workload cannot be distributed over multiple threads. Another optimization is also activated by default which is assigning the verification of the induction variables values and the functions counters values, if they exist, to separate trace generator-verifier couple threads instead of handling them along with the memory offsets. This helps balance the load better among the threads.

5.2.1 Matrix Multiplication

Matrix Implementation: One Dimension Down per Call

Given the perfect affine NLR loop model, Rec2Poly can generate, for the first recursive implementation presented of the matrix multiplication problem, equivalent iterative codes that can be optimized, tiled and parallelized by Pluto for later executions. The control flow graph of the executor is already shown in Figure 5.2. In addition, Rec2Poly generates the inspector codes responsible for verifying the correctness of the executor since it is generated based on offline profiling.

Note that there are no local variables in the impacting function, and so no globalization needs to be performed and no invocation counter variable is injected. Also, there are no loops and so no induction variables must be verified. Furthermore, a parameter saver is not needed.

```
...
if.then3:                                         ; preds = %land.lhs.true
  %idxprom = sext i32 %i to i64
  %arrayidx = getelementptr inbounds double*, double** %a, i64 %idxprom
  %0 = load double*, double** %arrayidx, align 8
  %idxprom4 = sext i32 %k to i64
  %arrayidx5 = getelementptr inbounds double, double* %0, i64 %idxprom4
  %1 = load double, double* %arrayidx5, align 8
  %idxprom6 = sext i32 %k to i64
  %arrayidx7 = getelementptr inbounds double*, double** %b, i64 %idxprom6
  %2 = load double*, double** %arrayidx7, align 8
  %idxprom8 = sext i32 %j to i64
  %arrayidx9 = getelementptr inbounds double, double* %2, i64 %idxprom8
  %3 = load double, double* %arrayidx9, align 8
  %mul = fmul double %1, %3
  %idxprom10 = sext i32 %i to i64
  %arrayidx11 = getelementptr inbounds double*, double** %c, i64 %
    idxprom10
  %4 = load double*, double** %arrayidx11, align 8
  %idxprom12 = sext i32 %j to i64
  %arrayidx13 = getelementptr inbounds double, double* %4, i64 %idxprom12
  %5 = load double, double* %arrayidx13, align 8
  %add = fadd double %5, %mul
  store double %add, double* %arrayidx13, align 8
  %add14 = add nsw i32 %k, 1
  call void @MatrixMultiplication(double** %a, double** %b, double** %c,
    i32 %i, i32 %j, i32 %add14)
  br label %if.end24
...
```

Listing 5.5 – Matrix Multiplication Function if.then3 Basic Block LLVM IR

The default inspector generated by Rec2Poly for this program consists of:

- one control trace generator,

- seven memory trace generators since the only existing computation in the only impacting function in this program, `C[i][j]+=A[i][k]*B[k][j]`, requires seven memory access instructions, six loads and one store (check out Listing 5.5), whose relative addresses must be re-generated and verified at run-time for every optimized execution.
- one control verifier, and
- seven memory verifiers.

This inspector requires sixteen threads, eight for the trace generators and eight for verifiers. Also, Rec2Poly dedicates another thread for the backup recursive code in case a misspeculation is detected during the execution. Accordingly, during the inspection, seventeen threads must execute and if the verification holds, then the backup thread is cancelled and the optimized executor runs using the forty threads available.

Yet, the inspector can be optimized further. As we can see in Basic Block "if.then3" in Listing 5.5, there are redundant memory accesses that can be removed, e.g., the touched memory location $C[i][j]$ into which the result of the multiplication must be written, is also read to perform the increment. Accordingly, Rec2Poly can generate an optimized version of the inspector involving five memory trace generators and five verifiers instead of seven memory trace generators and seven verifiers. In total, this inspector requires twelve threads, six for the trace generators and six for the verifiers. Besides, there is the thread dedicated for the backup original code.

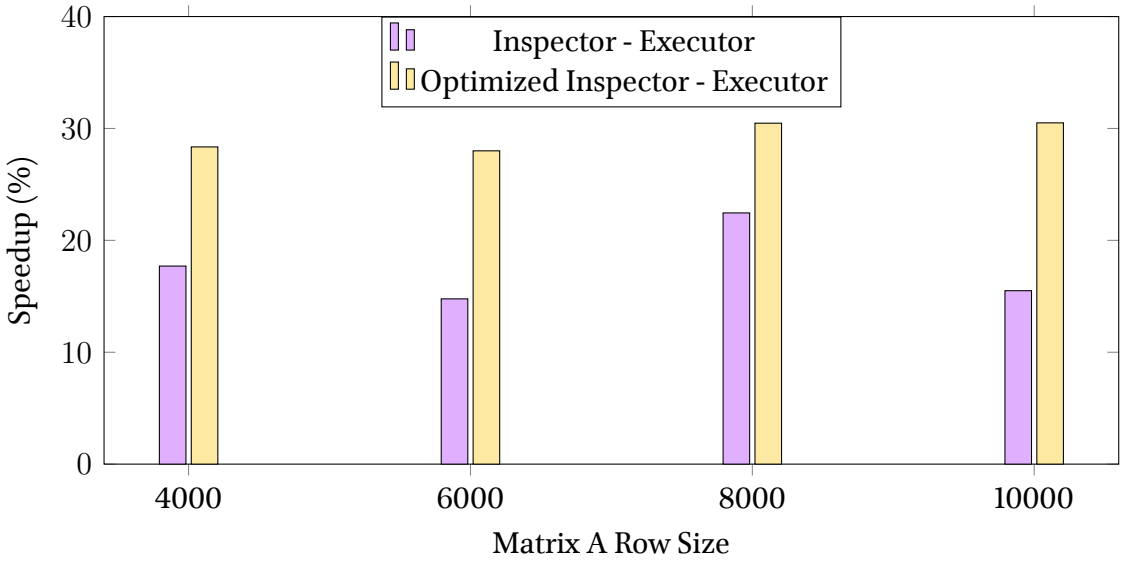


Figure 5.8 – Matrix Multiplication Experimental Results - Rec2Poly Speedup

In Figure 5.8, we show the experimental results, the speedups obtained when executing the inspector-executor code and the optimized inspector with the executor code with respect to the original recursive matrix multiplication code. The speedup percentage is computed as follows: $\frac{(T_o - T_{i/e})}{T_o} \times 100$ where T_o is the execution time of the original code and $T_{i/e}$ is the execution time of the code generated by Rec2Poly.
As we notice, the default inspector-executor performed better than the original code for

about 15.5%, and the optimized inspector-executor version performed even faster; the performance gain was about 30.5% for matrices A of size 10000×900 and B of size 900×1000.

Recursive General Matrix Multiplication (GEMM)

The offline profiling of this recursive program yields a parametrically affine loop nest which can be used to generate, for later program executions, iterative codes for which a precise data dependence analysis can be performed and the outer loop nest can be broken into independent iterations to be parallelized accordingly. Rec2Poly as usual generates an inspector to verify the executor code at run-time. The default inspector generated is composed of a parameter saver, seven trace generators and seven verifiers. It has:

- one trace generator-verifier couple that handles the control flow of this program;
- one trace generator-verifier couple that handles the verification of the existing loops induction variables values and the function invocation counter values, that helps verifying the globalized variables accessed;
- five memory trace generator-verifier couples; it is preferable for every memory-related couple to handle, for load balancing purposes, only one memory access per basic block. Since the maximal number of memory instructions in a basic block in this program is five (in "for.body6" of function sgemm_trans2), we need five of them.

Then, this inspector (Inspector I) requires fifteen threads, one for the parameter saver, seven for the trace generators and seven for the verifiers. As always, we have the backup original code running on the sixteenth thread. When the inspector terminates correctly, the optimized executor is launched and parallelized using OpenMP on forty threads.

There is a possibility to optimize this inspector and remove the redundant memory accesses as arrays b and c for instance are referenced using functions of the same indices multiple times; it is enough to verify one of the four accesses to array b in "for.body6" of Function "sgemm_trans2" per basic block visit to make sure that the rest are correct (same for c accesses). As a result, Rec2Poly can generate Inspector II, an optimized version of the inspector consisting of only four trace generator-verifier couples of which only two are dedicated to the inspection of the memory flow of the recursive code part against the NLR loop model. This inspector only requires nine threads, one for the parameter saver, four for the trace generators and four for the verifiers; the backup recursive code runs on a separate thread.

Furthermore, in Listing 5.3, we show an impacting function, in this program, including an interesting loop nest. Light copies of this function and its constituting loops will be generated in the control trace generators and their corresponding NLR loops will be generated in the verifiers by Rec2Poly as described in Chapter 4 in Section 4.4. Rec2Poly can modify the outermost and the innermost loops of the corresponding loop nests in the trace generators and the verifiers to terminate after three iterations as described in Subsection 4.4.5. Accordingly, Rec2Poly generates Inspector III as an optimized version of Inspector II.

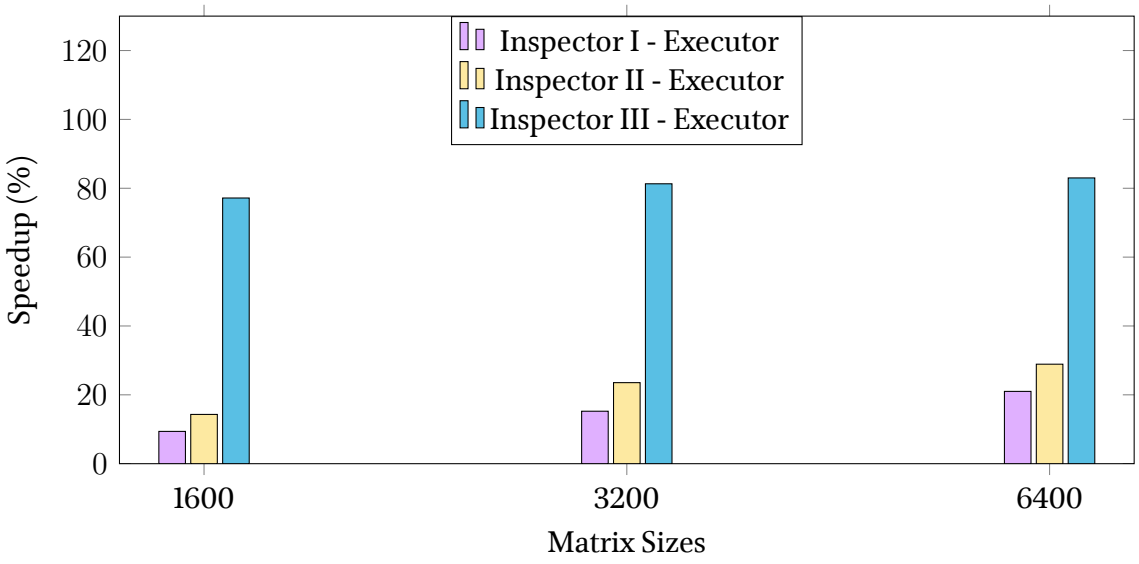


Figure 5.9 – Program GEMM Experimental Results - Rec2Poly Speedup

In Figure 5.9 we show the experimental results obtained for multiple executions of the recursive GEMM program and the corresponding inspector-executor codes for different matrices sizes. All of our optimized codes outperformed the original recursive code. The inspector-executor code, with the default inspector (I), performed better than the original code by about 20%. Yet, the inspector-executor code with the optimized inspector (II) and the redundant memory accesses removal performed even better; it was faster than the original code by about 28.9%. Finally, the inspector-executor code with the most aggressively optimized inspector (III) performed faster by 83%.

5.2.2 Heat

Similarly, the Heat program can be rewritten as an inspector-executor code based on a previously generated NLR model corresponding to the memory and control execution behavior of the Heat recursive code part for the same input size.

The default inspector of this program that can be generated by Rec2Poly is expected to be hefty. As we can see from Listing 5.4, the first loop nest in the “compstripe” function body envelops a costly computation that requires numerous memory accesses. In the NLR model in Figure 5.7, we can count up to sixteen affine expressions besides the zeros corresponding to the global scalars accessed in the Basic Block “for.body10” in “compstripe”. So, we have at least sixteen memory accesses repeating again and again within an expensive loop nest that must be tracked and verified, in new execution contexts of Heat against the supposedly equivalent NLR model, by the inspector. This basic block contains the largest number of memory accesses in the whole program.

In general, Rec2Poly creates a trace generator and a verifier for every memory access per basic block in the inspector which means that Rec2Poly needs to generate sixteen memory trace generators and sixteen verifiers for Heat to handle the heaviest block. In addition, Rec2Poly creates a control trace generator and a verifier and dedicate a trace generator and verifier to take care of the function invocation counters and loop induction variables values. In total, Rec2Poly generates for the Heat inspector, eighteen trace

generator-verifier couples besides the parameter saver which requires thirty-seven threads plus the backup code thread.

Hence, it was highly interesting to investigate how powerful the inspector optimizations can be in such case. For this reason, we conducted experiments on the inspector part only activating multiple optimization levels and we show the experimental results in Figure 5.10. Note that, in the experiments, we executed the codes for different upper bounds of the loop in Function "heat" that initiates the third recursive call to Function "divide".

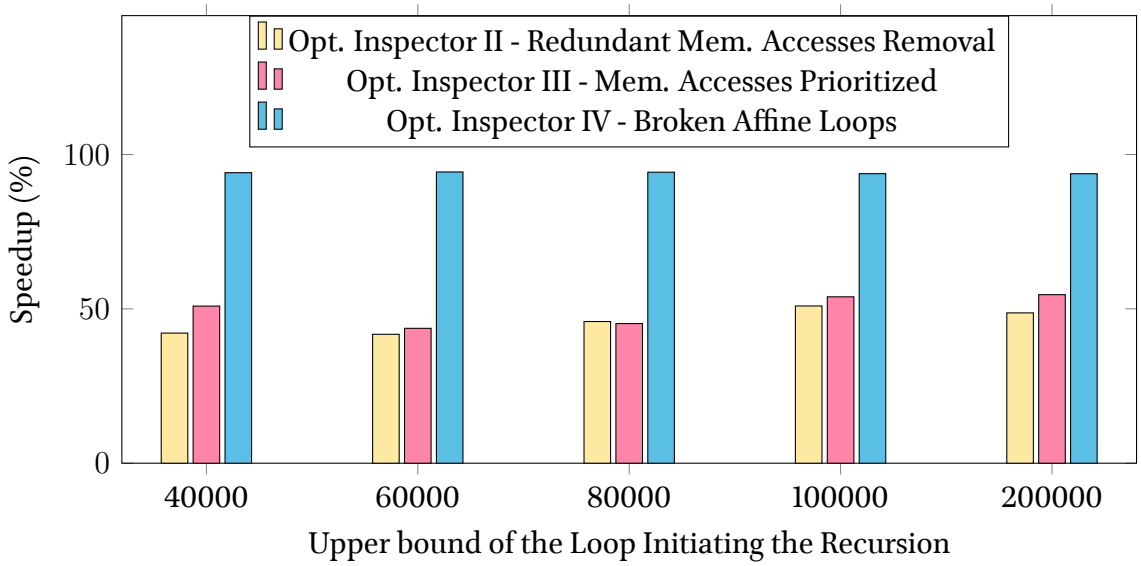


Figure 5.10 – Heat - Optimized Inspectors Speedup w.r.t. Inspector I

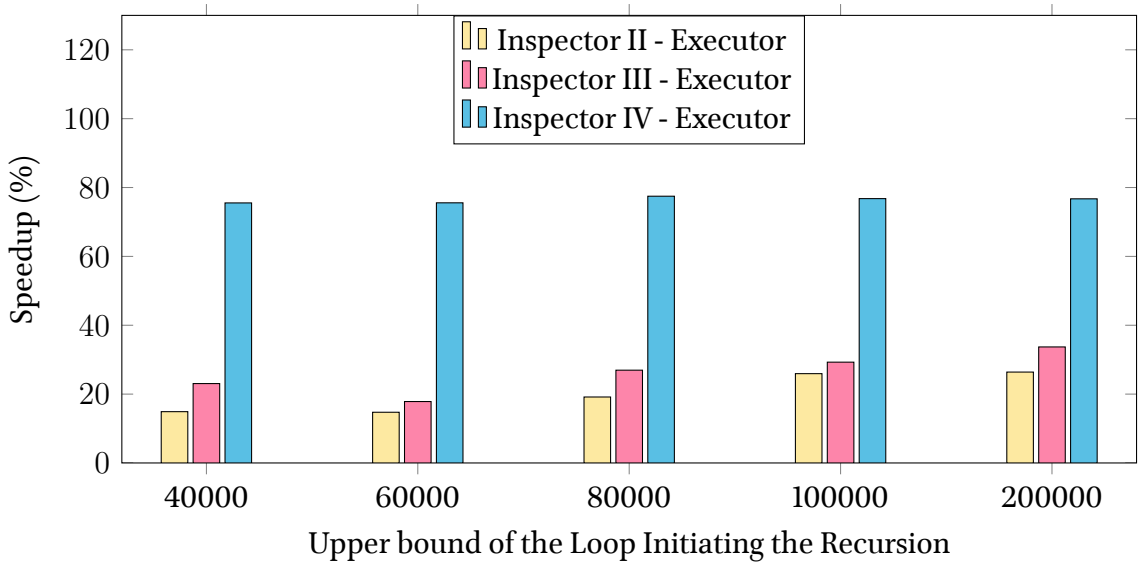


Figure 5.11 – Heat Inspector-Executor Experimental Results - Rec2Poly Speedup

The default inspector is called Inspector I. The first optimized inspector generated by Rec2Poly is Inspector II in which redundant memory accesses were removed. In this

inspector version, there is no need to verify all the sixteen accesses in "for.body10"; it is sufficient to verify only three of them since there were many redundant accesses to array `old`. This inspector requires only five trace generator-inspector couples instead of eighteen besides the parameter saver; it launches eleven threads besides the backup thread. Inspector II outperformed Inspector I by about 48.7% for heat loop upperbound equal to $200,000$ which was a significant percentage in comparison to what we got activating the same optimization for the previous examples.

We also embedded another manual optimization feature for the inspector in Rec2Poly. This optimization option allows the users to select costly basic blocks that Rec2Poly must consider a priority while generating the memory trace generators and verifiers. Accordingly, Rec2Poly creates a trace generator-verifier couple dedicated to only one memory access in the whole recursive code part, one of those existing in one of the costly basic blocks. The rest of the memory accesses in the whole model are tackled by a single trace generator-verifier couple.

In the case of Heat, we selected "for.body10" and "for.inc53" since they exist within the most expensive loop nest in the whole model. Consequently, Rec2Poly created four memory trace generators and verifiers for each of the accesses inside these basic blocks and one generator and verifier for the rest of the accesses, invocation counters and induction variables. There was no need for this optimization to be enabled for the previous experiments as there were no as obvious costly basic blocks. This optimized inspector, Inspector III, required five memory trace generators and five verifiers, one control trace generator and one verifier, and a parameter saver which meant thirteen threads besides the backup thread. This inspector ran faster than inspectors II and I by about 11.5% and 54.6% respectively.

Then, we tested the optimized Inspector IV generated by Rec2Poly by limiting the existing non-flow-impacting affine loops to three iterations at most. This inspector aggressively outperformed Inspector I by 93.7%.

Finally, we present the inspector-executor experimental results, carried out using the optimized versions of the inspector. Note that, as for the executor part, we only optimized the most expensive code part whose corresponding NLR loop nest is shown in Figure 5.7.

The performance speedup of the inspector-executor codes with respect to the original recursive code is shown in Figure 5.11. Inspector II - executor ran faster than the original code by about 26.4%, Inspector III performed faster than the original code by 33.7%, and, finally, Inspector IV performed way better by 76.7%.

5.3 Challenges: Limitations and Proposed Solutions

As we have observed, Rec2Poly already optimizes some recursive programs. Yet, there are still opportunities to optimize it more, so it generates more optimized codes. While the executor codes are generally powerfully optimized, the inspectors generated by Rec2Poly may be costly and induce much overhead for some recursive codes. This can be investigated and ameliorated further.

In addition, not every recursive code behaves in an affine manner, thus not every recursive code can benefit from Rec2Poly. For now, Rec2Poly can handle recursive codes whose memory and control behavior is affine or parametrically-affine where the mem-

```
for i0 = 0 to 3555575
  for i1 = 0 to 2
    val walksub::BB1
  for i1 = 0 to 2
    val walksub::BB2
```

(a) NLR Model I

```
for i0 = 0 to 1507265
  for i1 = 0 to 2
    val gravsub::BB1
  for i1 = 0 to 2
    val gravsub::BB2
  for i1 = 0 to 2
    val gravsub::BB3
  for i1 = 0 to 2
    val gravsub::BB4
```

(b) NLR Model II

```
for i0 = 0 to 3555575
  val walksub::BB5
```

(c) NLR Model III

```
for i0 = 0 to 39
  for i1 = 0 to 2
    val walksub::BB1
  for i1 = 0 to 2
    val walksub::BB2
for i0 = 0 to 4
  for i1 = 0 to 2
    val gravsub::BB1
  for i1 = 0 to 2
    val gravsub::BB2
  for i1 = 0 to 2
    val gravsub::BB3
  for i1 = 0 to 2
    val gravsub::BB4
for i0 = 0 to 15
  for i1 = 0 to 2
    val walksub::BB1
  for i1 = 0 to 2
    val walksub::BB2
for i0 = 0 to 2
  for i1 = 0 to 2
    val gravsub::BB1
  for i1 = 0 to 2
    val gravsub::BB2
  for i1 = 0 to 2
    val gravsub::BB3
  for i1 = 0 to 2
    val gravsub::BB4
for i0 = 0 to 39
  for i1 = 0 to 2
    val walksub::BB1
  for i1 = 0 to 2
    val walksub::BB2
  ...
```

(d) NLR Model IV

```
for i0 = 0 to 7
  for i1 = 0 to 2
    val walksub::BB1
  for i1 = 0 to 2
    val walksub::BB2
for i0 = 0 to 4
  val walksub::BB3
for i0 = 0 to 7
  for i1 = 0 to 2
    val walksub::BB1
  for i1 = 0 to 2
    val walksub::BB2
for i0 = 0 to 3
  val walksub::BB3
for i0 = 0 to 2
  for i1 = 0 to 7
    for i2 = 0 to 2
      val walksub::BB1
    for i2 = 0 to 2
      val walksub::BB2
  val walksub::BB3
  val walksub::BB3
for i0 = 0 to 3
  val walksub::BB3
for i0 = 0 to 2
  val gravsub::BB1
for i0 = 0 to 2
  val gravsub::BB2
for i0 = 0 to 2
  val gravsub::BB3
for i0 = 0 to 2
  val gravsub::BB4
for i0 = 0 to 5
  val walksub::BB3
for i0 = 0 to 2
  val gravsub::BB1
for i0 = 0 to 2
  val gravsub::BB2
for i0 = 0 to 2
  val gravsub::BB3
for i0 = 0 to 2
  val gravsub::BB4
...
```

(e) NLR Model V

Figure 5.12 – Barnes Control Behavior NLR Modeling Experiments

ory addresses touched are *partially* affine. Nevertheless, Rec2Poly cannot handle codes whose control behavior cannot be modeled as loops; such codes have a control trace in which interleaved basic blocks yield a complicated NLR model. As an example for such a case, we study the recursive program Barnes from the REAPAR benchmark suite [114]. This program is very complex: it involves seven different recursions, i.e., there are seven strongly connected components (SCCs) in its call graph. When this program is profiled by Rec2Poly as any other recursive program, we get a complex NLR model showing interleaved executions of the impacting basic blocks.

For the sake of this illustrative example, we choose to analyze and profile only the control behavior of one recursive function in Barnes that is called "walksub" and its reachable function called "gravsub". We instrument these functions five different times to get different traces. The NLR modeling of these traces are displayed in Figure 5.12. The first NLR model shown, in Subfigure 5.12a, corresponds to the trace generated by instrumenting only two impacting basic blocks in Function "walksup". As we can see, this NLR model is made up of a "beautiful" loop nest. Note that, even if the memory addresses accessed in these impacting basic blocks are not shown in the NLR model here, they are actually defined as affine expressions of the surrounding loop indices. In Subfigure 5.12b, we show the NLR model of the execution trace of four impacting basic blocks in Function "gravsub" which is also an interesting affine loop nest. In Subfigure 5.12c, we show the NLR model of the execution trace of one basic block in Function "walksup" which is "BB3".

In Subfigure 5.12d, we have the NLR model of the trace including the basic blocks previously considered in Function "walksup" and those in Function "gravsub". This NLR model is much more complex than the first two models in which the basic blocks IDs of every function were modeled separately; it also includes a significantly longer sequence of affine loop nests. Furthermore, when the execution of the Basic Block "BB5" in Function "walksub" is also traced besides the other basic blocks, the corresponding NLR model obtained, in Subfigure 5.12e, becomes even more complex and larger than the previously shown models. It shows that the execution of this basic block interleaves with that of the others.

Therefore, if we consider profiling the control and memory behavior of the whole recursive code part by Rec2Poly, we surely do not get a neat loop model that helps performing any code transformation and generation.

Yet, as we can observe, when fewer basic blocks IDs and memory addresses are traced separately, it becomes possible to model them as "beautiful" loop nests. Accordingly, it is interesting to perform polyhedral modeling of recursive codes incrementally which may lead to a valid re-scheduling of the basic blocks based on dependence analysis. Such an approach may eventually yields an affine control behavior of a recursive code's execution; it can be referred to as *affinization* or *polyhedralization*

Another interesting modeling approach that can be also adopted by Rec2Poly is to consider modeling the behavior of recursive programs as loops having, as an upper bound, a variable or a list of parameters. As we can see, in Figure 5.12, in SubFigure 5.12b, the repeating loop nests in the NLR model are similar and include the same basic blocks IDs. The only difference among these loops is their different upper bound. This behavior is probably caused by the multiple invocation and activation of the same functions with different parameters each time, which affects the execution flow of their basic blocks and the memory access patterns. The model corresponding to the trace initially modeled as

shown in SubFigure 5.12d, would be similar to what is presented in figure 5.13. Instead of having, as an upper bound, a constant or an affine expression by NLR, we have a list of values. However, such an approach cannot be considered as polyhedral modeling of recursive functions, but adopting it to model, and transform recursive codes accordingly, may enable some interesting optimizations to recursions.

```
for i0 = 0 to P
  for i1 = 0 to [39,15,39,...]
    for i2 = 0 to 2
      val walksub::BB1
    for i1 = 0 to 2
      val walksub::BB2
  for i1 = 0 to [4,2,5,...]
    for i2 = 0 to 2
      val gravsub::BB1
    for i1 = 0 to 2
      val gravsub::BB2
    for i1 = 0 to 2
      val gravsub::BB3
    for i1 = 0 to 2
      val gravsub::BB4
```

Figure 5.13 – Recursion Behavior Loop Model with Variable Upper Bounds

Finally, in this chapter, through some experiments, we show that Rec2Poly succeeds already to optimize recursive programs and that it still has a great potential to perform even better and handle more complex recursive behaviors.

In the next chapter, we conclude and sum up what has been presented in this thesis.

Chapter 6

Conclusion and Perspectives

"Computer science is the operating system for all innovation." — Steve Ballmer

A lot of domains have thrived on computer science. Computers have conquered almost every innovation environment introducing considerably more efficiency. More efficiency is obviously better; the faster and better computers perform, the more efficiency we get and more.

To sum up, computer hardware has been the main target for amelioration and speedup for a long time now. Inventing multi-cores and multi-processors architectures has been a major milestone in computers lifetime. Yet, taking care of the hardware alone has not been enough to attain an optimal computer speedup. In this context, there have been dedicated many studies and works to optimize the software, adapt it to the underlying hardware and employ parallelism. Implementing parallel programs, for instance, has become possible through some languages and libraries, but implementing a reliable parallel code manually has remained a challenging mission for programmers. Accordingly, automating software optimization and parallelization has always been favored. Advanced compilers and static optimizers have become capable of automatically applying interesting software optimizations, e.g., tail recursive call elimination or even agressive polyhedral optimizations/parallelizations for statically-affine loops. Furthermore, through progressive research, it has become possible to unveil optimization opportunities, that are initially hidden at compile time, as soon as the run-time behavior is discovered. Such opportunities have permitted to automatically and speculatively transform, aggressively optimize and parallelize certain expensive control structures, exclusively loops. Existing static and dynamic loop structures optimization techniques have been thoroughly presented in this thesis in Chapter 2. On the other hand, as we have discussed in Chapter 3, although recursions and recursive functions are considered time-expensive structures as loops and can be transformed to loops, unlike loops, they have not benefited from similar advanced and dynamically-discovered optimizations opportunities.

In conclusion, this thesis has been devoted to diverge from the existing recursions optimization approaches that are static and probably classic, and create the opportunity to find out when recursive codes behave as "affine" loops in particular, and correspondingly allow a speculative rewriting of the recursive code parts to affine loops which are the best fitting candidates for efficient data locality and powerful optimization and parallelization.

6.1 Summary of Contributions

The main contribution of this thesis has been the framework Rec2Poly. To our knowledge, Rec2Poly is the first attempt of speculative program optimization involving the rewriting of the target code.

We have rigorously explained all about Rec2Poly in Chapter 4. In summary, it is a speculative polyhedral recursion optimizer based on Clang/LLVM that allows transforming recursive code parts to optimizable affine loop nests whenever possible. It involves a static analysis and code preparation phase, offline profiling phase and inspector-executor code generation phase.

Rec2Poly is pioneering because it is the first to seek a polyhedral-compliant run-time behavior in recursions. After it statically analyses the target source code for existing recursions, obtains the corresponding significant compile-time information, and prepares the code for the later phases, Rec2Poly performs its offline profiling phase. In this phase, Rec2Poly generates the control and memory trace of the recursive code part and feeds it to the NLR algorithm. The NLR algorithm was mostly used to model memory accesses, yet Rec2Poly uses it to model the control behavior of a program too. If NLR succeeds to produce an affine loop model, this model can then be used to guide the rewriting of the target recursive code for following executions.

Moreover, what distinguishes Rec2Poly is that it extends the use of the inspector-executor paradigm. Since the code generation phase of Rec2Poly is dependent on offline profiling, the validity of the NLR loops must be proved in every new execution context. Accordingly, Rec2Poly generates a code that implements an inspector-executor dedicated strategy.

Before Rec2Poly, this methodology has been exclusively adopted to speculatively optimize and monitor loop transformations and verify memory access patterns. However, Rec2Poly uses this paradigm in the realm of recursions as the inspector part involves recursive code parts. Also, Rec2Poly generates a fast parallel inspector that verifies the control flow against that of the affine loops in the NLR model corresponding to a previous execution besides the memory access patterns.

If the predictive loop model is proved to hold in the current execution context by the inspector, then the executor containing the affine loops equivalent to the recursion is launched.

Since the loops generated by Rec2Poly in the executor are affine or parametrically-affine, they can benefit from aggressive polyhedral optimizations and parallelization by a polyhedral optimizer (Pluto) or be parallelized after a possible accurate dependence analysis is performed.

Finally, we have shown, through the experiments presented in Chapter 5 on the recursive Matrix Multiplication and Heat programs, that with such an approach, some recursive programs may benefit from efficient affine loops optimization and parallelization, and even advanced transformations of the polyhedral model.

6.2 Future Perspectives

Rec2Poly is a promising framework for speculative rewriting of recursive programs as loop candidates for efficient parallelization and optimization using an inspector-executor

mechanism. However, there is still some work to be done to complete this framework especially at the executor level, add more features and further improve the performance of the generated inspector-executor code especially the inspector code.

While the inspector-executor mechanism is adapted to such speculative optimizations, the final performance of the generated code is mostly relying on the performance of the inspector. This issue is solvable in case we have recursive code parts involving flow-non-impacting loops; this is because we have added to Rec2Poly an inspector optimization option that, when activated, allows breaking such loops in the generated inspector components (trace generators and verifiers). These loops may be responsible for a great part of the inspector execution, so terminating them after a small number of iterations, that is enough to guarantee the execution flow correctness, may significantly reduce the time overhead in the inspector. Yet, this case is not the general case; there are recursive codes parts accessing memory without containing any loops, e.g., the first matrix multiplication program experimented in Chapter 5. The inspectors of such codes cannot be aggressively optimized and the performance of the final inspector-executor codes generated remain dependent on the inspectors, while the executors are probably much faster. Also, some recursive codes, like the first recursive matrix multiplication implementation we presented, may exhibit a similar loop behavior for all input; the produced NLR loop models for all the executions consist of similar loop nests with same depths embedding the same trace generating affine functions, but only having different upper bounds according to the program input sizes. The inspector should be smarter when handling such codes. Therefore, in the future, more inspector optimization strategies should be investigated to lower further the inspector time-overhead.

Moreover, it would be interesting to implement an online profiling technique, based on sampling and on a prediction, and transform the codes on-the-fly as Apollo does when optimizing loops.

Furthermore, not all recursive codes may exhibit a linear control behavior like some may not exhibit a linear memory behavior. In this thesis, we handled the situation where the memory accesses are not fully affine. However, as for the codes with non-affine control behavior, there are interesting proposals that can be studied in the future. For instance, it would be interesting to handle sequences of recursion traces in which there are repetitive trace parts that repeat sometimes for a different number of times than the others, or traces of loops, already existing in the code in impacting functions, whose upper bounds differ every time the functions are activated again. Such traces may fit into loop nests with variable upper bounds; they are obviously not affine, but there may still be a clever way to parallelize them.

Also, the possibility to reschedule some instructions, in the target recursive code, that are initially yielding a non-affine control trace should be investigated; we may consequently get an affine control behavior of the code's execution. Such an approach could be called *affinization* or *polyhedralization*, i.e., code rewriting for affine or polyhedral runtime behavior.

In the end of this thesis, Rec2Poly is just a beginning. Rec2Poly is a revolution that has just begun in the realm of recursions; and more generally and more ambitiously, in the realm of dynamic code rewriting for efficient optimizations.

Bibliography

Personal Bibliography

[66] Salwa Kobeissi and Philippe Clauss. "The Polyhedral Model Beyond Loops Recursion Optimization and Parallelization Through Polyhedral Modeling". In: *IMPACT 2019 - 9th International Workshop on Polyhedral Compilation Techniques, In conjunction with HiPEAC 2019*. Valencia, Spain, Jan. 2019. URL: https://hal.inria.fr/hal-02059558.

[67] Salwa Kobeissi, Alain Ketterlin, and Philippe Clauss. "Rec2Poly: Converting Recursions to Polyhedral Optimized Loops Using an Inspector-Executor Strategy". In: *SAMOS 2020: Embedded Computer Systems: Architectures, Modeling, and Simulation*. Oct. 2020, pp. 96–109. DOI: 10.1007/978-3-030-60939-9_7. URL: https://hal.inria.fr/hal-02971434.

General Bibliography

[1] Samer Abdallah. *Memoisation: Purely, Left-recursively, and with (Continuation Passing) Style*. 2017. arXiv: 1707.04724 [cs.LO].

[2] Aravind Acharya and Uday Bondhugula. "PLUTO+: Near-Complete Modeling of Affine Transformations for Parallelism and Locality". In: *SIGPLAN Not.* 50.8 (Jan. 2015), pp. 54–64. ISSN: 0362-1340. DOI: 10.1145/2858788.2688512. URL: https://doi.org/10.1145/2858788.2688512.

[3] Gagan Agrawal, Joel Saltz, and Raja Das. "Interprocedural Partial Redundancy Elimination and Its Application to Distributed Memory Compilation". In: *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*. PLDI '95. La Jolla, California, USA: Association for Computing Machinery, 1995, pp. 258–269. ISBN: 0897916972. DOI: 10.1145/207110.207157. URL: https://doi.org/10.1145/207110.207157.

[4] Pierre Amiranoff, Albert Cohen, and Paul Feautrier. "Beyond Iteration Vectors: Instancewise Relational Abstract Domains". In: *Static Analysis*. Ed. by Kwangkeun Yi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 161–180. ISBN: 978-3-540-37758-0.

[5] J. Arsac and Y. Kodratoff. "Some Techniques for Recursion Removal from Recursive Functions". In: *ACM Trans. Program. Lang. Syst.* 4.2 (Apr. 1982), pp. 295–322. ISSN: 0164-0925.

[6] J. Arsac and Y. Kodratoff. "Some Techniques for Recursion Removal from Recursive Functions". In: *ACM Trans. Program. Lang. Syst.* 4.2 (Apr. 1982), pp. 295–322. ISSN: 0164-0925. DOI: 10.1145/357162.357171. URL: https://doi.org/10.1145/357162.357171.

[7] Utpal K. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations*. USA: Kluwer Academic Publishers, 1993. ISBN: 079239318X.

[8] Pablo Barrio, Chandler Carruth, and James Molloy. *Recursion Inlining In LLVM*. https://llvm.org/devmtg/2015-04/slides/recursion-inlining-2015.pdf. 2015.

[9] Cédric Bastoul. *Generating loops for scanning polyhedra: CLooG user's guide*. Tech. rep. 2002.

[10] Cédric Bastoul. "Code Generation in the Polyhedral Model Is Easier Than You Think". In: *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*. Juan-les-Pins, France, Sept. 2004, pp. 7–16.

[11] Cédric Bastoul. *Improving Data Locality in Static Control Programs*. 2004.

[12] Cédric Bastoul. *OpenScop: A Specification and a Library for Data Exchange in Polyhedral Compilation Tools*. Tech. rep. University of Paris-Sud, France, Sept. 2011.

[13] Cédric Bastoul. *Contributions to High-Level Program Optimization. Habilitation Thesis. Paris-Sud University, France*. Dec. 2012.

[14] Cédric Bastoul, Albert Cohen, Sylvain Girbal, Saurabh Sharma, and Olivier Temam. "Putting Polyhedral Loop Transformations to Work". In: *LCPC'16 International Workshop on Languages and Compilers for Parallel Computers, LNCS 2958*. College Station, Texas, Oct. 2003, pp. 209–225.

[15] D. Baxter, R. Mirchandaney, and J. H. Saltz. "Run-Time Parallelization and Scheduling of Loops". In: *Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA '89. Santa Fe, New Mexico, USA: Association for Computing Machinery, 1989, pp. 303–312. ISBN: 089791323X. DOI: 10.1145/72935.72967. URL: https://doi.org/10.1145/72935.72967.

[16] Richard Bellman. *Dynamic Programming*. USA: Princeton University Press, 2010. ISBN: 0691146683.

[17] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. "The Polyhedral Model Is More Widely Applicable Than You Think". In: *Compiler Construction*. Ed. by Rajiv Gupta. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 283–303.

[18] A. J. Bernstein. "Analysis of Programs for Parallel Processing". In: *IEEE Transactions on Electronic Computers* EC-15.5 (1966), pp. 757–763. DOI: 10.1109/PGEC.1966.264565.

[19] R. S. Bird. "Tabulation Techniques for Recursive Programs". In: *ACM Comput. Surv.* 12.4 (Dec. 1980), pp. 403–417. ISSN: 0360-0300. DOI: 10.1145/356827.356831. URL: https://doi.org/10.1145/356827.356831.

[20] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. "Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model". In: *Compiler Construction*. Ed. by Laurie Hendren. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 132–146.

[21] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. "A practical automatic polyhedral parallelizer and locality optimizer". In: *PLDI '08*. Tucson, AZ, USA: ACM, 2008, pp. 101–113. ISBN: 978-1-59593-860-2.

[22] Pierre Boulet, Alain Darte, Georges-André Silber, and Frédéric Vivien. "Loop Parallelization Algorithms: From Parallelism Extraction to Code Generation". In: *Parallel Comput.* 24.3–4 (May 1998), pp. 421–444. ISSN: 0167-8191. DOI: 10.1016/S0167-8191(98)00020-9. URL: https://doi.org/10.1016/S0167-8191(98)00020-9.

[23] Derek Bruening, Srikrishna Devabhaktuni, and Saman Amarasinghe. "Softspec: Software-based Speculative Parallelism". In: *In 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3*. 1998.

[24] R. M. Burstall and John Darlington. "A Transformation System for Developing Recursive Programs". In: *J. ACM* 24.1 (Jan. 1977), pp. 44–67. ISSN: 0004-5411. DOI: 10.1145/321992.321996. URL: https://doi.org/10.1145/321992.321996.

[25] *C-Memo*. URL: https://sourceforge.net/projects/c-memo/.

[26] Pierre-Yves Calland, Alain Darte, Yves Robert, and Frédéric Vivien. "Plugging anti and output dependence removal techniques into loop parallelization algorithm". In: *Parallel Computing* 23.1 (1997). Environment and tools for parallel scientific computing, pp. 251–266. ISSN: 0167-8191. DOI: https://doi.org/10.1016/S0167-8191(96)00108-1. URL: http://www.sciencedirect.com/science/article/pii/S0167819196001081.

[27] J. W. Cannon, W. J. Floyd, and W. R. Parry. "Finite Subdivision Rules". In: *Conform. Geom. Dyn* 5 (2001), pp. 153–196.

[28] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. "Rock: A High-Performance Sparc CMT Processor". In: *IEEE Micro* 29.2 (2009), pp. 6–16. DOI: 10.1109/MM.2009.34.

[29] Chun Chen, Jacqueline Chame, and Mary Hall. "A Framework for Composing High-Level Loop Transformations". In: (July 2008).

[30] *Clang: a C language family frontend for LLVM*. https://clang.llvm.org/.

[31] Albert Cohen and Jean-François Collard. "Instance-wise Reaching Definition Analysis for Recursive Programs using Context-free Transductions". In: *Parallel Architectures and Compilation Techniques (PACT*). Best student paper award. Paris, France, 1998, pp. 332–340. URL: https://hal.archives-ouvertes.fr/hal-01257320.

[32] Albert Cohen, Sylvain Girbal, and Olivier Temam. "A Polyhedral Approach to Ease the Composition of Program Transformations". In: *Euro-Par 2004 Parallel Processing*. Ed. by Marco Danelutto, Marco Vanneschi, and Domenico Laforenza. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 292–303.

[33] Albert Cohen, Marc Sigler, David Parello, Sylvain Girbal, Olivier Temam, and Nicolas Vasilache. “Facilitating the Search for Compositions of Program Transformations”. In: *In ACM Int. Conf. on Supercomputing (ICS’05*. 2005, pp. 151–160.

[34] Rebecca L. Collins, Bharadwaj Vellore, and Luca P. Carloni. “Recursion-driven Parallel Code Generation for Multi-core Platforms”. In: *Proceedings of the Conference on Design, Automation and Test in Europe*. DATE ’10. Dresden, Germany: European Design and Automation Association, 2010, pp. 190–195. ISBN: 978-3-9810801-6-2. URL: http://dl.acm.org/citation.cfm?id=1870926.1870972.

[35] L. Dagum and R. Menon. “OpenMP: an industry standard API for shared-memory programming”. In: *IEEE Computational Science and Engineering* 5.1 (1998), pp. 46–55. DOI: 10.1109/99.660313.

[36] Raja Das, Mustafa Uysal, Joel Saltz, and Yuan-Shin Hwang. *Communication Optimizations for Irregular Scientific Computations on Distributed Memory Architectures*. Tech. rep. USA, 1993.

[37] R. H. Dennard, F. H. Gaensslen, H. Yu, 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 (1974), pp. 256–268.

[38] Chen Ding and Ken Kennedy. “Improving Cache Performance in Dynamic Applications through Data and Computation Reorganization at Run Time”. In: *SIGPLAN Not.* 34.5 (May 1999), pp. 229–241. ISSN: 0362-1340. DOI: 10.1145/301631.301670. URL: https://doi.org/10.1145/301631.301670.

[39] Paul Feautrier. “Parametric Integer Programming”. In: *RAIRO Recherche opérationnelle* 22.3 (1988), pp. 243–268. URL: http://camlunity.ru/swap/Library/Conflux/Techniques%20-%20Code%20Analysis%20and%20Transformations%20%28Polyhedral%29/Integer%20Programming/parametric_integer_programming.pdf.

[40] Paul Feautrier. “Dataflow Analysis of Array and Scalar References”. In: *International Journal of Parallel Programming* 20.1 (1991), pp. 23–53.

[41] Paul Feautrier. “Some efficient solutions to the affine scheduling problem Part I One-dimensional Time”. In: *International Journal of Parallel Programming* 21 (Aug. 1996). DOI: 10.1007/BF01407835.

[42] Paul Feautrier. “Some efficient solutions to the affine scheduling problem Part II Multidimensional time”. In: *International Journal of Parallel Programming* 21 (Jan. 1997). DOI: 10.1007/BF01379404.

[43] Paul Feautrier. “A Parallelization Framework for Recursive Tree Programs”. In: *Proceedings of the 4th International Euro-Par Conference on Parallel Processing*. Euro-Par ’98. Berlin, Heidelberg: Springer-Verlag, 1998, pp. 470–479. ISBN: 3540649522.

[44] Paul Feautrier and Christian Lengauer. “Polyhedron Model”. In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Springer US, 2011, pp. 1581–1592. ISBN: 978-0-387-09765-7.

[45] Message P Forum. *MPI: A Message-Passing Interface Standard*. Tech. rep. USA, 1994.

[46] Richard A. Frost and Rahmatullah Hafiz. "A New Top-down Parsing Algorithm to Accommodate Ambiguity and Left Recursion in Polynomial Time". In: *SIGPLAN Not.* 41.5 (May 2006), pp. 46–54. ISSN: 0362-1340. DOI: 10.1145/1149982.1149988. URL: https://doi.org/10.1145/1149982.1149988.

[47] *functools — Higher-order functions and operations on callable objects*. https://docs.python.org/3/library/functools.html.

[48] *GCC, the GNU Compiler Collection*. https://gcc.gnu.org/.

[49] 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 (June 2006), pp. 261–317. DOI: 10.1007/s10766-006-0012-3.

[50] Samuel Greengard. "Can Nanosheet Transistors Keep Moore's Law Alive?" In: *Commun. ACM* 63.3 (Feb. 2020), pp. 10–12. ISSN: 0001-0782. DOI: 10.1145/3379493. URL: https://doi.org/10.1145/3379493.

[51] Martin Griebl. *Automatic Parallelization of Loop Programs for Distributed Memory Architectures*. 2004.

[52] Tobias Grosser, Armin Größlinger, and Christian Lengauer. "Polly – Performing polyhedral optimizations on a low-level intermediate representation". In: *Parallel Processing Letters* 22.04 (2012).

[53] Manish Gupta, Sayak Mukhopadhyay, and Navin Sinha. "Automatic Parallelization of Recursive Procedures". In: *International Journal of Parallel Programming* 28.6 (Dec. 2000), pp. 537–562.

[54] Suyash Gupta, Rahul Shrivastava, and V Krishna Nandivada. "Optimizing Recursive Task Parallel Programs". In: *Proceedings of the International Conference on Supercomputing*. ICS '17. Chicago, Illinois: ACM, 2017, 11:1–11:11. ISBN: 978-1-4503-5020-4. DOI: 10.1145/3079079.3079102. URL: http://doi.acm.org/10.1145/3079079.3079102.

[55] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, a. gara, G. Chiu, P. Boyle, N. Chist, and C. Kim. "The IBM Blue Gene/Q Compute Chip". In: *IEEE Micro* 32.2 (2012), pp. 48–60. DOI: 10.1109/MM.2011.108.

[56] Peter G. Harrison and Hessam Khoshnevisan. "A new approach to recursion removal". In: *Theoretical Computer Science* 93.1 (1992), pp. 91–113. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(92)90213-Y. URL: http://www.sciencedirect.com/science/article/pii/030439759290213Y.

[57] Alexandra Jimborean. "Adapting the polytope model for dynamic and speculative parallelization". In: (Sept. 2012).

[58] Alexandra Jimborean, Luis Mastrangelo, Vincent Loechner, and Philippe Clauss. "VMAD: an Advanced Dynamic Program Analysis & Instrumentation Framework". In: *CC - 21st International Conference on Compiler Construction*. Ed. by M. O'Boyle. Vol. 7210. Lecture Notes in Computer Science. Tallinn, Estonia: Springer, Mar. 2012, pp. 220–237. URL: https://hal.inria.fr/hal-00664345.

[59] Youngjoon Jo and Milind Kulkarni. "Enhancing Locality for Recursive Traversals of Recursive Structures". In: *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '11. Portland, Oregon, USA: Association for Computing Machinery, 2011, pp. 463–482. ISBN: 9781450309400. DOI: 10.1145/2048066.2048104. URL: https://doi.org/10.1145/2048066.2048104.

[60] Youngjoon Jo and Milind Kulkarni. "Automatically Enhancing Locality for Tree Traversals with Traversal Splicing". In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '12. Tucson, Arizona, USA: Association for Computing Machinery, 2012, pp. 355–374. ISBN: 9781450315616. DOI: 10.1145/2384616.2384643. URL: https://doi.org/10.1145/2384616.2384643.

[61] Richard M Karp, Raymond E Miller, and Shmuel Winograd. "The Organization of Computations for Uniform Recurrence Equations". In: *Journal of the ACM* 14.3 (1967), pp. 563–590. URL: http://dl.acm.org/citation.cfm?id=321418.

[62] W. Kelly and W. Pugh. "A unifying framework for iteration reordering transformations". In: *Proceedings 1st International Conference on Algorithms and Architectures for Parallel Processing*. Vol. 1. 1995, 153–162 vol.1. DOI: 10.1109/ICAPP.1995.472180.

[63] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and Dave Wonnacott. "The Omega Calculator and Library, Version 1.1.0". In: (1996). URL: http://www.cs.utah.edu/~mhall/cs6963s09/lectures/omega.ps.

[64] Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001. ISBN: 1558602860.

[65] Alain Ketterlin and Philippe Clauss. "Prediction and Trace Compression of Data Access Addresses Through Nested Loop Recognition". In: *Proceedings of the 6th IEEE/ACM International Symposium on Code Generation and Optimization*. CGO'08. Boston, MA, USA: ACM, 2008, pp. 94–103.

[66] Salwa Kobeissi and Philippe Clauss. "The Polyhedral Model Beyond Loops Recursion Optimization and Parallelization Through Polyhedral Modeling". In: *IMPACT 2019 - 9th International Workshop on Polyhedral Compilation Techniques, In conjunction with HiPEAC 2019*. Valencia, Spain, Jan. 2019. URL: https://hal.inria.fr/hal-02059558.

[67] Salwa Kobeissi, Alain Ketterlin, and Philippe Clauss. "Rec2Poly: Converting Recursions to Polyhedral Optimized Loops Using an Inspector-Executor Strategy". In: *SAMOS 2020: Embedded Computer Systems: Architectures, Modeling, and Simulation*. Oct. 2020, pp. 96–109. DOI: 10.1007/978-3-030-60939-9_7. URL: https://hal.inria.fr/hal-02971434.

[68] C. Koelbel, P. Mehrotra, and J. Van Rosendale. "Supporting Shared Data Structures on Distributed Memory Architectures". In: *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*. PPOPP '90. Seattle, Washington, USA: Association for Computing Machinery, 1990, pp. 177–186. ISBN: 0897913507. DOI: 10.1145/99163.99183. URL: https://doi.org/10.1145/99163.99183.

[69] Christian Lengauer. "Polly—Performing Polyhedral Optimizations on a Low-Level Intermediate Representation". In: *Parallel Processing Letters* 22 (Dec. 2012). DOI: 10.1142/S0129626412500107.

[70] Shun-Tak Leung and John Zahorjan. "Improving the Performance of Runtime Parallelization". In: *SIGPLAN Not.* 28.7 (July 1993), pp. 83–91. ISSN: 0362-1340. DOI: 10.1145/173284.155341. URL: https://doi.org/10.1145/173284.155341.

[71] Amy W. Lim, Gerald I. Cheong, and Monica S. Lam. "An Affine Partitioning Algorithm to Maximize Parallelism and Minimize Communication". In: *Proceedings of the 13th International Conference on Supercomputing*. ICS '99. Rhodes, Greece: Association for Computing Machinery, 1999, pp. 228–237. ISBN: 158113164X. DOI: 10.1145/305138.305197. URL: https://doi.org/10.1145/305138.305197.

[72] 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. Paris, France: Association for Computing Machinery, 1997, pp. 201–214. ISBN: 0897918533. DOI: 10.1145/263699.263719. URL: https://doi.org/10.1145/263699.263719.

[73] Amy Wingmui Lim. "Improving Parallelism and Data Locality with Affine Partitioning". AAI3028136. PhD thesis. 2001. ISBN: 0493404236.

[74] Wei Liu, James Tuck, Luís Ceze, Wonsun Ahn, Karin Strauss, José Renau, and Josep Torrellas. "POSH: A TLS Compiler that Exploits Program Structure". In: *PPoPP 2006* (*Principles and Practice of Parallel Programming*). Association for Computing Machinery, Inc., Mar. 2006. URL: https://www.microsoft.com/en-us/research/publication/posh-a-tls-compiler-that-exploits-program-structure/.

[75] Y. A. Liu. "CACHET: an interactive, incremental-attribution-based program transformation system for deriving incremental programs". In: *Proceedings 1995 10th Knowledge-Based Software Engineering Conference*. 1995, pp. 19–26. DOI: 10.1109/KBSE.1995.490115.

[76] Yanhong A. Liu and Scott D. Stoller. "Dynamic programming via static incrementalization". In: *In Proceedings of the 8th European Symposium on Programming*. Springer-Verlag, 1999, pp. 288–305.

[77] Yanhong A. Liu and Scott D. Stoller. "From Recursion to Iteration: What Are the Optimizations?" In: *SIGPLAN Not.* 34.11 (Nov. 1999), pp. 73–82. ISSN: 0362-1340. DOI: 10.1145/328691.328700. URL: https://doi.org/10.1145/328691.328700.

[78] *LLVM Language Reference Manual*. https://llvm.org/docs/LangRef.html.

[79] *LLVM Loop Terminology (and Canonical Forms)*. https://llvm.org/docs/LoopTerminology.html.

[80] *LLVM's Analysis and Transform Passes*. https://llvm.org/docs/Passes.html.

[81] Vincent Loechner. *PolyLib: A library for manipulating parameterized polyhedra*. 1999. URL: https://repo.or.cz/polylib.git/blob_plain/HEAD:/doc/parampoly-doc.ps.gz.

[82] Benoit B Mandelbrot. *The fractal geometry of nature*. San Francisco, CA: Freeman, 1982. URL: https://cds.cern.ch/record/98509.

[83] Juan Manuel Martinez Caamaño. "Fast and Flexible Compilation Techniques for Effective Speculative Polyhedral Parallelization". Theses. Université de Strasbourg, Sept. 2016. URL: https://hal.inria.fr/tel-01377758.

[84] Juan Manuel Martinez Caamano, Manuel Selva, Philippe Clauss, Artiom Baloian, and Willy Wolff. "Full runtime polyhedral optimizing loop transformations with the generation, instantiation, and scheduling of code-bones". In: *Concurrency and Computation: Practice and Experience* 29.15 (June 2017).

[85] Aristeidis Mastoras and George Manis. "Ariadne - Directive-based Parallelism Extraction from Recursive Functions". In: *J. Parallel Distrib. Comput.* 86.C (Dec. 2015), pp. 16–28. ISSN: 0743-7315.

[86] *Matrix Multiplication | Recursive*. https://www.geeksforgeeks.org/matrix-multiplication-recursive/.

[87] J. Mayfield, T. Finin, and M. Hall. "Using Automatic Memoization as a Software Engineering Tool in Real-World AI Systems". In: CAIA '95. USA: IEEE Computer Society, 1995, p. 87. ISBN: 0818670703.

[88] D. MICHIE. ""Memo" Functions and Machine Learning". In: (1968). DOI: https://doi.org/10.1038/218306c0.

[89] R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nico, and K. Crowley. "Principles of Runtime Support for Parallel Processors". In: *Proceedings of the 2nd International Conference on Supercomputing*. ICS '88. St. Malo, France: Association for Computing Machinery, 1988, pp. 140–152. ISBN: 0897912721. DOI: 10.1145/55364.55378. URL: https://doi.org/10.1145/55364.55378.

[90] N. Mitchell, L. Carter, and J. Ferrante. "Localizing non-affine array references". In: *1999 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.PR00425)*. 1999, pp. 192–202. DOI: 10.1109/PACT.1999.807526.

[91] Yasuharu Mizutani, Daisuke Nakajima, Noriyuki Fujimoto, and Kenichi Hagihara. "Evaluation of a compiler with user-selectable execution strategies for parallel recursion". In: *Systems and Computers in Japan* 35.9 (), pp. 92–103. DOI: 10.1002/scj.10009. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/scj.10009. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/scj.10009.

[92] G. E. Moore. "Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff." In: *IEEE Solid-State Circuits Society Newsletter* 11.3 (2006), pp. 33–35.

[93] Gordon Moore. "Progress In Digital Integrated Electronics [Technical literaiture, Copyright 1975 IEEE. Reprinted, with permission. Technical Digest. International Electron Devices Meeting, IEEE, 1975, pp. 11-13.]" In: *Solid-State Circuits Newsletter, IEEE* 20 (Oct. 2006), pp. 36–37. DOI: 10.1109/N-SSC.2006.4804410.

[94] Akimasa Morihata and Kiminori Matsuzaki. "Automatic Parallelization of Recursive Functions Using Quantifier Elimination". In: *Proceedings of the 10th International Conference on Functional and Logic Programming*. FLOPS'10. Sendai, Japan: Springer-Verlag, 2010, pp. 321–336. ISBN: 3-642-12250-7, 978-3-642-12250-7. DOI: 10.1007/978-3-642-12251-4_23. URL: http://dx.doi.org/10.1007/978-3-642-12251-4_23.

[95] Peter Norvig. "Techniques for Automatic Memoization with Applications to Context-Free Parsing". In: *Comput. Linguist.* 17.1 (Mar. 1991), pp. 91–98. ISSN: 0891-2017.

[96] *onlinefractaltools*. https://onlinefractaltools.com/draw-levy-fractal.

[97] *Over 50 years of Moore's Law - Intel*. https://www.intel.com/content/www/us/en/silicon-innovations/moores-law-technology.html.

[98] David Padua. "POSIX Threads (Pthreads)". In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Boston, MA: Springer US, 2011, pp. 1592–1593. ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_447. URL: https://doi.org/10.1007/978-0-387-09766-4_447.

[99] Eunjung Park, Louis-Noel Pouche, John Cavazos, Albert Cohen, and P. Sadayappan. "Predictive Modeling in a Polyhedral Optimization Space". In: *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '11. USA: IEEE Computer Society, 2011, pp. 119–129. ISBN: 9781612843568.

[100] Dmitry Petrashko, Ondřej Lhoták, and Martin Odersky. "Miniphases: Compilation Using Modular and Efficient Tree Transformations". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: Association for Computing Machinery, 2017, pp. 201–216. ISBN: 9781450349888. DOI: 10.1145/3062341.3062346. URL: https://doi.org/10.1145/3062341.3062346.

[101] *PLUTO - An automatic parallelizer and locality optimizer for multicores*. http://pluto-compiler.sourceforge.net.

[102] *Polly - LLVM Framework for High-Level Loop and Data-Locality Optimizations*. https://polly.llvm.org/index.html.

[103] R. Ponnusamy, J. Saltz, and A. Choudhary. "Runtime Compilation Techniques for Data Partitioning and Communication Schedule Reuse". In: *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*. Supercomputing '93. Portland, Oregon, USA: Association for Computing Machinery, 1993, pp. 361–370. ISBN: 0818643404. DOI: 10.1145/169627.169752. URL: https://doi.org/10.1145/169627.169752.

[104] S. Pop, A. Cohen, C. Bastoul, Sylvain Girbal, G. Silber, and Nicolas Vasilache. "GRAPHITE: Loop Optimizations Based on the Polyhedral Model for GCC". In: 2006.

[105] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and Nicolas Vasilache. "Iterative Optimization in the Polyhedral Model: Part I, One-Dimensional Time". In: *Fifth International Symposium on Code Generation and Optimization (CGO 2007), 11-14 March 2007, San Jose, California, USA*. IEEE Computer Society, 2007, pp. 144–156. DOI: 10.1109/CGO.2007.21. URL: https://doi.org/10.1109/CGO.2007.21.

[106] 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. Albuquerque, New Mexico, USA: Association for Computing Machinery, 1991, pp. 4–13. ISBN: 0897914597. DOI: 10.1145/125826.125848. URL: https://doi.org/10.1145/125826.125848.

[107] William Pugh. "Uniform Techniques for Loop Optimization". In: *5th International Conference on Supercomputing (ICS'91)*. ACM. 1991, pp. 341–352.

[108] Samyam Rajbhandari, Jinsung Kim, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, Robert J. Harrison, and P. Sadayappan. "On Fusing Recursive Traversals of K-d Trees". In: *Proceedings of the 25th International Conference on Compiler Construction*. CC 2016. Barcelona, Spain: Association for Computing Machinery, 2016, pp. 152–162. ISBN: 9781450342414. DOI: 10.1145/2892208.2892228. URL: https://doi.org/10.1145/2892208.2892228.

[109] Lawrence Rauchwerger. "Run-time parallelization: Its time has come". In: *Parallel Computing* 24.3 (1998), pp. 527–556. ISSN: 0167-8191. DOI: https://doi.org/10.1016/S0167-8191(98)00024-6. URL: https://www.sciencedirect.com/science/article/pii/S0167819198000246.

[110] Lawrence Rauchwerger, Nancy Amato, and David Padua. "A Scalable Method for Run-Time Loop Parallelization". In: *International Journal of Parallel Programming* 23 (Feb. 2003). DOI: 10.1007/BF02577866.

[111] Lawrence Rauchwerger and David Padua. "The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization". In: 1995, pp. 218–232.

[112] Lawrence Rauchwerger and David A Padua. "The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization". In: *IEEE Transactions on Parallel and Distributed Systems* 10.2 (1999), pp. 160–180.

[113] Mahesh Ravishankar, Roshan Dathathri, Venmugil Elango, Louis-Noël Pouchet, J. Ramanujam, Atanas Rountev, and P. Sadayappan. "Distributed Memory Code Generation for Mixed Irregular/Regular Computations". In: *SIGPLAN Not.* 50.8 (Jan. 2015), pp. 65–75. ISSN: 0362-1340. DOI: 10.1145/2858788.2688515. URL: https://doi.org/10.1145/2858788.2688515.

[114] *REAPAR, Automatic Parallelization of Irregular Recursive Programs*. http://www.haenssgen.de/uni/reapar.

[115] Eric S. Roberts, ed. *Thinking Recursively*. USA: John Wiley & Sons, Inc., 1986. ISBN: 0471816523.

[116] Manuel Rubio-Sanchez. *Introduction to Recursive Programming*. 1st. USA: CRC Press, Inc., 2017. ISBN: 1498735282.

[117] Radu Rugina and Martin Rinard. "Automatic Parallelization of Divide and Conquer Algorithms". In: *SIGPLAN Not.* 34.8 (May 1999), pp. 72–83. ISSN: 0362-1340. DOI: 10.1145/329366.301111. URL: http://doi.acm.org/10.1145/329366.301111.

[118] Laith Sakka, Kirshanthan Sundararajah, and Milind Kulkarni. "TreeFuser: A Framework for Analyzing and Fusing General Recursive Tree Traversals". In: *Proc. ACM Program. Lang.* 1.OOPSLA (Oct. 2017). DOI: 10.1145/3133900. URL: https://doi.org/10.1145/3133900.

[119] J. H. Saltz, R. Mirchandaney, and K. Crowley. "Run-time parallelization and scheduling of loops". In: *IEEE Transactions on Computers* 40.5 (1991), pp. 603–612.

[120] J. Saltz and R. Mirchandaney. "The Preprocessed Doacross Loop". In: *ICPP*. 1991.

[121] Joel Saltz, Chialin Chang, Guy Edjlali, Yuan-Shin Hwang, Bongki Moon, Ravi Ponnusamy, Shamik Sharma, Alan Sussman, Mustafa Uysal, Gagan Agrawal, Raja Das, and Paul Havlak. "Programming Irregular Applications: Runtime Support, Compilation and Tools". In: *Emphasizing Parallel Programming Techniques*. Ed. by Marvin V. Zelkowitz. Vol. 45. Advances in Computers. Elsevier, 1997, pp. 105–153. DOI: https://doi.org/10.1016/S0065-2458(08)60707-X. URL: https://www.sciencedirect.com/science/article/pii/S006524580860707X.

[122] Dimitris Saougkos, Aristeidis Mastoras, and George Manis. "Fine Grained Parallelism in Recursive Function Calls". In: *Parallel Processing and Applied Mathematics*. Ed. by Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Waśniewski. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 121–130. ISBN: 978-3-642-31500-8.

[123] Alexander Schrijver. *Theory of Linear and Integer Programming*. USA: John Wiley & Sons, Inc., 1986. ISBN: 0471908541.

[124] I. Šimeček and P. Tvrdík. "High Performance Recursive Linear Algebra Library". English. In: *Seminar on Numerical Analysis*. Ostrava: Ústav geonomy AV ČR, 2007, pp. 116–119. ISBN: 80-86407-12-8.

[125] J. G. Steffan and T. C. Mowry. "The potential for using thread-level data speculation to facilitate automatic parallelization". In: *Proceedings 1998 Fourth International Symposium on High-Performance Computer Architecture*. 1998, pp. 2–13. DOI: 10.1109/HPCA.1998.650541.

[126] J. Greggory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. "A Scalable Approach to Thread-Level Speculation". In: *Proceedings of the 27th Annual International Symposium on Computer Architecture*. ISCA '00. Vancouver, British Columbia, Canada: Association for Computing Machinery, 2000, pp. 1–12. ISBN: 1581132328. DOI: 10.1145/339647.339650. URL: https://doi.org/10.1145/339647.339650.

[127] Greg Stitt and Jason Villarreal. “Recursion Flattening”. In: *Proceedings of the 18th ACM Great Lakes Symposium on VLSI*. GLSVLSI ’08. Orlando, Florida, USA: Association for Computing Machinery, 2008, pp. 131–134. ISBN: 9781595939999. DOI: 10.1145/1366110.1366143. URL: https://doi.org/10.1145/1366110.1366143.

[128] Gilbert Strang. *Introduction to Linear Algebra*. Fourth. Wellesley, MA: Wellesley-Cambridge Press, 2009. ISBN: 9780980232714 0980232716 9780980232721 0980232724 9788175968110 8175968117.

[129] Michelle Mills Strout, Mary Hall, and Catherine Olschanowsky. “The Sparse Polyhedral Framework: Composing Compiler-Generated Inspector-Executor Code”. In: *Proceedings of the IEEE* 106.11 (2018), pp. 1921–1934. DOI: 10.1109/JPROC.2018.2857721.

[130] Michelle Mills Strout, Alan LaMielle, Larry Carter, Jeanne Ferrante, Barbara Kreaseck, and Catherine Olschanowsky. “An approach for code generation in the Sparse Polyhedral Framework”. In: *Parallel Computing* 53 (2016), pp. 32–57. ISSN: 0167-8191. DOI: https://doi.org/10.1016/j.parco.2016.02.004. URL: https://www.sciencedirect.com/science/article/pii/S0167819116000557.

[131] Aravind Sukumaran-Rajam. “Beyond the Realm of the Polyhedral Model: Combining Speculative Program Parallelization with Polyhedral Compilation”. Theses. Université de Strasbourg, Nov. 2015. URL: https://hal.inria.fr/tel-01251748.

[132] Aravind Sukumaran-Rajam and Philippe Clauss. “The Polyhedral Model of Nonlinear Loops”. In: *ACM Trans. Archit. Code Optim.* 12.4 (Dec. 2015), 48:1–48:27. ISSN: 1544-3566.

[133] Kirshanthan Sundararajah and Milind Kulkarni. *Scheduling Transformations and Dependence Tests for Recursive Programs*. Nov. 2018.

[134] Kirshanthan Sundararajah and Milind Kulkarni. “Composable, Sound Transformations of Nested Recursion and Loops”. In: PLDI 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 902–917. ISBN: 9781450367127. DOI: 10.1145/3314221.3314592. URL: https://doi.org/10.1145/3314221.3314592.

[135] Kirshanthan Sundararajah, Laith Sakka, and Milind Kulkarni. “Locality Transformations for Nested Recursive Iteration Spaces”. In: *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’17. Xi’an, China: ACM, 2017, pp. 281–295. ISBN: 978-1-4503-4465-4. DOI: 10.1145/3037697.3037720. URL: http://doi.acm.org/10.1145/3037697.3037720.

[136] Peiyi Tang. “Complete Inlining of Recursive Calls: Beyond Tail-Recursion Elimination”. In: *Proceedings of the 44th Annual Southeast Regional Conference*. ACM-SE 44. Melbourne, Florida: Association for Computing Machinery, 2006, pp. 579–584. ISBN: 1595933158. DOI: 10.1145/1185448.1185574. URL: https://doi.org/10.1145/1185448.1185574.

[137] *The LLVM Compiler Infrastructure*. http://www.llvm.org.

[138] Konrad Trifunovic. "Efficient search-based strategies for polyhedral compilation : algorithms and experience in a production compiler. (Stratégies exploratoires efficaces pour la compilation polyédrique : algorithmes et expérience dans un compilateur de production)". PhD thesis. University of Paris-Sud, Orsay, France, 2011. URL: https://tel.archives-ouvertes.fr/tel-00661334.

[139] N. Vasilache, A. Cohen, and L. Pouchet. "Automatic Correction of Loop Transformations". In: *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*. 2007, pp. 292–304. DOI: 10.1109/PACT.2007.4336220.

[140] Nicolas Vasilache, Cedric Bastoul, Albert Cohen, and Sylvain Girbal. "Violated Dependence Analysis". In: *Proceedings of the 20th Annual International Conference on Supercomputing*. ICS '06. Cairns, Queensland, Australia: Association for Computing Machinery, 2006, pp. 335–344. ISBN: 1595932828. DOI: 10.1145/1183401.1183448. URL: https://doi.org/10.1145/1183401.1183448.

[141] Anand Venkat, Mahdi Soltan Mohammadi, Jongsoo Park, Hongbo Rong, Rajkishore Barik, Michelle Mills Strout, and Mary Hall. "Automating Wavefront Parallelization for Sparse Matrix Computations". In: *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2016, pp. 480–491. DOI: 10.1109/SC.2016.40.

[142] 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. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 299–302.

[143] Mitchell Wand. "Continuation-Based Program Transformation Strategies". In: *J. ACM* 27.1 (Jan. 1980), pp. 164–180. ISSN: 0004-5411. DOI: 10.1145/322169.322183. URL: https://doi.org/10.1145/322169.322183.

[144] Yusheng Weijiang, Shruthi Balakrishna, Jianqiao Liu, and Milind Kulkarni. "Tree Dependence Analysis". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '15. Portland, OR, USA: Association for Computing Machinery, 2015, pp. 314–325. ISBN: 9781450334686. DOI: 10.1145/2737924.2737972. URL: https://doi.org/10.1145/2737924.2737972.

[145] Doran K. Wilde. *A Library for Doing Polyhedral Operations*. Tech. rep. 785. IRISA, Dec. 1993.

[146] P.G. Wodehouse, ed. *Thank You, Jeeves*. 1934.

[147] Janet Wu, J. Saltz, S. Hiranandani, and H. Berryman. "Runtime Compilation Methods for Multicomputers". In: *ICPP*. 1991.

[148] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. "Performance Evaluation of Intel® Transactional Synchronization Extensions for High-Performance Computing". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '13. Denver, Colorado: Association for Computing Machinery, 2013. ISBN: 9781450323789. DOI: 10.1145/2503210.2503232. URL: https://doi.org/10.1145/2503210.2503232.

Annexe A

Résumé en Français

A.1 Introduction et Contexte

Du développement du code à l'exécution finale sur une plate-forme matérielle, un logiciel passe par de nombreuses phases de transformation, ce qui rend les codes exécutables finaux significativement différents du code source initial. Le but final est évidemment de générer un exécutable sémantiquement équivalent au code source d'entrée, mais dont le comportement d'exécution est satisfaisant en ce qui concerne le temps d'exécution, la taille du code, la consommation d'énergie ou la sécurité. Les compilateurs appliquent de nombreuses passes d'optimisation sur le code source d'entrée qui modifient souvent, voire suppriment des instructions, des structures de contrôle ou des structures de données. De telles transformations de code peuvent s'appliquer du niveau instruction jusqu'à la structure de code globale.

Certains compilateurs et optimiseurs avancés peuvent paralléliser automatiquement des codes séquentiels en détectant des régions parallèles dans ces codes et en appliquant automatiquement sur ceux-ci les transformations correspondantes.

A.1.1 Optimiseurs polyédriques

D'une part, un bon nombre de ces optimiseurs sont basés sur le modèle polyédrique (par exemple Polly [52], Pluto [21], etc.). Le modèle polyédrique est un cadriciel mathématique au niveau source (statique) qui apporte une abstraction et une représentation substantielles pour les programmes, en particulier les nids de boucles affines accédant à des tableaux multidimensionnels via des références de tableaux affines, c'est-à-dire des programmes avec des parties de contrôle statiques (SCoP). Ce cadriciel fournit de puissantes analyses et des transformations d'optimisation et de parallélisation agressives et automatiques de boucles (par exemple, pavage de boucles, torsion de boucles, l'échange de boucles, etc.). Dans le domaine de la compilation de programmes et de l'optimisation des codes impératifs, les boucles sont des cibles importantes d'optimisation parce qu'elles sont utilisées couramment dans les programmes et généralement responsables de grandes parties intensives en calcul. Pourtant, de nombreux programmes ne profitent toujours pas des optimisations polyédriques en raison soit des idiosyncrasies superficielles des langages (par exemple, les boucles de type "while"), soit des différences de structure radicales (par exemple les fonctions récursives). Parfois, il s'avère que les

boucles, qui n'ont pas de structure affine au moment de la compilation et ne rentrent pas dans le modèle polyédrique, peuvent en fait présenter un comportement conforme au modèle polyédrique au moment de l'exécution pour au moins de grandes parties de l'exécution de programme. Par conséquent, il peut encore y avoir des opportunités d'optimisation cachées au moment de la compilation qui peuvent être découvertes et saisies dès que le comportement d'exécution est découvert.

A.1.2 Systèmes de spéculation au niveau thread

D'autre part, il existe la technique de spéculation au niveau thread [111, 125, 126] qui exécute de manière spéculative des régions parallèles du code avant de connaître toutes les valeurs d'entrée et les dépendances. Simultanément, le code séquentiel est exécuté en parallèle dans un thread séparé. Au moment de l'exécution, les accès à la mémoire sont suivis et la vérification est effectuée, et dans le cas où une spéculation invalide est prouvée (par exemple, violation de dépendance), un mécanisme de récupération est exécuté. La récupération implique l'abandon des threads spéculatifs non valides et la relance du code séquentiel à partir du dernier point cohérent. Cette technique a été principalement dédiée à l'optimisation des structures en boucle. Cependant, son succès n'est pas garanti et le gain de performance offert peut ne pas être excellent en raison d'une charge déséquilibrée, de spéculations invalides, des optimisations de boucle simples qu'il offre contrairement à celles des optimiseurs polyédriques et des communications inter-thread dans le processus de détection des violations de dépendance.

Inspecteur-exécuteur

Le mécanisme inspecteur-exécuteur est une technique bien connue dans le domaine de la compilation et de l'optimisation des programmes utilisé pour guider et valider les transformations de code. Il a été vu et utilisé dans des travaux antérieurs exclusivement dédiés à la parallélisation spéculative de boucles [110, 36]. Cette technique implique deux processus principaux : (1) Inspecteur et (2) Exécuteur. L'inspecteur et l'exécuteur peuvent être automatiquement construits à partir des boucles d'origine à optimiser. Chaque fois que cela est possible, la boucle d'origine est divisée en deux boucles où :

- La première est exécutée par le processus inspecteur : il s'agit d'une version allégée de la boucle qui permet de surveiller l'exécution et d'effectuer des calculs légers pour guider les transformations de la boucle et les vérifier. Il calcule les adresses mémoire qui sont censées être accédées, afin de pouvoir calculer avec précision les dépendances de données, obtenir les bornes supérieures de la boucle et d'autres informations liées à la boucle. Ces informations sont nécessaires pour garantir une transformation valide.

- La deuxième boucle est exécutée par le processus exécuteur : c'est une boucle qui effectue en fait les accès mémoire et les calculs existants. Cette boucle est optimisée et parallélisée de manière fiable en fonction des informations collectées par l'inspecteur.

Optimiseur de boucles Polyédrique spéculatif automatique (APOLLO)

À cet égard, Apollo, un optimiseur polyédrique spéculatif [132, 84, 131, 83], combine les deux approches présentées ci-dessus. Il a été mis en œuvre pour capturer les comportements polyédriques transitoires de boucles statiquement non affines à l'aide du profilage dynamique et exploiter les puissants outils polyédriques pour les optimiser de manière agressive au moment de l'exécution.

Mais qu'en est-il des structures non-boucles comme les fonctions récursives, qui sont des fonctions qui participent à un cycle d'appels, qui ne peuvent pas bénéficier de ces techniques d'optimisation et de parallélisation automatiques ? Existe-t-il des alternatives dans ces cas qui offrent un gain de performance satisfaisant ?

A.2 État de l'Art

Dans une exécution de programme, les fonctions récursives, comme les boucles, sont également parmi les structures coûteuses en temps les plus remarquables, responsables d'une grande partie de l'ensemble de l'exécution. En général, les fonctions récursives implémentent des algorithmes complexes, notamment pour le calcul haute performance, l'analyse et le traitement d'énormes structures de données, par ex. les matrices, les graphiques et les arbres. Une approche récursive est généralement adoptée lors de la résolution de tout problème dont la solution repose sur des instances plus petites et plus simples, selon une stratégie "diviser pour régner". Cela facilite également l'expression de calculs génériques pour des paramètres tels que des profondeurs de recherche ou des dimensions de problème. Bien que les fonctions récursives soient des candidats intéressants pour l'optimisation et la parallélisation, elles ne bénéficient pas, contrairement aux boucles, de techniques avancées de parallélisation et d'optimisation automatiques puissantes. Dans la littérature traitant de l'optimisation de la récursivité, les fonctions récursives peuvent être soit gérées directement "telles quelles" [53, 85, 117, 34, 94, 91, 122, 54] ou transformées en boucles d'abord [75, 85].

Manipulées Directement Telles Quelles

Les fonctions récursives sont principalement parallélisées selon une stratégie parallélisation de tâche de sorte que plusieurs appels sont exécutés simultanément si les dépendances de données entre les appels le permettent. Comme travaux récents, nous mentionnons les travaux de Gupta et al., proposant DECAF [54], qui est une technique pour optimiser les programmes parallèles de tâches récursives en réduisant les coûts de création de tâches et de terminaison. Il y a aussi Adriadne [85] qui est un compilateur qui extrait le parallélisme basé sur les directives des appels des fonctions récursives. Ce travail est de portée plus large que les travaux présentés dans cette section, car il extrait trois formes de parallélisme et une transformation pour chacune d'elles : (1) réduction parallèle : élimination de la récursivité et répartition de la charge de travail en tâches indépendantes, (2) parallélisation thread-safe des fonctions récursives contenant des appels récursifs indépendants et (3) élimination de la récursivité : conversion de la récursivité en itération qui fait qu'Adriadne rentre également dans la deuxième catégorie présentée dans la section suivante. Par ailleurs, il existe des techniques récentes qui permettent

la modélisation polyédrique des invocations récursives. PolyRec [135, 133, 134] optimise les programmes récursifs imbriqués par des transformations d'ordonnancement polyédriques. Pour permettre de telles optimisations, PolyRec représente les instances de fonctions récursives et leurs dépendances sous forme de polyèdres, et applique des transformations d'ordonnancement. Néanmoins, leur approche est exclusivement consacrée à des formes particulières de récursivité telles que les invocations récursives sont imbriquées et les données sont organisées en deux arbres, les arbres intérieur et extérieur.

Transformées d'abord en Boucles

Il est bien connu qu'il est toujours possible de remplacer une fonction récursive par une boucle imbriquée équivalente et vice versa [5]. En général, à chaque appel de fonction, des informations sur la fonction activée / invoquée (par exemple, paramètres de fonction, informations de variables locales, adresse de retour, etc.) sont empilées sur la pile d'appels, c'est aussi le cas pour les fonctions récursives. En conséquence, une récursivité peut être remplacée par une boucle et une structure de données qui imite la pile de programmes. Dans des cas particuliers, comme le cas d'une fin de récurrence dans laquelle l'appel récursif est la dernière instruction de la fonction récursive, les données sauvegardées dans la pile ne sont pas nécessaires car il n'y a plus de calculs à faire après le retour des fonctions récursives. Cela a inspiré les compilateurs avancés comme Clang [30] et GCC [48] pour éliminer les appels récursifs de queue et les transformer en boucles. De plus, de nombreuses études ont été menées sur l'élimination de la récursivité [75, 85]. Cependant, en général, la transformation statique de récursivité en boucles génère des boucles dont les structures sont complexes et non affines et, par conséquent, ne peuvent pas bénéficier d'autres optimisations avancées.

A.3 Problématique et motivation

Les fonctions récursives, comme les boucles, sont des structures coûteuses qui doivent être ciblées et optimisées de manière agressive. Bien qu'il existe de nombreuses études sur l'optimisation des structures récursives et de nombreux optimiseurs de récursivité automatiques disponibles, ils sont tous activés et appliqués au moment de la compilation sur la base d'une analyse statique. Ils conduisent évidemment à des exécutions plus efficaces ; mais ces optimisations sont relativement limitées par rapport aux options d'optimisation qui existent réellement pour les structures de boucle. Les techniques existantes ne capturent pas le moment où les codes récursifs peuvent être réécrits sous forme de boucles affines qui sont les candidats les plus appropriés pour une localisation de données efficace et de puissantes transformations polyédriques d'optimisation et de parallélisation. En conséquence, la motivation de cette thèse est d'aller au-delà des limites classiques fixées pour les techniques de suppression de récursivité et de saisir cette chance et de découvrir les opportunités d'optimisation que le modèle polyédrique peut offrir aux fonctions récursives dont le comportement à l'exécution est affine. Notre travail n'est pas uniquement dédié aux fonctions récursives avec une structure particulière, il est plutôt dédié aux fonctions récursives à comportement particulier, polyédrique. En conséquence, non seulement nous nous appuyons sur une analyse statique, mais également sur une analyse dynamique pour guider notre transformation de récursivité en itération.

C'est pour cela, nous présentons comme contribution principale de notre thèse le cadriciel Rec2Poly.

A.4 Rec2Poly

Le cadriciel Rec2Poly est un optimiseur dynamique et spéculatif des fonctions récursives basé sur le compilateur LLVM / Clang [137]. Il découvre, grâce à une technique de profilage hors ligne, un comportement d'exécution affine de codes récursifs qui manipulent des structures de données coûteuses, par exemple des tableaux. En cas de succès, il construit un code sémantiquement équivalent où tous les flux d'exécution liés aux fonctions récursives sont remplacés par des boucles affines et permet d'appliquer des optimisations et des parallélisations de boucles polyédriques puissantes. Rec2poly génère également un mécanisme de vérification à l'exécution suivant un schéma inspecteur-exécuteur, pour garantir la validité du code itératif généré pour les différentes données d'entrée. En conséquence, pour y parvenir, Rec2Poly est composé de trois phases principales :

1. Analyse statique du code et phase de préparation
2. Phase de profilage hors ligne
3. Phase de génération de code Inspecteur-Exécuteur

Les analyses, transformations et optimisations dans ces phases sont principalement mises en œuvre au fur et à mesure de passes LLVM [80], en traitant la représentation intermédiaire LLVM (IR) du code [78].

Les phases de Rec2Poly et leurs principaux constituants sont illustrées à la figure A.1.

Au début, Rec2Poly effectue une analyse statique. Il analyse en profondeur le code récursif cible afin d'identifier les fonctions récursives. Il identifie également les fonctions qui peuvent être appelées par des fonctions récursives, ou qui peuvent invoquer eux même, directement ou indirectement, des fonctions récursives. Nous appelons ces fonctions identifiées, des *fonctions impactantes*. Ensuite, Rec2Poly construit la tranche arrière statique (*Backward Static Slice* (*BSS*)) correspondant à chaque instruction de stockage mémoire (écriture) dans les fonctions impactantes, et collecte les identifiants de tous les blocs de base qui contiennent au moins une instruction impliquée dans le calcul de l'adresse de la mémoire cible ou de la valeur stockée, c'est-à-dire dans le BSS. Nous appelons ces blocs de base, des blocs de base impactants. Ensuite, Rec2Poly prépare le code pour les phases suivantes et effectue la globalisation des variables locales. Ceci est réalisé en insérant, au début d'une fonction impactante en LLVM IR, un compteur d'invocation et en transformant chaque structure de données locale ou une variable scalaire en une structure de données globale indexée à l'aide de ce compteur. Cela permet de garder une trace des variables initialement locales dans les fonctions qui nous intéressent et de résoudre les dépendances entre leurs différentes invocations. Ensuite, ce code globalisé étendu est ré-analysé pour obtenir les informations nécessaires liées à la récursivité en tenant compte des modifications appliquées au code récursif d'origine.

En utilisant les informations d'analyse ainsi collectées et la version étendue du code récursif cible, Rec2Poly commence par la deuxième phase, la phase de profilage lors de

l'exécution, en générant une version instrumentée du code cible. Le code instrumenté est obtenu en ajoutant à chaque fonction impactante, les instructions pour générer la trace de sortie. La trace générée décrit le contrôle et le comportement mémoire à l'exécution du programme ; elle est composée des identifiants de blocs de base impactant, des valeurs de compteurs d'invocation et des adresses mémoire référencées, dans les blocs de base impactants, via des instructions de chargement (lecture mémoire) et de stockage (écriture). Une fois le code instrumenté exécuté, la trace générée est donnée comme entrée à une version étendue de l'outil logiciel de reconnaissance de boucle imbriquée NLR [65]. NLR génère, dans la mesure du possible, une représentation de l'ensemble de la trace en boucles, voire en boucles affines calculant des expressions affines.

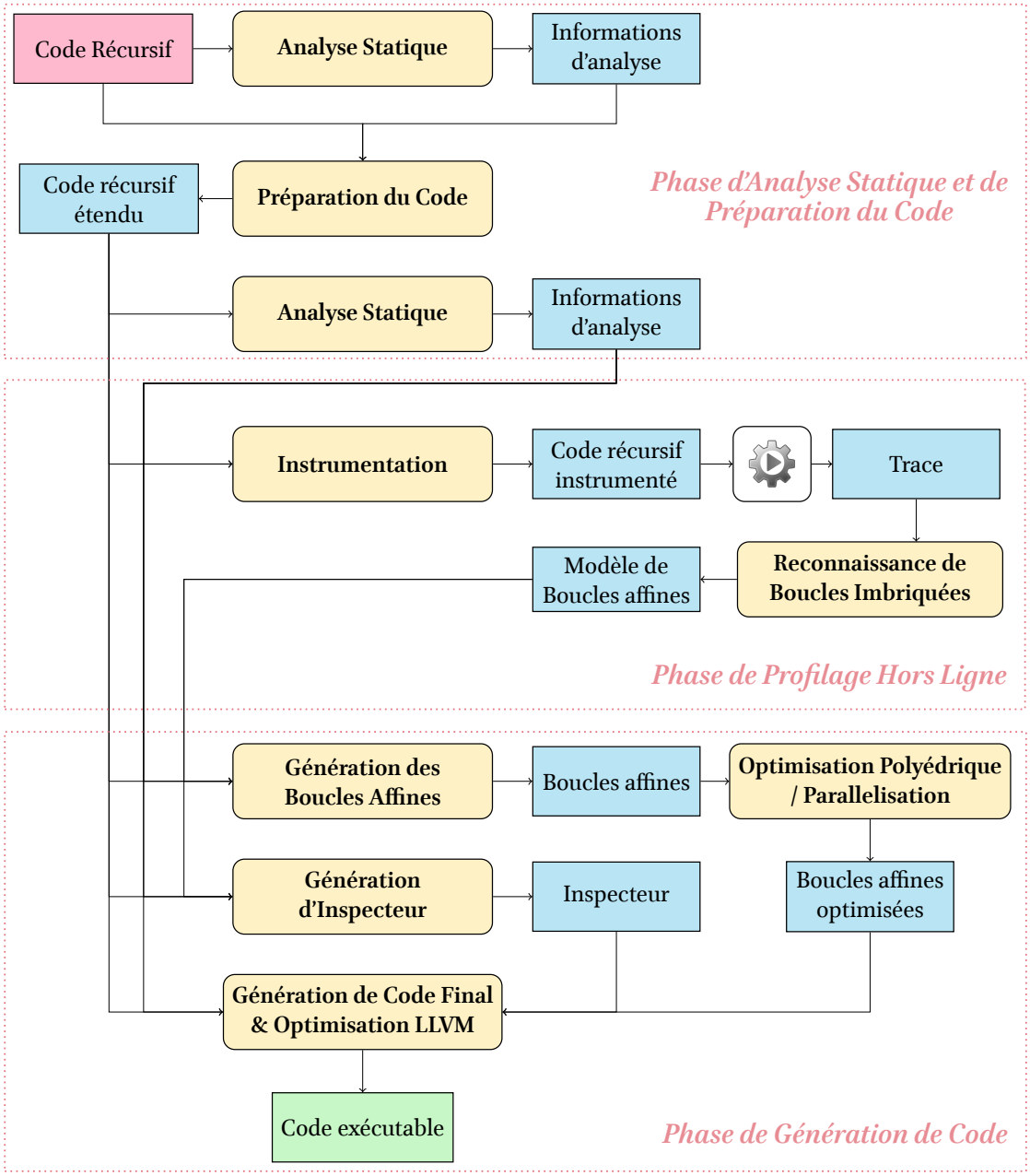


FIGURE A.1 – Rec2Poly

Les informations d'analyse de code et le résultat de profilage obtenus, c'est-à-dire le modèle de boucle ainsi généré, peuvent ensuite être utilisés par Rec2Poly pour effectuer la phase de génération et d'optimisation de code, et construire une version optimisée compte tenu du code récursif étendu. Pour cela, Rec2Poly génère un code itératif composé de séquences de nids de boucles optimisables remplaçant les fonctions impactantes existant dans le code récursif d'origine. Étant donné que les boucles de remplacement peuvent être des boucles entièrement affines, elles peuvent bénéficier de transformations d'optimisation et de parallélisation polyédriques. Pour cela, Rec2Poly utilise le compilateur polyédrique Pluto [101]. Sinon, une analyse de dépendance dédiée peut encore être effectuée et les boucles générées peuvent être encore puissamment parallélisées en utilisant OpenMP. Le modèle de boucle affine, obtenu à partir du profilage hors ligne basé sur une exécution du code récursif cible, peut également être utile pour les optimisations et exécutions de code ultérieures, même pour différentes entrées de même taille, c'est-à-dire la même taille du problème.

Cependant, étant donné que les exécutions seront spéculatives, leur validité doit toujours être garantie au moment de l'exécution. Le programme optimisé généré est vérifiable au moment de l'exécution. C'est le code d'inspecteur parallèle rapide généré par Rec2Poly qui collecte les informations manquantes et vérifie, au moment de l'exécution, que les boucles affines générées s'exécutant en parallèle se comportent toujours en conformité avec le code récursif d'origine.

Enfin, le code final est généré sur la base du schéma inspecteur-exécuteur composé du : code inspecteur, code exécuteur contenant les boucles optimisées et le code récursif d'origine. Ce code peut être en outre optimisé à l'aide des optimisations LLVM / Clang disponibles.

Lors de l'exécution du programme généré, le code récursif d'origine est exécuté simultanément avec l'inspecteur en utilisant les threads POSIX. En effet, au cas où une erreur de spécification serait détectée au moment de l'exécution, le code entier de l'inspecteur-exécuteur doit être annulé et un état correct de l'exécution du programme doit être récupéré avec la plus faible surcharge de temps. Le thread exécutant le code récursif poursuit son exécution tant que l'inspecteur est toujours dans le processus de vérification ; lorsqu'une exécution correcte est assurée, le thread est abandonné et l'exécuteur lance ses boucles optimisées parallèles.

Notre approche, Rec2Poly, est considérée comme une réécriture de code dynamique. Elle a été inspirée par l'approche Apollo, mais au lieu de gérer statiquement les boucles non affines, elle gère des structures sans boucles, des fonctions récursives. L'originalité du cadriciel Rec2Poly est double :

1. il recherche un comportement d'exécution conforme au modèle polyédrique dans les fonctions récursives, et

2. il utilise un schéma inspecteur-exécuteur pour vérifier non seulement les modèles d'accès à la mémoire comme d'habitude pour cette stratégie, mais aussi le flot de contrôle par rapport à celui des boucles affines.

Nous avons précédemment introduit la preuve de concept des phases d'analyse, de profilage et de transformation de récursivité en boucles optimisées de Rec2Poly dans notre travail [66]. Ensuite, dans un travail ultérieur [67], nous avons présenté l'extension de ces phases et l'introduction de la fonction de vérification basée sur le paradigme

inspecteur-exécuteur. Dans cette thèse, nous revisitons ces études, discutons et approfondissons les parties liées aux inspecteurs. Les principales phases de Rec2Poly sont présentées et discutées en détail dans les sections suivantes.

A.5 Phase d'Analyse Statique et de Préparation du Code

A.5.1 Analyse statique

Tout d'abord, Rec2Poly, notre outil basé sur LLVM-Clang, prend en entrée un code source récursif cible et le transforme en sa représentation intermédiaire (IR) qui sera analysée et transformée dans les étapes suivantes. Nous n'activons pas la passe LLVM d'élimination des appels terminaux qui transforme les appels récursifs terminaux en boucles et ceci pour deux raisons : (1) la façon dont la fonction récursive cible est transformée peut ne pas donner lieu à une boucle affine et (2) s'il y a plusieurs appels récursifs imbriqués dans le code cible, un seul appel terminal peut être éliminé. Rec2Poly vérifie si le programme comporte des fonctions récursives et, si tel est le cas, identifie celles-ci et les fonctions qui y participent. Afin de détecter les fonctions récursives, il utilise le graphe d'appel extrait de la représentation intermédiaire du programme.

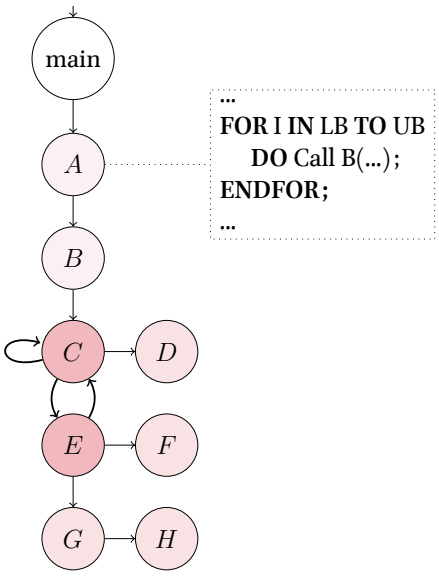


FIGURE A.2 – Exemple de graphe d'appel d'un programme récursif arbitraire

La figure A.2 montre un exemple de graphe d'appel d'un programme composé de neuf fonctions : main, A, B, C, D, E, F, G et H où : la fonction main appelle A qui appelle B qui appelle C ; C appelle elle-même, E et D ; E rappelle C et appelle F et G ; G appelle H. Dans cet exemple, notez que la fonction C montre une récursivité directe avec elle-même et une récursivité indirecte via la fonction E. À partir du graphe d'appel, Rec2Poly recherche des composants fortement connectés (SCC), qui sont des sous-graphes où chaque nœud est accessible à partir de tous les autres nœuds. Dans ce contexte, un cycle dans un SCC signifie qu'une récursivité se produit parmi les fonctions associées aux nœuds impliqués dans ce cycle. Si le cycle est constitué d'un seul nœud, c'est-à-dire une boucle, alors il s'agit

d'une récursivité directe. Sinon, c'est indirect. Pour l'exemple de la figure A.2, il y a un SCC avec une boucle sur C montrant une récursivité directe, et un cycle de C vers E et E vers C, montrant une récursivité indirecte entre C et E.

Reconnaissance d'accessibilité de la récursivité

Nous nous intéressons au suivi des blocs de base impactants, qu'ils soient exécutés directement ou indirectement par les fonctions récursives. Pour cette raison, en plus des fonctions récursives elles-mêmes, notre système détermine également leur accessibilité dans le programme. L'accessibilité désigne toutes les fonctions qui peuvent être atteintes par une séquence d'appels initiée par les fonctions récursives elles-mêmes. Sur la figure 2, l'accessibilité des fonctions récursives C et E comprend : D (directement appelée par C), F et G (directement appelée par E) et H (indirectement appelée par E à G).

Reconnaissance de source de récursivité

Non seulement nous suivons les fonctions constituant une récursivité et leur accessibilité dans un programme, mais aussi la source de ces fonctions. La fonction source initiale peut être une fonction invoquant, à partir d'une boucle, une fonction récursive, directement ou indirectement, à travers une chaîne de fonctions invoquées. De plus, en cas d'invocation indirecte de la fonction récursive à partir d'une boucle, toutes les autres fonctions participant à la séquence d'appels à partir de la fonction source initiale jusqu'à la fonction récursive sont considérées comme faisant partie de cette source. L'analyse des fonctions sources est nécessaire car sinon, la phase de profilage serait incomplète. Un comportement de boucle détecté par la suite et associé à la récursivité elle-même serait incorrect lorsque la fonction récursive est invoquée à partir d'une boucle. De plus, cela aide à comprendre comment une fonction récursive se comporte par rapport à ses invocations itératives, et évidemment à construire des boucles affines qui sont équivalentes à toute cette partie du programme original.

Par exemple, l'ensemble des fonctions sources de la récursivité de la figure A.2 comprend A et B car A appelle B à partir du corps d'une boucle for, et B à son tour appelle la fonction récursive C. La boucle en A ne peut pas bénéficier d'optimisations polyédriques efficaces de boucle, du fait de l'existence de l'appel récursif imbriqué. S'il est possible de la remplacer par des boucles affines équivalentes, la boucle pourra éventuellement profiter d'optimisations sophistiquées.

Identification des blocs de base impactants

Ces blocs de base sont identifiés de la manière suivante. Dans la représentation intermédiaire LLVM du programme, notre système marque les instructions de stockage significatives en mémoire. Ensuite, pour chaque instruction de stockage de ce type, il marque également chaque instruction qui la mène et y contribue, c'est-à-dire sa tranche arrière statique (BSS). Une BSS est l'ensemble des instructions existant dans le code d'un programme qui peuvent affecter une certaine valeur, c'est-à-dire, dans notre cas, une valeur stockée et une adresse mémoire liées à la fonction impactante.

Analyse du comportement mémoire intra-fonction et inter-fonctions

En plus de l'identification d'un comportement en boucle de la récursivité ciblée, la séquence d'adresses mémoire touchées par chaque instruction mémoire, à l'intérieur des blocs de base impactants, doit être modélisée avec succès par des fonctions affines des indices de boucles les entourant. Cependant, parmi différentes instances d'exécution du même code récursif cible, avec exactement les mêmes données d'entrée et la même plateforme matérielle, les adresses mémoire touchées ne sont évidemment pas les mêmes, car les structures de données ne sont pas toujours allouées aux mêmes adresses mémoire. Néanmoins, le comportement de la mémoire par rapport aux adresses de base des structures de données peut encore être identique parmi les instances d'exécution. De plus, nous nous intéressons aux cas où le comportement mémoire relatif peut être modélisé par des fonctions affines d'indices de boucles les entourant. Ainsi, les décalages de mémoire relatifs aux adresses de base sont collectés à partir de l'instrumentation. Lors de la manipulation de structures de données locales à des fonctions, la phase d'analyse nécessite deux étapes :

- Analyse intra-fonction : chaque accès mémoire est associé à son adresse de base correspondante visible dans le périmètre de la fonction courante, c'est-à-dire que les paramètres de la fonction sont le point d'analyse le plus éloigné de la fonction.
- Analyse inter-fonctions : si les structures de données accédées sont des paramètres de fonction, l'analyse intra-fonction ne suffit pas. L'analyse de la mémoire se prolonge plus loin en dehors de la fonction pour suivre les arguments fournis à la fonction. L'analyse inter-fonctions associe chaque accès à son adresse de base réelle dans le programme.

Par contre, lors de la gestion de structures de données globales, les adresses mémoire accédées peuvent être directement associées à leurs adresses de base.

A.5.2 Préparation du Code

Conservation du Code d'Origine

Avant que Rec2Poly n'applique des modifications au code, il est préférable de conserver une copie originale intacte du code, donc Rec2Poly clone toutes les fonctions à l'exception de la fonction principale. Toutes les allocations mémoire et les variables doivent être créées dans des fonctions qui ne sont pas des fonctions main du code d'origine. La fonction main lance alors la partie de code des clones au lieu du code d'origine. Ensuite, cette partie de code mort sera utilisée, dans la phase de génération de code, pour créer le thread de sauvegarde invoquant le code récursif d'origine.

En conséquence, toute analyse effectuée et autres modifications appliquées au code sont considérées comme étant appliquées au code récursif cloné, sauf indication contraire explicite. Par conséquent, par défaut, une fonction impactante fait référence à son clone.

Insertion de compteur d'appel de fonction et globalisation des variables locales

Le premier objectif de Rec2Poly est de détecter un comportement affine des fonctions impactantes, vis-à-vis de leur flot de contrôle, ainsi que de leurs accès mémoire : pour

chaque instruction mémoire, la séquence d'adresses mémoire touchées doit potentiellement être représentée comme des expressions affines des indices de boucles les entourant. Cependant, à chaque invocation d'une fonction impactante, ses structures de données locales sont évidemment allouées sur la pile d'exécution. Ainsi, les accès à ces structures locales ne peuvent jamais présenter d'accès mémoire affine à travers toutes les invocations de la fonction. De plus, dans les boucles affines censées remplacer les fonctions impactantes, ces structures de données doivent évidemment être référencées encore. C'est pourquoi Rec2Poly ajoute un compteur d'invocation à certaines fonctions impactantes et transforme toutes les structures de données importantes qui sont locales à ces fonctions en tableaux globaux, qui sont indexés par le compteur d'invocation de fonction. De cette manière, les références à ces données globalisées peuvent présenter des comportements affines, que les fonctions associées soient appelées à la suite d'un flot de contrôle affine. Ceci n'est effectué que pour les fonctions impactantes avec des accès mémoire initiaux, car il peut y avoir des fonctions impactantes qui n'effectuent qu'une décomposition de problème, par exemple avec des calculs légers et des variables locales qui n'ont pas besoin d'être globalisées.

A.6 Phase de Profilage Hors Ligne

Rec2Poly doit détecter un comportement de boucle affine des fonctions récursives qui peut permettre de transformer les récursivités en boucles optimisables. Le comportement des récursivités à découvrir correspond à la fois aux comportements mémoire et contrôle des fonctions impactantes impliquées. Dans l'optimiseur de boucle spéculatif, Apollo, notre inspiration dans un domaine différent, l'analyse du comportement mémoire des boucles concernées suffit à guider les transformations de code. Cependant, Apollo a l'intention de transformer les boucles en d'autres boucles affines. Ainsi, il convertit une structure de contrôle en une structure similaire. D'autre part, Rec2Poly transforme les appels récursifs en des structures de contrôle totalement différentes, des boucles. En conséquence, une telle transformation nécessite de comprendre le comportement du contrôle récursif en plus du comportement mémoire.

Pour cela, Rec2Poly effectue une analyse d'exécution dédiée via une technique de profilage hors ligne. Cette technique se compose des deux étapes suivantes :

A.6.1 Instrumentation

Rec2Poly génère, à partir de la version globalisée du code récursif cible, une version instrumentée du code pour produire la trace d'exécution de contrôle et d'accès mémoire des fonctions impactantes.

Le comportement du contrôle est décrit par les blocs de base impactants exécutés au moment de l'exécution. Rec2Poly doit générer les identifiants de ces blocs de base. D'autre part, le comportement de la mémoire est décrit par les adresses mémoire accédées relatives (offsets). Les offsets sont calculés pour toutes les instructions mémoire existantes en fonction de leurs adresses de base associées obtenues à partir de l'analyse du comportement mémoire. Une instruction de différence de pointeur est insérée, dans la représentation LLVM IR, pour chaque instruction de chargement ou de stockage, en soustrayant l'adresse de base correspondante de l'adresse mémoire réelle touchée.

Ensuite, Rec2Poly supprime toutes les instructions d'affichage existantes de LLVM IR et y ajoute les instructions nécessaires à l'écriture de la trace de sortie. Pour chaque bloc de base impactant, une instruction est ajoutée pour générer l'identifiant de bloc de base et les offsets de toutes les adresses mémoire qui y sont accédées lors de l'exécution du programme.

Cependant, bien que l'incrémentation des compteurs d'appels de fonctions, en tant que variables globales, nécessite des accès à la mémoire, Rec2Poly les gère différemment. Au lieu d'instrumenter leur offset mémoire (évidemment zéro), il ajoute des instructions pour afficher leurs valeurs réelles au moment de l'exécution dans les blocs de base d'entrée des fonctions impactantes. Aussi, Rec2Poly instrumente les valeurs des variables d'induction des boucles existant dans le code. Ceci est important pour la phase de génération de code.

A.6.2 Reconnaissance de boucles imbriquées (NLR)

Lorsque la version instrumentée est exécutée, elle produit une trace d'exécution qui est composée de tuples. Chaque tuple est principalement composé de l'ID de bloc de base impactant, et les offsets relatifs des adresses mémoire accédées par toutes les instructions mémoire du bloc de base courant.

Il est possible d'avoir des valeurs du compteur d'invocation de fonction ou des indices de boucle.

Après avoir été générée en exécutant le programme récursif instrumenté, la trace est analysée par une version étendue de NLR.

L'algorithme NLR prend en entrée une trace de l'exécution d'un programme et construit une séquence de nids de boucles qui produisent la même trace d'origine lors de leur exécution. Les applications de cet algorithme comprennent :

1. la modélisation du comportement du programme pour toute quantité mesurée telle que les accès à la mémoire

2. la compression de trace d'exécution, et

3. la prédiction de valeur, c'est-à-dire l'extrapolation de boucles en construction (lors de la lecture de l'entrée) pour prédire les valeurs à venir.

Dans notre outil, non seulement nous utilisons NLR pour modéliser les accès mémoire, ce qui est l'un de ses objectifs d'origine, mais aussi pour modéliser des séquences d'identifiants de blocs de base, ce qui est plus singulier. Compte tenu de notre trace d'un programme récursif cible, si NLR construit des nids de boucles affines comprenant les ID de blocs de base intéressants et les adresses mémoire interpolées par les indices de boucle construits, alors la génération de boucles affines équivalentes peut être effectuée.

Deux exemples de sorties NLR sont représentés sur les figures A.3 et A.4. Les boucles générées montrent la façon dont les blocs de base (BB1, BB2, BB3, BB4) à l'intérieur des fonctions (F1, F2) sont appelés en suivant un comportement de boucle affine, et comment la mémoire est référencée par des adresses relatives qui peuvent être modélisées comme des fonctions affines d'indices de boucle. Notez que sur la figure A.4, NLR utilise l'une de ses fonctionnalités les plus avancées qui détecte que la modélisation affine d'une trace peut dépendre de certains paramètres inconnus. NLR découvre ces valeurs et présente

```
for i0 = 0 to 99
 val F1::BB1
 for i1 = 0 to 9
   val F2::BB1
   , 0
   for i2 = 0 to 9
     val F2::BB2
     , 1*i0
     , 4*i0 + 2*i1 + 1*i2
     , 4*i0 + 1*i1 + 1*i2
   val F2:BB3
 val F2::BB4
```

FIGURE A.3 – Modèle de boucles affines NLR du contrôle et du comportement mémoire

```
for i0 = 0 to 99
 val F1::BB1
 for i1 = 0 to 9
   val F2::BB1
   , 0
   for i2 = 0 to 9
     val F2::BB2
     , 1*i0
     , [10:3,5,...,1][i0] + 2*i1 + 1*i2
     , [10:7,1,...,6][i0] + 1*i1 + 1*i2
   val F2:BB3
 val F2::BB4
```

FIGURE A.4 – Modèle de boucle paramétriquement affines NLR du contrôle et du comportement mémoire

un comportement mémoire qui n'est en fait pas totalement affine, que nous appelons paramétriquement affine de telle sorte que certains coefficients dans les fonctions affines peuvent être des listes de valeurs. Dans l'exemple montré, chaque liste contient 10 valeurs entières qui sont successivement utilisées pour calculer l'adresse mémoire référencée. Par exemple, [10 : 3,5, ..., 1] [i0] signifie que :

$$[10:3,5,...,1][i_0] = \begin{cases} 3 & \text{si } i_0 = 0 \\ 5 & \text{si } i_0 = 1 \\ ... & \\ 1 & \text{si } i_0 = 9 \end{cases}$$

A.7 Phase de Génération de Code

Etant donné que la génération de code est basée sur le profilage hors ligne d'une exécution de programme précédente, l'exactitude du code final généré doit être vérifiée pendant les nouvelles exécutions de programme. Pour cela, Rec2Poly génère le code final basé sur le mécanisme d'un Inspecteur-Exécuteur.

Bien qu'il ait été exclusivement utilisé dans le domaine des codes itératifs, et pour vérifier leurs accès mémoire, comme c'est le cas dans Apollo, Rec2Poly étend ce mécanisme et l'amène au domaine des codes récursifs et la vérification du contrôle.

A.7.1 Génération d'Inspecteur Parallèle Rapide

Le modèle de boucles affines généré par NLR est ensuite utilisé par Rec2Poly pour générer l'inspecteur. Son rôle sera de vérifier que les boucles affines optimisées et parallélisées, qui remplacent le programme récursif, sont toujours correctes dans le contexte d'exécution courant. Il est composé de trois principaux types de composants :

1. Les générateurs de trace, qui sont des versions minimales du programme récursif original, consacrés à la production du même type de traces d'exécution que ce-

lui qui a été généré lors de la phase de profilage, c'est-à-dire des tuples constitués de fonctions et d'identifiants de blocs de base, d'adresses mémoire référencées et les valeurs des compteurs d'appels de fonctions et des variables d'induction des boucles ;

2. Les vérificateurs, dont le rôle est de vérifier si les traces générées sont toujours conformes au modèle de boucles affines NLR ;

3. Un enregistreur de paramètres, dont le rôle est de collecter les valeurs d'entrée de fonction qui sont utilisées par les instructions des blocs de base impactants.

Nous illustrons leurs fonctionnalités et comment Rec2Poly modifie l'IR d'un code récursif donné pour construire son inspecteur approprié.

Générateurs de Traces

Un générateur de traces est composé de clones minimaux légers des fonctions impactantes, c'est-à-dire des fonctions source, récursives et accessibles. Son rôle est de générer une trace représentant le flot de contrôle réel ou la séquence d'adresses mémoire touchées et les valeurs des compteurs d'appels de fonctions et des variables d'induction des boucles. Après le clonage des fonctions impactantes et leurs blocs de base, Rec2Poly supprime les instructions qui impliquent l'accès à la mémoire telles que les stockages et les chargements. Les instructions qui sont fondamentales pour préserver un comportement de contrôle correct de ces fonctions doivent être préservées telles que les branches, les conditions, les instructions et les appels liés aux boucles. Nous supposons pour cette étude que les branches conditionnelles ne dépendent d'aucun accès mémoire. De plus, dans les clones, les fonctions impactantes référencées dans les instructions d'appel doivent être remplacées par leurs propres clones. On s'attend à ce qu'un générateur de trace produise une trace afin que cette dernière puisse être vérifiée par rapport au modèle de boucles affines NLR. Pour cela, des buffers ou des tableaux de mémoire globaux sont ajoutés à l'IR.

L'Inspecteur doit être nettement plus rapide que le programme récursif d'origine, de sorte que le couple final Inspecteur-Exécuteur offre des accélérations significatives. La génération d'une trace complète de tuples complets de valeurs, similaire à la trace générée lors de la phase de profilage, serait trop coûteuse. Ainsi, afin de garantir un processus de génération de trace rapide, un inspecteur, créé par Rec2Poly, est composé de plusieurs générateurs de trace, c'est-à-dire de multiples clones de fonctions impactantes, qui sont exécutés en parallèle chacun par un thread parallèle distinct. Chacun de ces générateurs est responsable de la génération d'une sous-partie de la trace, par exemple, l'un génère l'ensemble des ID de flot de contrôle, et les autres génèrent des séquences d'adresses mémoire touchées, des valeurs d'index de boucle, etc. En conséquence, Rec2poly doit s'attaquer à un problème d'équilibrage de charge entre les threads en décidant du nombre d'accès à la mémoire un seul générateur de trace doit gérer et lesquels. Par défaut, chaque générateur de trace mémoire ou thread de vérification gère un accès à la mémoire par bloc de base, vu que le couple générateur de trace de contrôle-vérificateur gère un seul bloc de base ID par visite.

Vérificateurs

Pour chaque générateur de trace, Rec2Poly crée un vérificateur de trace correspondant basé sur le modèle de boucles affines NLR. Chaque vérificateur est généré comme une nouvelle fonction qui implémente les boucles NLR et les versions minimales de leurs blocs de base inclus. Lors de l'exécution, les vérificateurs de trace sont également lancés dans des threads parallèles.

Enregistreur de paramètre

Certains arguments d'entrée des fonctions impactantes peuvent être des valeurs transmises par la fonction appelante et utilisées directement par les instructions. Ces paramètres doivent être collectés spécifiquement afin d'instancier les boucles de remplacement. Comme les générateurs de trace, un enregistreur de paramètres est constitué d'une version légère minimale de la partie de code impliquant des fonctions impactantes. Il enregistre les valeurs d'entrée de fonction dans un tableau de buffers global à l'entrée de chaque fonction impactante. Les enregistreurs de paramètres sont également exécutés simultanément avec les générateurs de traces et les vérificateurs.

Optimisations Supplémentaires de l'Inspecteur

Dans certains cas, l'inspecteur n'a pas besoin de gérer tous les accès mémoire à l'intérieur d'un bloc de base impactant. Par exemple, dans les cas d'accès à des tableaux, si le même tableau est accédé plusieurs fois via des indices calculés à l'aide de la même variable d'induction, alors un seul de ces accès nécessite d'être manipulé et vérifié par l'inspecteur. En outre, il existe des blocs de base qui se répètent toujours ensemble, par exemple les blocs de base d'une boucle ; un seul doit être tracé et vérifié au moment de l'exécution car le reste peut être pris en compte avec lui. De plus, dans le cas où le code comprend des boucles qui affectent le résultat final mais pas le contrôle ou les accès mémoire du programme, elles peuvent être interrompues dans les générateurs de trace ainsi que leurs équivalences dans les vérificateurs, après au maximum trois itérations ; nous avons besoin de trois itérations maximum pour être conforme à NLR puisque NLR détecte des boucles dans les traces se répétant trois fois ou plus.

La figure A.5 montre le graphe d'appel de l'inspecteur correspondant à l'exemple arbitraire du programme récursif de la figure A.2. Les fonctions impactantes pour lesquelles Rec2Poly crée des clones légers pour construire l'inspecteur sont : A, B, C, D, E, F, G et H. Cette figure montre que la fonction principale lance $M+1$ threads parallèles. Les $M+1$ threads nécessaires à l'inspecteur appartiennent à un enregistreur de paramètres, $M/2$ générateurs de trace et $M/2$ vérificateurs. Notez que, pour chaque thread qui lance un générateur de trace, il existe un thread qui lance une fonction de vérification. L'enregistreur de paramètres et les $M/2$ générateurs de traces nécessitent au minimum $M/2+1$ différents clones légers des fonctions impactantes. Les clones de i^{me} fonctions sont appelés : A_l^i, B_l^i, C_l^i, D_l^i, E_l^i, F_l^i, G_l^i and H_l^i.

Les $M/2$ vérificateurs nécessitent la création de $M/2$ fonctions légères, chacune étant construite en utilisant des blocs de base légers basés sur le modèle NLR. La i^{me} fonction de vérification est appelée v^i ; v^i est supposé être construite à partir de $L+1$ blocs de base impactants légers : BB_0^i, BB_1^i,..., BB_L^i. Dans les fonctions de vérification, le graphe

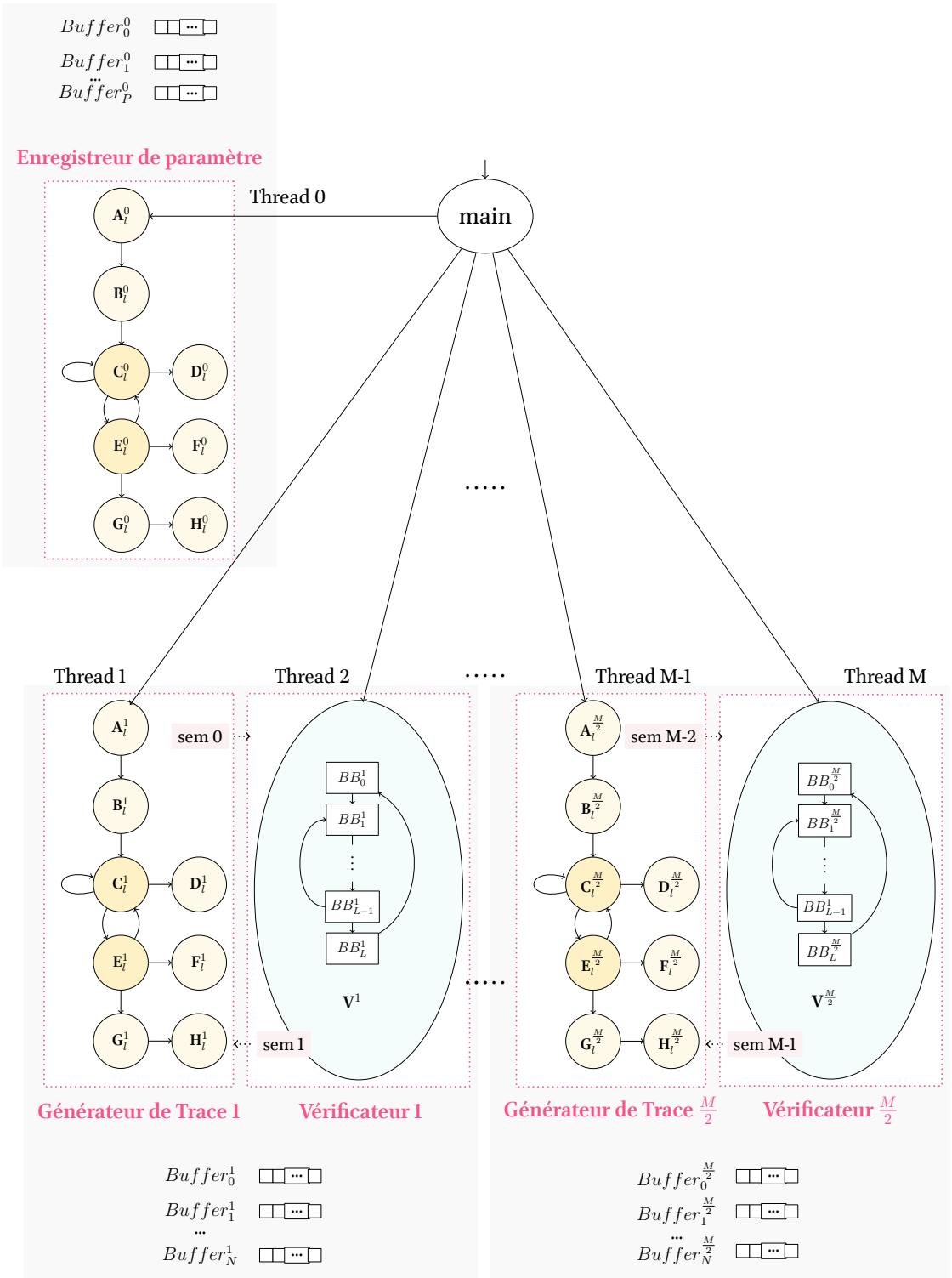


FIGURE A.5 – Exemple de graphe d'appels détaillé pour un inspecteur

de flot de contrôle composé de nœuds représentant les blocs de base est visualisé ; dans ce graphique, les boucles construites apparaissent comme des cycles. Chaque générateur de trace et son vérificateur associé ont leur propre ensemble de $N + 1$ buffers à traiter, et ils se synchronisent, communiquent pour contrôler leurs accès aux buffers en utilisant deux sémaphores. En conséquence, l'inspecteur a besoin de M sémaphores. Par exemple, Thread 1 et Thread 2 communiquent en utilisant les sémaphores sem 0 et sem 1. Chaque générateur de trace produit et enregistre sa propre partie de trace à l'aide de ses buffers dédiés qui sont vérifiés par le vérificateur associé. D'autre part, l'enregistreur de paramètres a son propre ensemble de $P + 1$ buffers utilisés pour sauvegarder les valeurs de paramètres des clones de fonctions impactantes.

A.7.2 Génération de l'Exécuteur

Cette dernière partie de la phase de génération de code prend en entrée le programme récursif après la globalisation des variables, et le modèle de boucle affine NLR correspondant obtenu lors de la phase de profilage. Rec2Poly peut construire le programme de boucles affines de remplacement en :

- clonant les instructions impactantes dans les blocs de base impactants ;
- remplaçant les adresses mémoire référencées par les fonctions affines NLR correspondantes ajoutées aux adresses de base associées collectées au moment de l'exécution ;
- remplaçant les indices de boucles existants par leur expression affine en termes des nouveaux indices de boucles.
- remplaçant le compteur d'invocation utilisé par son expression affine en termes des boucles NLR construites.
- remplaçant l'utilisation des valeurs d'entrée de fonctions manquantes par les valeurs collectées au moment de l'exécution par l'inspecteur.

Enfin, la fonction appelée dans le programme d'origine pour initier la récursivité est remplacée par la nouvelle fonction créée constituée du code itératif construit à partir du modèle de boucles affines NLR. Comme mentionné précédemment, NLR peut produire deux types de modèles de boucles pour un contrôle et un comportement de la mémoire entièrement ou paramétriquement affines. Nous utilisons différentes approches pour optimiser chacun de ces types de boucles affines, comme expliqué ci-dessous.

Boucles avec Contrôle et Comportement Mémoire Affines

Le cas le plus favorable correspond à des boucles affines pures qui sont prêtes à être optimisées à l'aide d'un optimiseur polyédrique automatique comme Pluto. Cependant, puisque nous générons et transformons du code en représentation intermédiaire LLVM, nous devons alimenter Pluto avec une représentation OpenScop des boucles affines comme c'est le cas dans Apollo.

Boucles Parametriquement Affines

Dans ce cas, les optimiseurs automatiques polyédriques ne peuvent pas être utilisés. Cependant, une parallélisation de boucle peut être obtenue, ce qui nécessite un processus d'analyse des dépendances dédié. Il consiste à calculer les plages d'adresses mémoire touchées par les instructions de stockage et de chargement à chaque itération, afin de construire des ensembles d'itérations indépendants. Enfin, la boucle externe est divisée en deux boucles imbriquées : la boucle externe itérant sur des listes de valeurs d'indices de boucle et la boucle interne sur les valeurs d'indices à l'intérieur de chaque liste. La boucle externe est parallélisé en threads parallèles.

A.8 Expériences

Les programmes suivants ont été compilés avec Clang version 6.0 et les options "-O3 -march = native", et fonctionnent sur deux processeurs Intel Xeon E5-2650 v3 à 2,30 GHz de dix cœurs chacun. Les programmes parallèles ont été exécutés en utilisant vingt threads sur les vingt cœurs de la plate-forme matérielle. Les optimisations de l'exécuteur avec les boucles affines sont appliquées grâce à Pluto.

Quant à celles de l'exécuteur avec les boucles paramétriquement affines, elles sont appliquées sur la base de l'approche correspondante décrite dans la section précédente. Par défaut, quelques optimisations sont activées pour les inspecteurs. Par exemple, les boucles sont interrompues après au maximum trois itérations dans le générateur et le vérificateur de trace de contrôle dans un inspecteur. En plus, il y a un couple générateur-vérificateur de trace séparé dédié pour gérer les variables d'induction de boucles et les valeurs du compteur d'invocation des fonctions. En ce qui concerne les générateurs et vérificateurs de traces mémoire, nous avons expérimenté différentes options d'optimisation disponibles dans Rec2Poly.

A.8.1 Multiplication récursive de matrices

Notre première expérience est pédagogique, menée sur une implémentation en C de la multiplication récursive de matrices. Ce programme n'implique qu'une seule fonction récursive avec trois appels récursifs. Le code C de la fonction récursive est donné en figure A.1. Le modèle NLR généré par Rec2Poly pour ce programme (pour des matrices de taille 1000×1000) est présenté à la figure A.6.

L'inspecteur par défaut généré par Rec2Poly est composé de huit couples générateur-vérificateur de trace, un couple qui gère le contrôle, sept couples qui gèrent les générateurs de trace mémoire car il y a sept accès requis pour effectuer le seul calcul existant dans la boucle `C [i] [j] + = A [i] [k] * B [k] [j]`. Cette version de l'inspecteur nécessite seize threads. La version de sauvegarde s'exécute sur un thread supplémentaire. Cependant, Rec2Poly peut optimiser encore plus cet inspecteur et supprimer les accès mémoire redondants. L'inspecteur optimisé comprend six couples générateur-vérificateur de traces dont cinq au lieu de sept sont dédiés à la vérification du flot mémoire de cette récursivité. Cet inspecteur nécessite douze threads en plus du thread de sauvegarde exécutant le code récursif d'origine. L'exécuteur implique des nids de boucles

affine parfaits composés de trois boucles pouvant être pavées et parallélisées par Pluto pour être exécutées par une quarantaine de threads.

```
void MatrixMultiplication(double **A , double **B , double **C , int i ,
    int j , int k){

    if (i>=ROW1) //all rows of A are handled
        return;

    if(k<COLUMN1 && j<COLUMN2) //not all columns of A & rows of B handled
    {
        C[i][j] += A[i][k]*B[k][j];
        MatrixMultiplication( A , B , C , i , j , k+1 );
    }

    else if (j<COLUMN2) //not all columns of B are handled
        MatrixMultiplication( A , B , C , i , j+1 , 0 );

    else //not all rows of A are handled
        MatrixMultiplication( A , B , C , i+1 , 0 , 0 );
}
```

Listing A.1 – Fonction récursive en C du produit de matrices

```
for i0 = 0 to 999
  for i1 = 0 to 999
    for i2 = 0 to 999
      val MatrixMultiplication::if.then3
        , 1*i0 , 1*i2 , 1*i2 , 1*i1 , 1*i0 , 1*i1 , 1*i1
```

FIGURE A.6 – Modèle NLR du contrôle et du comportement mémoire du produit de matrice récursif

Dans la figure A.7, nous montrons le pourcentage d'accélération du temps d'exécution, par rapport au code de multiplication matricielle récursif d'origine, lorsqu'on exécute le code inspecteur-exécuteur et l'inspecteur optimisé avec le code exécuteur. Comme nous l'avons remarqué, l'inspecteur-exécuteur par défaut fonctionne mieux que le code d'origine avec une amélioration d'environ $15,5\%$, et la version optimisée de l'inspecteur-exécuteur est encore plus rapide ; l'accélération est d'environ $30,5\%$ pour les matrices A de taille 10000×900 et B de taille 900×1000.

A.8.2 Multiplication récursive de matrices GEMM

Notre deuxième expérience a été menée sur une autre implémentation en C de la multiplication matricielle récursive (GEMM) manipulant des sous-matrices par dichotomie successive jusqu'à un seuil donné. Il s'agit d'une récursivité indirecte entre quatre fonctions. Il existe une fonction accessible à partir de cette récursivité qui comprend une séquence de boucles affines accédant à la mémoire. La récursivité dans ce programme a

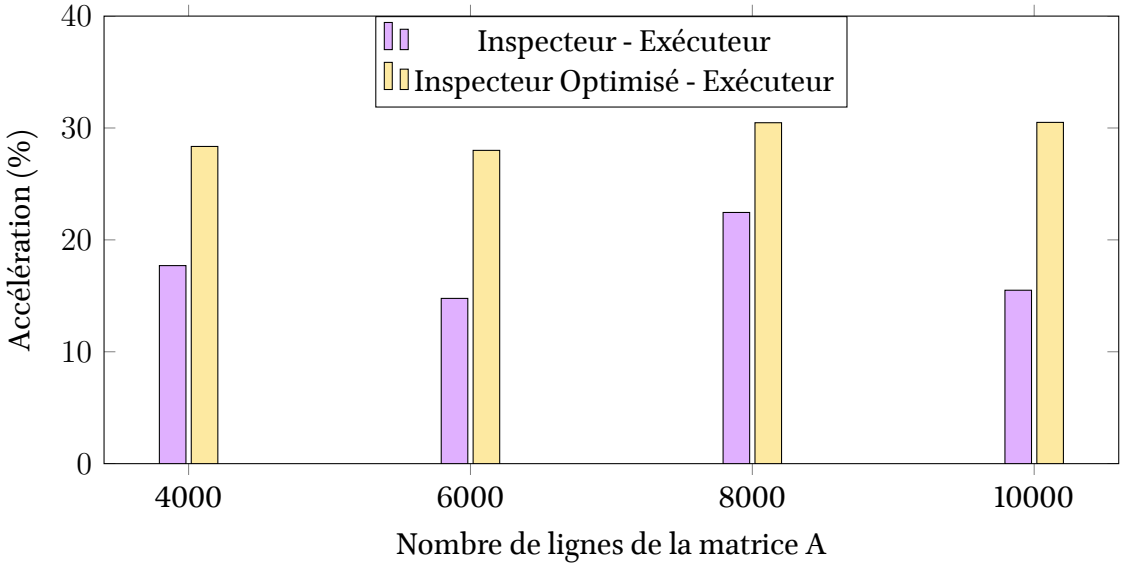


FIGURE A.7 – Résultats expérimentaux du programme Multiplication récursive de matrices - Accélération de Rec2Poly

un contrôle linéaire, mais un comportement mémoire paramétriquement affine. Ainsi, le code itératif gère des listes indépendantes d'itérations exécutées en parallèle.

L'inspecteur par défaut (inspecteur I) généré par Rec2Poly comprend un enregistreur de paramètres et sept couples générateur-vérificateur de traces, dont cinq sont liés à la mémoire. Ainsi, l'inspecteur I nécessite le lancement de quinze threads autres que le thread du code récursif de sauvegarde. Cependant, il existe également de nombreux accès mémoire redondants dans les fonctions impactantes de ce programme, ce qui signifie que l'inspecteur peut être optimisé davantage. Rec2Poly peut générer l'Inspecteur II qui ne nécessite que deux couples générateur-vérificateur de trace mémoire, c'est-à-dire quatre au total. Inspecteur II est lancé sur neuf threads au lieu de quinze. De plus, étant donné que les fonctions impactantes impliquent des boucles sans impact sur le flot, Rec2Poly peut générer une version d'inspecteur très optimisée de l'inspecteur II, qui est l'inspecteur III. Les boucles dans les générateurs de trace et les vérificateurs sont interrompues après au plus trois itérations.

Dans la figure A.8, nous montrons le pourcentage d'accélération du temps d'exécution, par rapport au programme GEMM récursif d'origine et pour des exécutions multiples des codes inspecteur-exécuteur pour différentes tailles de matrices. Tous nos codes optimisés ont surpassé en performance le code récursif d'origine. Le code de l'inspecteur-exécuteur, avec l'inspecteur par défaut (I), est plus rapide que le code d'origine d'environ 20%. Pourtant, le code de l'inspecteur-exécuteur avec l'inspecteur optimisé II et la suppression des accès redondants à la mémoire est encore plus efficace ; il est plus rapide que le code original d'environ $28,9\%$. Enfin, le code de l'inspecteur-exécuteur avec l'inspecteur (III) le plus agressivement optimisé est plus rapide de 83%.

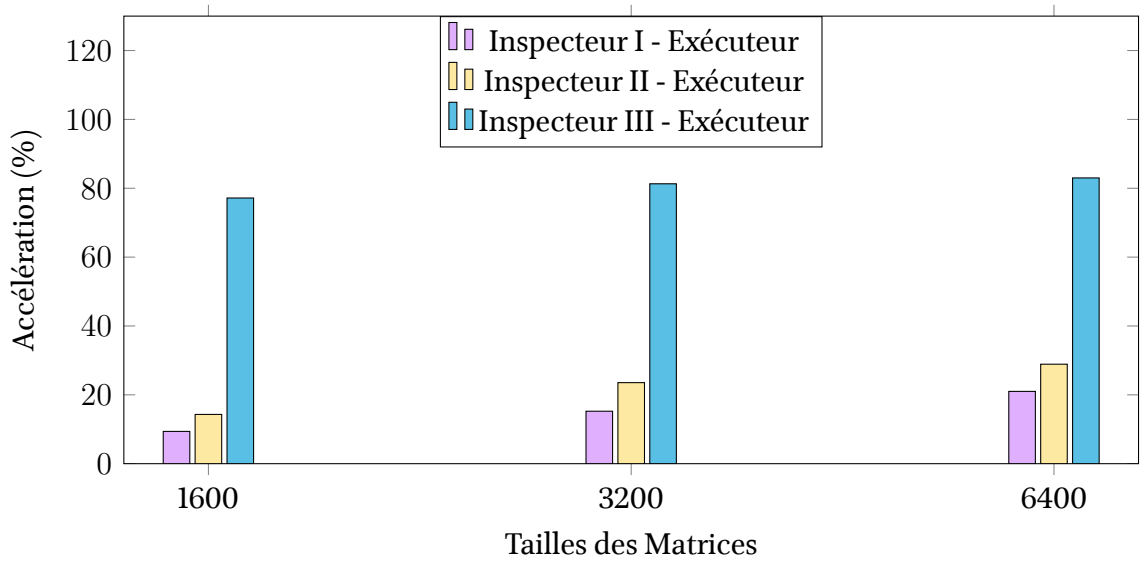


FIGURE A.8 – Résultats expérimentaux du programme GEMM - Accélération de Rec2Poly

A.8.3 Heat

Notre troisième expérience a été menée sur le programme Heat qui est une implémentation récursive en C d'un calcul de type stencil. Il comporte une récursivité directe à partir d'une boucle. Ses fonctions accessibles incluent également des boucles «for» accédant à la mémoire de nombreuses fois. Dans cet exemple, la récursivité perturbe l'existence de boucles, et il est intéressant de voir combien nous pourrons gagner en performance en terme de temps en supprimant la récursivité et en appliquant des optimisations polyédriques à l'imbrication de boucles sans récursivité. Le contrôle et le comportement de la mémoire sont tous les deux linéaires. Par conséquent, un code de boucles affines peut être construit automatiquement à partir du modèle de boucles affines NLR. Le code itératif affine équivalent à ce programme récursif peut ensuite être parallélisé à l'aide de Pluto.

L'inspecteur par défaut généré par Rec2Poly est lourd puisqu'il doit gérer beaucoup d'accès mémoire. Il existe un bloc de base avec au moins seize accès à la mémoire, ce qui nécessite la création de seize couples de vérificateurs / générateur de traces mémoire pour la vérification, ce qui induit une surcharge. En conséquence, pour transformer ce programme à l'aide de Rec2Poly, nous considérons les versions optimisées de cet inspecteur car Rec2Poly peut permettre trois optimisations différentes pour celui-ci.

Le bloc de base impactant avec les seize accès mémoire dans le programme initial comprend de nombreux accès mémoire redondants. Le tableau B est indexé plusieurs fois en utilisant une fonction affine des indices des boucles les entourant. Il suffit de vérifier seulement trois de ces accès. Ainsi, Rec2Poly génère l'inspecteur I qui nécessite un enregistreur de paramètres et cinq couples générateur-vérificateur de trace, un pour le contrôle, un pour les valeurs d'indices de compteur et de boucles et trois pour les accès mémoire. L'inspecteur I nécessite onze threads en plus du thread du code de sauvegarde fonctionnant en parallèle.

Nous avons également mis en œuvre une fonction d'optimisation qui permet aux utilisateurs de sélectionner en fonction du modèle NLR, des blocs de base lourds ou des

blocs de base accédant à la mémoire dans des boucles relativement coûteuses. Cette option d'optimisation n'était pas intéressante pour les autres expériences, mais elle peut être utile pour Heat. Elle attribue chaque accès mémoire à tout un couple générateur-vérificateur de trace mémoire qui lui est dédié. Le reste des accès est géré par un couple générateur-vérificateur séparé. Dans le modèle NLR pour Heat, il existe des blocs de base (dont l'un comprend les seize accès mémoire), exécutés à partir d'une boucle, correspondant à une boucle réelle dans le code source de Heat, avec une borne supérieure élevée par rapport aux autres boucles du modèle. En spécifiant ces blocs de base et le nombre d'accès à la mémoire (après la suppression des accès mémoire redondants), Rec2Poly peut générer automatiquement un inspecteur qui comprend un enregistreur de paramètres, un couple générateur-vérificateur de trace de contrôle et cinq couples générateur-vérificateur de trace mémoire. Ainsi il existe un couple générateur-vérificateur de trace mémoire pour chaque accès dans les blocs lourds, et un pour les accès restants dans l'ensemble de la partie récursive impactante. Cet inspecteur, l'inspecteur II, nécessite treize threads.

Enfin, comme il existe de nombreuses boucles sans impact sur le flot mémoire, Rec2Poly peut générer l'inspecteur III, similaire à l'inspecteur I mais avec des boucles s'exécutant trois fois au maximum.

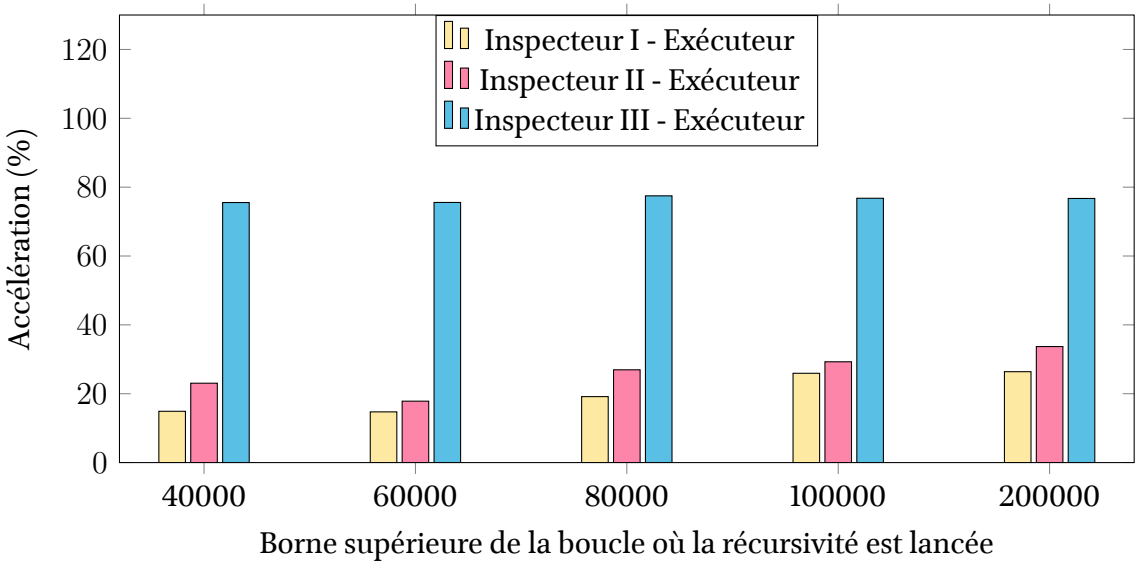


FIGURE A.9 – Résultats expérimentaux du programme Heat - Accélération de Rec2Poly

Les pourcentages d'accélération du temps d'execution, par rapport au code d'origine, pour des exécutions des codes inspecteur-exécuteur (pour des matrices de taille 1024×512) sont représentées à la figure A.9. Inspecteur I - exécuteur fonctionne plus rapidement que le code original d'environ $26,4\%$, Inspecteur II est plus rapide que le code original de $33,7\%$ et, enfin, Inspecteurr III est bien meilleur de $76,7\%$.

A.9 Conclusion

A notre connaissance, Rec2Poly est la première solution d'optimisation de programme spéculative impliquant la réécriture du code cible. Nous avons montré qu'en utilisant

une telle approche, certains programmes récursifs peuvent profiter d'optimisations efficaces des boucles affines, et même profiter de transformations avancées du modèle polyédrique.

Cependant, alors que le mécanisme inspecteur-exécuteur est adapté à de telles optimisations spéculatives, la performance finale est liée probablement principalement à la performance de l'inspecteur. Nous avons montré que l'inspecteur doit également être efficacement optimisé et parallélisé pour réduire sa surcharge en temps. Dans un proche avenir, nous étudierons encore des stratégies pour réduire encore plus la surcharge en temps des inspecteurs. Aussi, il serait intéressant de mettre en œuvre une technique de profilage en ligne comme c'est le cas dans Apollo. De plus, il serait intéressant d'essayer de gérer les fonctions récursives ayant un comportement de contrôle non linéaire. Par exemple, Rec2Poly peut être étendu pour capturer et gérer les fonctions récursives se comportant comme des boucles mais avec des bornes supérieures variables ; bien que de telles boucles ne soient pas affines, elles peuvent toutefois être intelligemment optimisées. En outre, des stratégies de ré-ordonnancement d'instructions, ou "d'affinisation", afin d'obtenir un comportement de contrôle et d'accès mémoire affines, peuvent également être envisagées.

Salwa Kobeissi

Speculative Rewriting of Recursive Programs as Loop Candidates for Efficient Parallelization and Optimization Using an Inspector-Executor Mechanism

Abstract

In this thesis, we introduce Rec2Poly, a framework for speculative rewriting of recursive programs as affine loops that are candidates for efficient optimization and parallelization. Rec2Poly seeks a polyhedral-compliant run-time control and memory behavior in recursions making use of an offline profiling technique. When it succeeds to model the behavior of a recursive program as affine loops, it can use the affine loop model to automatically generate an optimized and parallelized code based on the inspector-executor strategy for the next executions of the program. The inspector involves a light version of the original recursive program whose role is to collect, generate and verify run-time information that is crucial to ensure the correctness of the equivalent affine iterative code. The executor is composed of the affine loops that can be parallelized or even optimized using the polyhedral model.

Résumé

Dans cette thèse, nous proposons Rec2Poly, un cadriciel pour la réécriture spéculative des programmes récursifs sous forme de boucles affines qui sont candidates à une parallélisation et une optimisation efficaces. Rec2Poly cherche un flot de contrôle dynamique et un comportement mémoire conformes au modèle polyédrique dans les récursions, en utilisant une technique de profilage hors ligne. Lorsqu'il réussit à modéliser le comportement d'un programme récursif sous forme de boucles affines, il peut utiliser le modèle de boucle affine pour générer automatiquement un code optimisé et parallélisé basé sur la stratégie inspecteur-exécuteur pour les prochaines exécutions du programme. L'inspecteur implique une version allégée du programme récursif d'origine dont le rôle est de collecter, générer et vérifier les informations d'exécution qui sont essentielles pour garantir l'exactitude du code itératif affine équivalent. L'exécuteur est composé des boucles affines qui peuvent être parallélisées voire optimisées à l'aide du modèle polyédrique.