Profile guided hybrid compilation

Diogo Nunes Sampaio

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministérial : 25 mai 2016

Présentée par

Diogo Nunes Sampaio

Thèse dirigée par **Fabrice Rastello**

préparée au sein du **INRIA**
dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

Profile Guided Hybrid Compilation

Thèse soutenue publiquement le **14/12/2016**,
devant le jury composé de :

Steven DERRIEN
Professor, IRISA, Rennes, France, Rapporteur, Président
Ayal ZAKS
Software Engineer Manager, Intel, Haifa, Israel, Rapporteur
Alexandra JIMBOREAN
Assistant Professor Uppsala University, Sweden, Examinatrice
Christophe GUILLON
Senior Compiler Architect, STMicro, Grenoble, France, Examinateur
Louis-Noel POUCHET
Assistant Professor Colorado State University, Fort Collins, USA, Examinateur
Fabrice RASTELLO
Research Director, Inria, Directeur de thèse

Contents

Dedicado a José e Maria

Abstract

Heat dissipation limitations caused a paradigm change in how computational capacity of chips are scaled, from clock frequency increasing to building hardware with more parallelism. Computer applications must be adapted to explore such parallelism, a hard job left to software developers. To aid in this process many optimizing compilers and frameworks have been developed. In order to apply a transformation to a code, required to better utilize hardware characteristics, compilers must prove that the original programs semantics is preserved. One possible way to do so is by ensuring that the relative execution order between instructions in dependence are preserved in the transformed code. Two instructions are in dependence if both access the same memory cell and at least one is a write instruction. However precise dependence information is very difficult to retrieve.

Code containing memory base pointers that references regions which might overlap, depending on input values, or that contain statements with polynomial memory access expressions, limit the capacity of static analysis to determinate existence, or not, of dependence between pairs of instructions. In such scenario many valid transformations cannot be proven correct, rendering the compiler incapable of performing good optimizations. Dynamic compilers, that realize code analyses and transformation during the program execution, have access to much preciser dependence information, but suffer from limited resources, such as memory and time usage. Hybrid compilers retrieve run-time information from preliminary executions of the program and perform optimizations based on information from both run-time and source-code, but must deal with the fact that run-time behavior is input dependent and the chosen transformation might be invalid to different program inputs.

This works presents a new hybrid compiling technique for optimizing loop nest regions. It statically performs complex loop transformations, guided by instruction dependence information obtained from the source-code and from run-time. For all pairs of instructions in may-dependence, a run-time test verifies if actually the dependence exists, and if the applied transformation changes relative execution order of the two instructions of the original program. If at least one dependence is violated the original version of the loop must be executed, else it is safe to use the optimized code, that is, they express required and sufficient conditions for which the transformation is invalid. However, such tests are extremely costly, if the loop executes $O(n)$ iterations, the test must performs $O(n^2)$ evaluations. Such complex tests are converted into a simpler test, with constant number of expressions to be evaluated, that express required conditions for which the transformation is invalid. This simplification process uses a quantifier elimination scheme, based on Fourier-Motzkin Elimination. Developed techniques remove, as much as possible, imprecision due the use of reals algebra on integer systems.

The proposed quantifier elimination technique produces less imprecise simplifications of integer systems, than other existing tools. The soundness of the framework is demonstrated over 26 kernels using loops with polynomial memory access expressions and pointers in may-alias. Performing complex loop transformations, our framework generates tests that correctly allows the execution of valid transformations and blocks invalid ones, in all performed tests.

Keywords

Code optimization, Compilers, Data dependence, Hybrid Analyses, May-alias, May-dependence, Profiling, Polyhedral compilation, Quantifier Elimination, Symbolic Computation

List of Figures

List of Tables

Chapter 1

Introduction

Many compiler tools have been developed allowing exposition of parallelism and locality so as to better use modern computer resources. However, many difficulties are faced when trying to use these techniques in legacy code where it is not possible to statically obtain accurate instructions dependence information. Whenever applying a code transformation the compiler must prove that the original program semantics is preserved. To do so, it relies on accurate information of dependencies between instructions to prove that changing their original order of execution (schedule) there is no dependence violation. Dependencies represent a producer consumer relationship, where the existence of dependence between two instructions implies that one instruction produces information consumed or overwritten by the other one. The original execution order between two instructions in dependency cannot be changed in order to preserve semantics. Whenever static dependence analysis are inaccurate, the compiler most act conservatively and assume existence of dependencies wherever their nonexistence is not proven, highly limiting the possible code transformations applicable. This work relies on run-time information to overcome such limitation when optimizing loops. It presents a new compiler toolkit that statically applies complex loop transformations using dependencies information obtained from run-time. To assure semantics is preserved, lightweight run-time checks guard the optimized region validating if the applied transformation does not violate any dependencies for the given inputs, choosing between the original or optimized version of the code accordingly.

1.1 Challenges after the free lunch era

For decades CPUs[1] frequency, consequently processing capacity, had grown exponentially. Investments in developing good programming methodologies and compiler optimizations were mostly restricted to academia researchers and a few centers of high performance processing. As for the market, obtaining better performance was only a matter of waiting for the next generation of computers. Known as the "free lunch era" (Sutter, 2005), this growth came to a halt around the year of 2005, when heat dissipation limited further frequency scaling. As transistor scaling did not come to a halt, to continue the computational capacity growth, chip manufacturers went parallel, embedding multiple independent processing units in a single chip. The problem with this new trend is that it moved the responsibility of performance increase from chip manufacturers to software developers. From a world where productivity was the main concern in the software development industry, the new reality required production of efficient and parallel code.

[1] Central Processing Units

Software developers faced a very hard challenge, such as turning legacy, undocumented, or syntactically poor into efficient, parallel software. To make it all worse, very little support was given by the existing compilers of the time.

Retrieving good code transformations is, just as for developers, a very hard task for compilers. From a compilers perspective, a program defines a set of instructions to be executed by a given schedule. Among others things, code transformations consists in changing the scheduling of the execution of all these instructions to better use hardware parallelism. Validating the correctness of a transformation requires precise data dependence analysis, to ensure that the new schedule does not instantiate an instruction that consumes data before it is actually available, or overwrites it before it has been used. Although extensively studied, such analyses are far from being precise (Hind, 2001) nor *"applicable under the environmental conditions of realistic languages"* (Smaragdakis and Balatsouras, 2015), leaving the difficult task to the programmer. Once conquered the difficulties of transforming a code to exploit parallelism, one might find out that performance might be limited due IO[2] bandwidth limitations.

Same as CPU computing capacity, main memory bandwidth experienced exponential grow, however with a much different factor. If this difference prevailed, eventually all programs that used main memory would be *memory bound* Wulf and McKee (1995). That is, application execution times would have been bound solely by data movements to and from the main memory. It is clear that processor frequency scaling halted before reaching such plateau. But it is common to detect memory intensive applications (i.e. many data are retrieved from memory for performing just a few computations) that can saturate the available bandwidth of the main memory without saturating the parallel processing capacity Pirk et al. (2014). Further studies Williams et al. (2009) go deeper in applying, for a single system, different IO bandwidths and processing limits.

1.2 Motivation

Many different methodologies have been developed for tackling the difficult task of generating efficient code, such as:

- distributing domain specific high performance libraries or domain specific languages and specialized compilers (e.g. SDSL (Henretty et al., 2013)),
- new programming languages or extensions with embedded parallel structures and operations (e.g. X10 (Vijay et al., 2016; Charles et al., 2005)),
- hardware specific extensions to existing languages (e.g. CUDA-C (NVIDIA, 2016))
- enriching the current languages syntax to aid compiler analyses (e.g. `restrict` keyword (ISO, 1999, 6.7.3.1)),
- run-time inspection and transformations capable of using information only available during execution to decide, speculatively, upon good transformations (e.g. APOLLO (Sukumaran-Rajam et al., 2014; APOLLO, 2016)).

Each different methodology contains benefits and downsides. For example, techniques focused on a specific domain or hardware require specialization of the developer. Enriching code syntax requires human intervention when dealing with legacy code. Hybrid compilation usually requires a amount of time for profiling and an iterative compilation framework. After all, the compiler must now not only

[2] Input/Output

know about the underlying hardware. But it must as well know which technique better suits the problem being implemented, and must also have a deep comprehension of the code being optimized.

The Polytope model (Feautrier, 1992a,b; Bondhugula, 2008) underlying theory can be tracked back to late 60's (Karp et al., 1967). It was built up over two decades (Lamport, 1974; Cousot and Halbwachs, 1978; Feautrier, 1988a,b; Irigoin and Triolet, 1988; Pugh, 1991; Feautrier, 1992b). It consists of a sound formalism that allows to abstract, in an algebraic way, the iteration space of multi-dimensional loops and schedules for its iterations and instructions. In this framework, memory accesses must appear as affine expressions in terms of the loop counters, allowing effective detection of dependencies between them. Proof of concepts of such techniques were initially applied to Fortran or restricted C code, in the context of high performance computing.

Over a decade associated tools had been implemented and matured. Some examples are CLooG (Bastoul, 2004), PluTo (Bondhugula, 2008), ISL (Verdoolaege, 2010) and *PoCC* (Pouchet, 2010). Achieving promising performance results using automated source-to-source optimizers until the point when embedding developed techniques in main stream compilers became the next natural step: Projects such as GCC Graphite (Pop et al., 2006) and *LLVM Polly* (Grosser et al., 2012) vanguard the efforts of expanding polyhedral optimizations usage beyond small scientific kernels and controlled source-to-source environments. However effectuating this step has demonstrated to be very hard.

Obstacles for applying the polyhedral model grew from inherent problems of compiler intermediate representation (IR) lowering, e.g array linearisation and type erasure, from the use of pointers with possibly multiple indirections, and the existence of dynamic parts that static analyses cannot cope with (Trifunovic et al., 2010). The fact is that, the closer a code representation is to the binary, the less semantic it holds, and thus the more imprecise the results static analysis become. Much work has been done to overcome these problems, from high level semantic reconstruction out of IR (Grosser et al., 2015b), use of suitable code approximations (Benabderrahmane et al., 2008), to extending the polyhedral model (Größlinger, 2009; Venkat et al., 2014a). Speculative code transformations (Jimborean, 2012; Sukumaran-Rajam et al., 2014) demonstrate how run-time analyses can enable the use of polyhedral tools to applications that, during execution, present a behavior that fits to the polyhedral model, not retrieved statically. However, it imposes a online validation overhead with procedures to perform bookkeeping and rollback.

The goal of this project is to allow the optimization of applications with imprecise data dependence information. Formally speaking, it allows the use of polyhedral tools over code previously thought inaccessible to such framework: loop nest regions containing memory accesses with polynomial expressions and may-alias base pointers. It does so by using hybrid analysis (Rus et al., 2002) techniques, that is, a combination of static and dynamic analyses.

Observed run-time dependencies, retrieved by executing an instrumented version of the application, are used to statically optimize the code. When validating if the applied transformation preserves the original loop semantics, instruction dependencies that cannot be resolved statically are resolved at run-time through a test that describes, both sufficient and required, conditions for which the transformation is *invalid*. Such test is dependent on loop counters in such manner that its evaluation cost grows with the number of iterations performed by the transformed loop. Using a quantifier elimination technique, based on Fourier-Motzkin elimination, the test is transformed to an loop invariant test, that is, has an evaluation cost independent of the number of iterations performed by the optimized loop. This test guards the transformed loop region, selecting either the optimized loop or the original for execution, depending on the test result.

1.3 Contributions

This work presents a novel methodology for using application execution profiling to statically generate safe, yet complex, loop optimizations. Its contributions can be summarized as:

Lightweight dependence profiling: Our hybrid analysis uses information captured from run-time to help filling the gaps that cannot be solved by static dependence analysis. It retrieves run-time dependencies by instrumenting memory access instructions, that is, inserting operations into the program to monitor and dump into a trace file, memory positions accessed during execution. In possession of the trace file and the source-code our framework builds the dynamic dependence graph (DDG – described in §2.3), with observed dependence among dynamic instructions.

Two problems emerge when working with DDGs: First, tracking dependence among all instructions might generate spurious dependencies, that can prevent the optimizer to perform good transformations. For example, tracking dependencies of loop trip counter variables (also called induction variables) would tell the optimizer that there always exists dependencies among two consecutive iterations of any loop. These dependencies would prevent transformations that change execution order between two loop iterations. Second, the number of nodes in the graphs is proportional to the analyzed program complexity and operating over them is both memory and time consuming.

Our framework relies in Scalar Evolution (Pop et al., 2005) analyses to avoid tracking variables with a static known regression formula that expands to expressions affine in induction variables, ignoring spurious dependencies. To reduce generated graph sizes our framework allows limiting the number of loop iterations traced. To preserve important dependencies a presented clamping technique unites non-traced instructions in a single graph node.

Optimizing loops with inaccurate dependence information: Code optimizers require precise dependence information to retrieve profitable transformations and to validate that a given transformation preserves the program semantics. However, evaluating existence of dependencies might only be solvable at run-time. Examples of such code is the use of may-alias base pointers[3] and memory accesses with polynomial expressions, as existing analysis avoid handling polynomials due their complexity. In such scenarios, static dependence analysis provide inaccurate information, classifying many pairs of statements as may-dependent and preventing optimizers to perform complex transformations.

To overcome the problem of retrieving complex loop transformations, our framework uses observed run-time information as the absolute truth for instructions in may-dependence. It requires that the monitored execution describes the usual behavior of the application, and that a profitable optimization for that execution is also profitable in most cases.

Tight run-time data dependence validation: To validate correctness of the transformation, a run-time test is generated covering all possible breaks of data dependence. Done naively, the test evaluates with quadratic cost of the loop being optimized That is, if the original program has $O(n)$ complexity, the generated test is $O(n^2)$.

Such tests are expressed as systems of inequalities over quantifiers, related to loop trip counter (or loop *induction variables*). Using the simplifier mentioned below, all quantifiers are eliminated, resulting in an overall test with a constant execution complexity, that is, $O(1)$.

[3]May-alias pointers: Two memory pointers are in may-alias if the existence of overlapping of the regions they reference is unresolved.

Simplifier: The core element that allows the generation of lightweight and tight tests. Based on an extension of *Fourier-Motzkin Elimination* (FME), it projects a precise and costly dependence violation test into a simpler one. As the experimental results conducted show, the proposed *Quantifier Elimination Algorithm* (QEA) constitutes a realistic and competitive alternative to existing state of the art schemes based on cylindrical algebraic decomposition (CAD). Three difficulties successfully faced, and handled by this work are:

1. The amount of inequalities can grow exponentially with the number of variables to eliminate.
2. When dealing with multivariate polynomials, variable's coefficients might be complex expressions, with a sign not easily defined, in which all possible signs must be evaluated.
3. When dealing with integer valued inequalities, accumulated rounding imprecision propagated on every variable elimination step, can highly increase the final projection space, in other words, imprecise projection may lead the run-time test to conservatively choose not to branch to the optimized code while it is actually possible.

To tackle those problems an implementation of the theorem described by Schweighofer (Schweighofer, 2002) is presented. It consists of a method that determines if an inequality is implied by a system of inequalities. In the proposed scheme the method is used with multiple different objectives, such as for removing redundant inequalities generated from a variable elimination step, removing redundant systems of constraints, identifying variables coefficient signs, tightening loose constraints and, lastly, identifying and correcting constraints with rounding errors.

Chapter 2

Framework

This work advocates the use of run-time information for a static loop optimizer. It proposes a framework, that allows to apply complex polyhedral transformations on any loop, as long the memory accesses can be statically written as polynomial expressions in loop induction variables and parameters. The proposed framework functions is composed of the following phases:

1. **Profiling of the application**: With the use of known tools such as `perf`, the profiling of the application uses hardware counters, in addition to debug information, to detect loop regions responsible for most of the execution time. For such hot regions, lightweight instrumentation is performed so as to build multiple execution traces and retrieve existing dependencies information.

2. **Retrieving profitable optimizations**: If the code has affine memory accesses, polyhedral source-to-source optimizers can be used for obtaining desired optimizations. If memory accesses are polynomials, application execution is traced, using code instrumentation, and an affine representation of the program's data-flow is generated. This model holds some of the dependencies observed from run-time. Polyhedral optimizers are then used to obtain desirable transformations.

3. **Building the run-time test**: From the applied sequence of transformations and the original program source code, a precise run-time test for validating the correctness of the transformation is built. Two operations are data-dependent with one-another if they access the same storage position, and at least one is a write. If the execution order between those two instructions is changed, the program result will possibly not be preserved. A may-dependence is when it is, statically, not possible to determine if there exists a dependence between two operations. The generated test verifies, for all possible pairs of operations that are in may-dependence and turn out to be dependent at run-time, if their execution order on the original program is preserved by the transformation. Effectively, the test is performed on the Cartesian product of all operations, so if the application has complexity $O(n)$, the test has complexity $O(n^2)$.

4. **Simplifying the generated test**: Validity checks can be viewed as testing the emptiness of a parametric geometric region in a multi-dimensional space: The existence of an (integral) point in this region means that the transformation is non-valid. Algebraically, this region is defined as a system of polynomial inequalities in terms of induction variables (loop counters), and loop-invariant variables (parameters). Simplifying the generated test corresponds to "eliminating" induction variables from the system, by computing a system of inequalities that is an over-approximation of the original one, but made-up only of parameter variables. Using the geometric representation, this elimination

corresponds to a projection of the "forbidden" region to the sub-space of parameters. To this end, we developed a new quantifier-elimination scheme, an extension of the Fourier-Motzkin algorithm. The simplified tests are systems of inequalities between parameters (thus $O(1)$ test). Going back to the geometric representation, it corresponds to testing emptiness of the (over-approximated) projected region: If empty, the transformation is safe. If statically proven the projected system is not empty, then the transformation will always be considered unsafe and is discarded.

5. **Code versioning**: For the transformations not discarded in the previous step, along with the original code, a decision tree is build to select which version of the code is to be executed.

This chapter describes the proposed framework functionality and components. §3 compares the adopted methodology to related works. §4 provides experimental results that demonstrate soundness of the claim that our proposed technique allows more loops to be safely optimized by the generation of safe, tight, run-time tests. §5 and §6 provide technical implementation details of the Quantifier Elimination and the Dynamic Dependence Graph processes. For last §7 concludes and provide future work directions.

2.1 Theoretical Foundations

```
function FW-tri(*T){
for(k=0;k<n;k++)
 for (i=0;i<k;i++)
  for (j=0;j<i;j++)
   S1: T[i*(i-1)/2+j] min= T[k*(k-1)/2+i]+T[k*(k-1)/2+j]

for (k=0;k<n;k++)
 for (i=0;i<k;i++)
  for (j=k+1;j< n;j++)
   S2: T[j*(j-1)/2+i] min= T[k*(k-1)/2+i]+T[j*(j-1)/2+k]
}
```

Algorithm 1: `FW-tri`: Part of a Floyd-Warshall algorithm for a non directional distance graph stored in a packed lower triangular matrix.

To illustrate our notations and terminology, consider the example of Algorithm 1. The program is composed of two nested loops and any statement is uniquely identified by a vector formed by the loop induction variables (loop indices). In the given example, variables `k, i, j` are the loop indices, and the vector that identifies an instance of statement `S1` is given by the vector (k, i, j). With as many elements as the loop nest depth, this vector is called an *iteration vector* ($\vec{iv}$). To distinguish different statements (e.g. `S1, S2` in our example), we use a *canonical representation* (Feautrier, 1992b) of the nested loop, where each $\vec{iv}$ is associated to a unique statement instance. In this form, additional dimensions, called time dimensions, are added to disambiguate instructions and loops, spawning from the same point. Instead of holding a loop induction variable, these dimensions hold the order on which the element is held on the code. In the given example, the two outermost loops (`for(k=0;k<n;k++)`) are in the scope of the function `FW-tri`: a time dimension is inserted as the first element of the vector, becoming (t_0, k, i, j), where t_0 is 0 for the first loop and 1 for the second, making the $\vec{iv}$ / of `S1` become $(0, k, i, j)$ and the vector for `S2` become $(1, k, i, j)$. This representation can be interpreted as seen in Algorithm 2.

```
function FW-tri(*T){
 for(t0=0;t0<2;t++)
  if(t0==0) {
   for(k=0;k<n;k++)
    for (i=0;i<k;i++)
     for (j=0;j<i;j++)
      S1: T[i*(i-1)/2+j] min= T[k*(k-1)/2+i]+T[k*(k-1)/2+j]
 } else if(t0==1) {
  for (k=0;k<n;k++)
   for (i=0;i<k;i++)
    for (j=k+1;j< n;j++)
     S2: T[j*(j-1)/2+i] min= T[k*(k-1)/2+i]+T[j*(j-1)/2+k]
}
```

Algorithm 2: How the canonical $\vec{iv}$ representation can be interpreted for the `FW-tri` example.

A *schedule* is expressed as a function (written T_{S1} for statement $S1$) from the iteration space $\mathcal{D}$ to the scheduling space $\mathcal{D}'$. Both $\mathcal{D}$ and $\mathcal{D}'$ are affine spaces of possibly different dimension. $T_{S1}(0,k,i,j) = T_{S1}(0,k',i',j')$ means that $S1_{0,k,i,j}$ and $S1_{0,k',i',j'}$ are executed in parallel. Using $\prec_{lex}$ notation to represent the lexicographical order, $T_{S1}(0,k,i,j) \prec_{lex} T_{S1}(0,k',i',j')$ means that $S1_{0,k,i,j}$ is executed before $S1_{0,k',i',j'}$. A loop transformation is expressed as a schedule (also called scattering) function, and most compositions of loop transformations can be represented as affine schedules (Feautrier, 1992b). Consider the case where we want to exchange the order in which the counters i and j are iterated for the loop nest encapsulating `S1`, a transformation called *loop interchange*. The corresponding schedule function for $S1$ would write $T_{S1}(0,k,i,j) = (0,k,j,i)$. The canonical form allows expressing the schedule of the original code as the identity function from $\mathcal{D}$ to itself ($\mathcal{D}'$ can be the same as $\mathcal{D}$).

A *data dependence* between two statement instances expresses an ordering constraint (represented as $<_d$) on their respective executions. A data dependence is induced by read and write accesses to the same memory location. Here $S1_{0,k,i,j}$ writes to location $W_1(0,k,i,j) = T + i(i-1)/2 + j$ and reads from locations $R_2(0,k,i,j) = T + i(i-1)/2 + j$, $R_3(0,k,i,j) = T + k(k-1)/2 + i$, and $R_4(0,k,i,j) = T + k(k-1)/2 + j$. There is a read-after-write dependence (also known as *flow* or *RAW dependence*) between $S1_{0,k,i,j}$ and $S1_{0,k',i',j'}$ from W_1 to R_2 as soon as:

1. $(0,k,i,j) \in \mathcal{D}$ strictly precedes $(0,k',i',j') \in \mathcal{D}$ in the original schedule; which is written as $(0,k,i,j) \prec_{lex} (0,k',i',j')$, where $\prec_{lex}$ denotes a lexicographic comparison;
2. read/write locations overlap; which is written as $W_1(0,k,i,j) = R_2(0,k',i',j')$.

For example $S1_{0,3,2,1} <_d S1_{0,4,2,1}$ as $W_1(0,3,2,1) = R_2(0,4,2,1) \equiv T + 2$.

The other kinds of data dependencies we care about are write-after-write dependencies, also called *output* or *WAW dependence* and write-after-read dependencies, also known as *anti* or *WAR dependence*. A read-after-read, or *RAR*, usually is not considered a data dependency.

For any new schedule to be *valid* it must preserve all such dependencies. As an example, for the previously considered loop interchange $T_{S1}(0,k,i,j) = (0,k,j,i)$, the dependence $S1_{0,3,2,1} <_d S1_{0,4,2,1}$ is preserved since $T_{S1}(0,3,2,1) = (0,3,1,2) \prec_{lex} T_{S1}(0,4,2,1) = (0,4,1,2)$.

In the given example, the variable `T` is a base pointer, received as parameter of the function `FW-tri`. For the compiler, its value is unknown, but constant (invariant), inside this function. If in our example

multiple base pointers were received as parameters, without any pointer disambiguation information (e.g. through `restrict` annotation or inter-procedural analysis) the compiler would not be able to determine dependencies between instructions accessing memory addresses derived from those different base pointers. A second problem comes from the use of polynomial memory accesses. Existing compiler analysis consider polynomial expressions as uninterpreted functions, usually leading to the inability to perform any optimizing transformation. The proposed framework leverages all these steps to overcome the outlined problems based on a model built from profiling the run-time behavior of an application.

2.2 Application profiling

This work focus on optimizing loops in which memory accesses can be statically formulated by polynomial expressions of the $\vec{iv}$ and loop invariant variables (parameters). Algorithms 1 and 3 will serve as reference along this chapter. The latter one is an alternative implementation of the *dspmv* routine, present in BLAS/LAPACK (www.netlib.org, 1992) [1].

```
52 void kernel_dspmv(int ni, int nj, double alpha,
53  double beta, double *X, double *AP, double *Y){
54 int i, j, k;
55 double temp2;
56
57 for (i = 0; i < ni; i++){
58   for (j = 0; j < nj; j++ ){
59     S1: temp2 = 0;
60     for (k = 0; k < i; k++) {
61       S2: Y[k*ni+j] += alpha * X[i*ni+j] *
62                        AP[i*(i+1)/2+k];
63        S3: temp2 += X[k*ni+j] * AP[i*(i+1)/2+k];
64      }
65      S4: Y[i*ni+j] = beta * Y[i*ni+j]+alpha*X[i*ni+j] *
66                      AP[i*(i+1)/2+i] + alpha * temp2;
```

Algorithm 3: An alternative C implementation of the same of the `dspmv` routine obtained from BLAS2, using a packed lower matrix `AP` and x, y steps equal to 1.

Since pointers `X`, `AP` and `Y` are parameters, in practice compilers have very limited information about them, preventing static resolution of memory dependencies, and consequently disallowing the use of complex loop transformations. Source-to-source optimizers, using polyhedral techniques, were developed to be used when the programmer can assure the independence of memory addresses accessed from each base pointer, but neither source-to-source compilers, nor embedded compiler tools such as Polly (Grosser et al., 2012) or *Graphite* (Pop et al., 2006) can handle memory accesses with polynomial expressions, such as `AP[i*(i+1)/2+i]`.

This work relies in loop cloning which involves code size expansion. To avoid huge size expansions, it focuses on optimizing only critical parts of the program, so called *hot-spots*. We use `perf`, a well known performance debugging tool, to obtain a sampled distribution of time spent in each part of the

[1] The original application is hand optimized using temporary variables in a form that applying new transformations is not trivial. For simplicity, the example implements only the case of a lower triangular matrix `AP` and x and y strides of 1.

```
# Overhead  Command  Shared Object  Symbol
# ........  .......  .............  ....................
    99.89%  sample   sample         [.] kernel_dspmv.constprop
     0.11%  sample   sample         [.] init_array.constprop.2
     0.00%  sample   libc-2.23.so   [.] _dl_addr
```

Figure 2.1: Distribution of samples among a programs functions reported by `perf`

```
Sorted summary for file dspmv/sample
----------------------------------------------
65.00 dspmv.l.c:63
29.75 dspmv.l.c:62
 4.92 dspmv.l.c:61
```

Figure 2.2: distribution of samples among source lines reported by `perf`

program. Figure 2.1 illustrates how `perf` reports the distribution of samples among the different functions of a program, and Figure 2.2 the distribution among source-code lines of the `dspmv` example. The most time consuming loops can be detected by mapping this information to the programs AST, which we obtained from Clang (Apple, 2007). The AST of our example program is reported in Figure 2.3.

The framework also considers the amount of instructions in that region. The reason for this is that if a large outer loop, i.e. containing many instructions, holds an inner loop that is solely responsible for most of the applications execution time, it might be much simpler to optimize only the inner loop, instead of the entire loop nest. To filter cases such as this one, the framework uses the ratio of execution time by number of instructions in the loop. Observe that this simple heuristic might sometimes wrongly discard some optimization opportunities.

Once the hot-spots have been identified they are instrumented in order to profile them and build a data-flow representation that will be further used for retrieving profitable optimizations from Polyhedral optimizers. The next section describes how the instrumentation and profiling are done.

```
|-FunctionDecl <dspmv.l.c:52:1, line:68:1> line:53:6 used kernel_dspmv...
| `-ForStmt <line:58:4, line:66:6>
|   |-BinaryOperator <line:58:9, col:13> 'int' '='
|   `-ForStmt <line:59:7, line:66:6>
|     |-BinaryOperator <line:59:12, col:16> 'int' '='
|     |-BinaryOperator <line:60:9, col:17> 'double' '='
|     |-ForStmt <line:61:9, line:64:9>
|     | |-BinaryOperator <line:61:14, col:18> 'int' '='
|     |   |-CompoundAssignOperator <line:62:12, col:59> 'double' '+=' ...
|     |   `-CompoundAssignOperator <line:63:12, col:47> 'double' '+=' ...
|       `-BinaryOperator <line:65:9, col:82> 'double' '='
```

Figure 2.3: Clang Abstract Syntax Tree

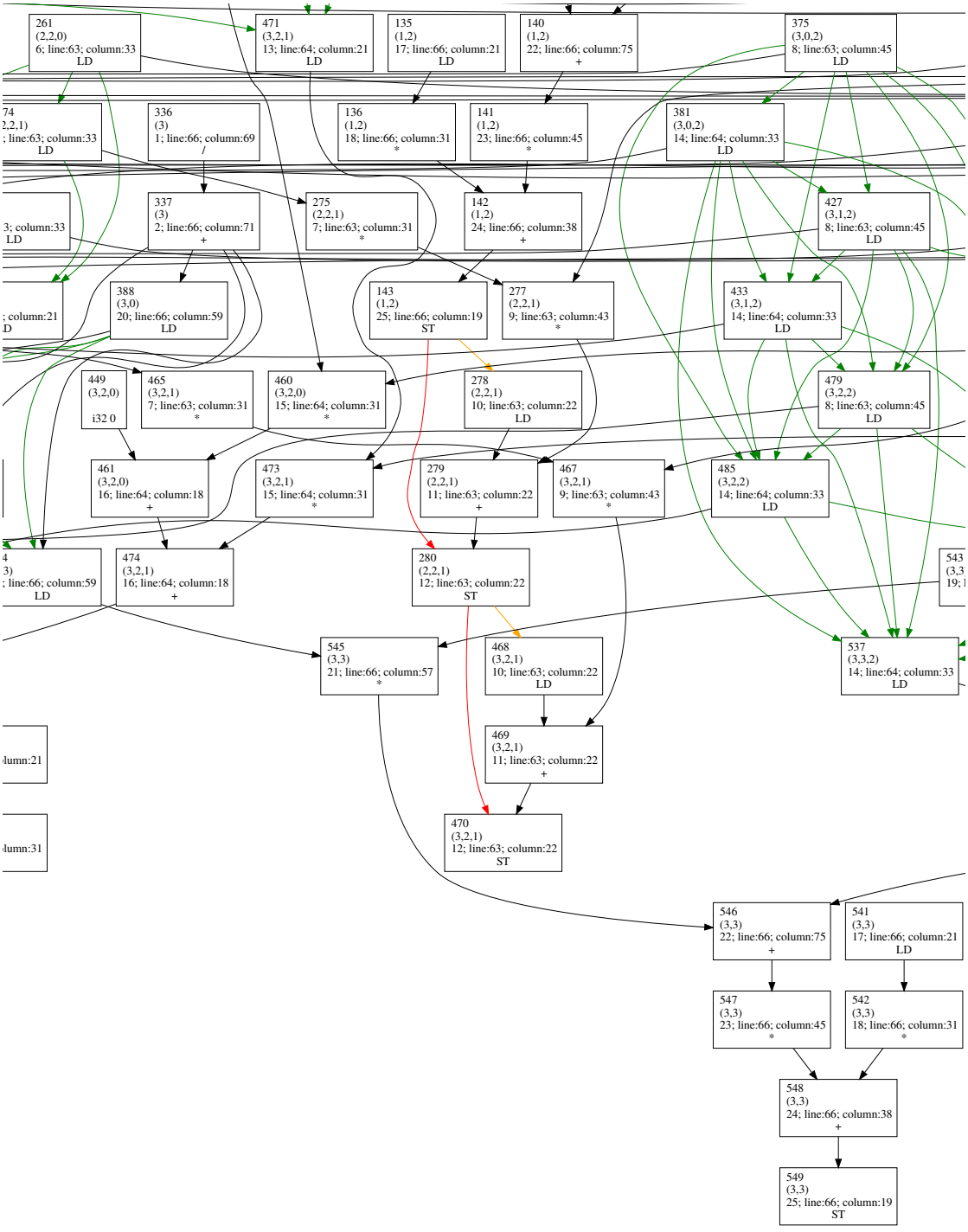

Figure 2.4: Partial view of the DDG combined with the *Dynamic Reuse Graph* (DRG) for `dspmv`. Each node represents a dynamic instruction, described by a unique node counter, the iteration vector, the static instruction identifier, the position (line and column)of the instruction in the source code. Green edges connect instructions that consumed the same data, that is, form a RAR. We differentiate data flow dependencies between scalar variables (visible through def-use chains in the compiler IR), and data flow dependencies between memory locations: Memory RAW are represented in orange and WAW are in red. Scalar RAW are in black. Scalar WAW are ignored as they can be removed using renaming.

2.3 Dynamic Dependence Graph (DDG)

Once the hot-spot regions have been selected, we can profile them. The goal of the profiling phase is to build a DDG: a Directed Acyclic Graph (DAG) where each node represents a dynamic instance of an IR instruction, and each edge represents a data-dependence, that is, a producer–consumer relationship. The construction of the DDG is done in two phases: the first phase is based on instrumentation of the code at the LLVM IR level; The instrumented code is executed, gathering run-time information into a trace file; The second phase uses the collected information so as to simulate the execution and effectively build the DDG.

Collected (chronologically) data during the first phase are: Accessed memory addresses; Alive values in the region entry-point; Base pointers; Loop entries; Loop *iteration counters*; Loop exits.

Using generated trace files along with the compiler IR of the program, the simulation replays the execution. So as to capture dependencies among memory instruction, a *shadow-memory* (Nethercote and Seward, 2007) is used. It consists of storing for every memory address used by the application the last node that wrote to it. Next, whenever a memory read or write operation is performed, a dependence edge is inserted from the last writer to the current reader/writer node. We also build a Data Reuse Graph (DRG) that also contains read-after-read dependencies. In practice, for each write, a set of all nodes that read since this write is built. Observe that write-after-read dependencies could be computed afterward by transitivity from WAW and RAW links.

To identify the operation performed by each node, they receive a label associating it with the static program instruction id that generated them. Merging all nodes of identical instruction-label-id allows to identify dependencies between static elements of the code. As it comes to dynamic dependencies, they are labeled by a distance vector: During the simulation, loop entry and exit events control a dynamic iteration vector. This vector, initially empty, has one new element (set to 0) inserted at its end every time a loop-entry event is detected. This last element is also removed by its corresponding loop-exit event, and in between incremented by one at every loop-iteration event. Whenever a node is created, it is labeled with the current vector, which can be interpreted as an $\vec{iv}$ (not in canonical form here), as explained in §2.1. Figure 2.4 provides an illustrative example of the DDG.

Once dependencies between instructions have been retrieved, the following question is: how to use this information to select code transformations? The next section describes how this graph is used to generate a representation that can be used by existing loop optimizers.

2.4 Retrieving profitable optimizations

Once obtained, the dependencies between operations can be used as an input to existing loop optimizers. The goal is to find new schedules that better exploit machine resources, by e.g. improving data locality or exposing parallelism. However, most existing tools operate on a tree-based representation of loop nests and instructions. They also require a *canonical iteration vector* to distinguish specific instances of each instruction. The DDG, that can be seen as an ocean of nodes, does not hold either qualifications. However the static instruction-id, together with the dynamic iteration vector can be used to map nodes into an AST so as to fulfill those two prerequisites.

Using the compiler's knowledge of the program semantic, the region *Control Flow Graph* (CFG), and the loop forest, a program AST is rebuilt. The constructed AST holds two kinds of nodes: instructions and loops. The hierarchy on the tree represents, from parent node to children, the loop hierarchy from outermost loop to innermost. Instructions are always leaf nodes. Each node is given a unique canonical

iteration vector as described in 2.1. This vector is created by traversing the AST from top to bottom and by: using the induction variables of loops for actual loop dimensions; using static enumerations for additional time dimensions. Lastly, each instruction node in the AST also receives the same static id given to its corresponding node in the DDG.

The mapping of DDG nodes to their respective nodes on the AST is done by filling the induction variables in the canonical iteration vector of the AST, with the dynamic time stamp from the DDG node, giving each node a unique position in the "canonical" iteration space. Each edge is labeled with a *distance vector*, computed by subtraction of the two nodes $consumer - producer$. This distance vector tells how far apart in the execution the two nodes of the dependence relationship are. Observe that the direction of distance vector is the inverse of the orientation of the producer–consumer relationship. Finally, this representation is then either written into a text file to be used with the loop optimizers or into a graphical representation that can be analyzed by the user (see figures 2.5 and 2.6 respectively).

```
Node: 100040000; Inst ID: 100040000; PATH: 0; Type: MV
Node: 5; Loop: 2; PATH: 1; IV: L0 [0, 19]{1}
Node: 6; Inst ID: 1; PATH: 1, L0, 1, 1; Type: /
Node: 7; Inst ID: 2; PATH: 1, L0, 1, 2; Type: +
Node: 9; Loop: 3; PATH: 1, L0, 1, 3; IV: L1 [0, 19]{1}
Node: 100090000; Inst ID: 100090000; PATH: 1, L0, 1, 3, L1, 0; Type: MV
Node: 10; Loop: 4; PATH: 1, L0, 1, 3, L1, 1, 1; IV: L2 [0, L0 + -1]{1}
Node: 11; Inst ID: 6; PATH: 1, L0, 1, 3, L1, 1, 1, L2, 1, 1; Type: LD (C0 + 80*L0 + 4*L1)
Node: 12; Inst ID: 7; PATH: 1, L0, 1, 3, L1, 1, 1, L2, 1, 2; Type: *
Node: 14; Inst ID: 8; PATH: 1, L0, 1, 3, L1, 1, 1, L2, 1, 3; Type: LD (C2 + 4*C1 + 4*L2)
Node: 15; Inst ID: 9; PATH: 1, L0, 1, 3, L1, 1, 1, L2, 1, 4; Type: *
Node: 16; Inst ID: 10; PATH: 1, L0, 1, 3, L1, 1, 1, L2, 1, 5; Type: LD (C3 + 4*L1 + 80*L2)
Node: 17; Inst ID: 11; PATH: 1, L0, 1, 3, L1, 1, 1, L2, 1, 6; Type: +
Node: 18; Inst ID: 12; PATH: 1, L0, 1, 3, L1, 1, 1, L2, 1, 7; Type: ST (C3 + 4*L1 + 80*L2)
Node: 19; Inst ID: 13; PATH: 1, L0, 1, 3, L1, 1, 1, L2, 1, 8; Type: LD (C0 + 4*L1 + 80*L2)
Node: 20; Inst ID: 14; PATH: 1, L0, 1, 3, L1, 1, 1, L2, 1, 9; Type: LD (C2 + 4*C1 + 4*L2)
Node: 21; Inst ID: 15; PATH: 1, L0, 1, 3, L1, 1, 1, L2, 1, 10; Type: *
Node: 22; Inst ID: 16; PATH: 1, L0, 1, 3, L1, 1, 1, L2, 1, 11; Type: +=
Node: 23; Inst ID: 17; PATH: 1, L0, 1, 3, L1, 1, 2; Type: LD (C3 + 80*L0 + 4*L1)
Node: 25; Inst ID: 18; PATH: 1, L0, 1, 3, L1, 1, 3; Type: *
Node: 26; Inst ID: 19; PATH: 1, L0, 1, 3, L1, 1, 4; Type: LD (C0 + 80*L0 + 4*L1)
Node: 27; Inst ID: 20; PATH: 1, L0, 1, 3, L1, 1, 5; Type: LD (4*L0 + 4*C1 + C2)
Node: 28; Inst ID: 21; PATH: 1, L0, 1, 3, L1, 1, 6; Type: *
Node: 29; Inst ID: 22; PATH: 1, L0, 1, 3, L1, 1, 7; Type: +
Node: 30; Inst ID: 23; PATH: 1, L0, 1, 3, L1, 1, 8; Type: *
Node: 31; Inst ID: 24; PATH: 1, L0, 1, 3, L1, 1, 9; Type: +
Node: 32; Inst ID: 25; PATH: 1, L0, 1, 3, L1, 1, 10; Type: ST (C3 + 80*L0 + 4*L1)
100040000 -> 7:RAW[(0) (1,0,2)];
7 -> 6:RAW[(1,0,1) (1,0,2)];RAW[(1,1,1) (1,1,2)];RAW[(1,2,1) (1,2,2)];RAW[(1,3,1) (1,3,2)];
100090000 -> 22:RAW[(1,1,3,0,0) (1,1,3,0,1,0,11)];RAW[(1,2,3,0,0) (1,2,3,0,1,0,11)];RAW[(1,3,3,0
100090000 -> 29:RAW[(1,0,3,0,0) (1,0,3,0,7)];RAW[(1,0,3,1,0) (1,0,3,1,7)];RAW[(1,0,3,2,0) (1,0,3
11 -> 11:RAR[(1,2,3,0,1,0,1) (1,2,3,0,1,1,1)];RAR[(1,3,3,0,1,0,1) (1,3,3,0,1,1,1)];RAR[(1,3,3,0,
12 -> 11:RAW[(1,1,3,0,1,0,1) (1,1,3,0,1,0,2)];RAW[(1,2,3,0,1,0,1) (1,2,3,0,1,0,2)];RAW[(1,3,3,0,
14 -> 20:RAR[(1,1,3,0,1,0,9) (1,1,3,1,1,0,3)];RAR[(1,3,3,0,1,2,9) (1,3,3,1,1,0,3)];RAR[(1,2,3,1,
14 -> 14:RAR[(1,1,3,0,1,0,3) (1,1,3,1,1,0,3)];RAR[(1,3,3,0,1,2,3) (1,3,3,1,1,0,3)];RAR[(1,2,3,1,
15 -> 12:RAW[(1,1,3,0,1,0,2) (1,1,3,0,1,0,4)];RAW[(1,2,3,0,1,0,2) (1,2,3,0,1,0,4)];RAW[(1,3,3,0,
15 -> 14:RAW[(1,1,3,0,1,0,3) (1,1,3,0,1,0,4)];RAW[(1,2,3,0,1,0,3) (1,2,3,0,1,0,4)];RAW[(1,3,3,0,
17 -> 15:RAW[(1,1,3,0,1,0,4) (1,1,3,0,1,0,6)];RAW[(1,2,3,0,1,0,4) (1,2,3,0,1,0,6)];RAW[(1,3,3,0,
17 -> 16:RAW[(1,1,3,0,1,0,5) (1,1,3,0,1,0,6)];RAW[(1,2,3,0,1,0,5) (1,2,3,0,1,0,6)];RAW[(1,3,3,0,
18 -> 17:RAW[(1,1,3,0,1,0,6) (1,1,3,0,1,0,7)];RAW[(1,2,3,0,1,0,6) (1,2,3,0,1,0,7)];RAW[(1,3,3,0,
18 -> 16:WAR[(1,1,3,0,1,0,5) (1,1,3,0,1,0,7)];WAR[(1,2,3,0,1,0,5) (1,2,3,0,1,0,7)];WAR[(1,3,3,0,
19 -> 26:RAR[(1,2,3,3,4) (1,3,3,0,1,0,8)];RAR[(1,2,3,0,4) (1,2,3,1,1,0,8)];RAR[(1,3,3,0,4) (1,3,
19 -> 19:RAR[(1,3,3,0,1,2,8) (1,3,3,1,1,0,8)];RAR[(1,3,3,2,1,2,8) (1,3,3,3,1,0,8)];RAR[(1,3,3,0,
19 -> 11:RAR[(1,2,3,0,1,1,1) (1,2,3,0,1,1,8)];RAR[(1,3,3,0,1,1,1) (1,3,3,0,1,1,8)];RAR[(1,3,3,0,
20 -> 20:RAR[(1,1,3,0,1,0,9) (1,1,3,1,1,0,9)];RAR[(1,3,3,0,1,2,9) (1,3,3,1,1,0,9)];RAR[(1,2,3,1,
20 -> 14:RAR[(1,1,3,0,1,0,3) (1,1,3,0,1,0,9)];RAR[(1,2,3,0,1,0,3) (1,2,3,0,1,0,9)];RAR[(1,3,3,0,
21 -> 20:RAW[(1,1,3,0,1,0,9) (1,1,3,0,1,0,10)];RAW[(1,2,3,0,1,0,9) (1,2,3,0,1,0,10)];RAW[(1,3,3,
21 -> 19:RAW[(1,1,3,0,1,0,8) (1,1,3,0,1,0,10)];RAW[(1,2,3,0,1,0,8) (1,2,3,0,1,0,10)];RAW[(1,3,3,
22 -> 21:RAW[(1,1,3,0,1,0,10) (1,1,3,0,1,0,11)];RAW[(1,2,3,0,1,0,10) (1,2,3,0,1,0,11)];RAW[(1,3,
22 -> 22:RAW[(1,2,3,0,1,0,11) (1,2,3,0,1,1,11)];RAW[(1,3,3,0,1,0,11) (1,3,3,0,1,1,11)];RAW[(1,3,
25 -> 23:RAW[(1,0,3,0,2) (1,0,3,0,3)];RAW[(1,1,3,0,2) (1,1,3,0,3)];RAW[(1,2,3,0,2) (1,2,3,0,3)];
26 -> 11:RAR[(1,1,3,0,1,0,1) (1,1,3,0,4)];RAR[(1,2,3,0,1,1,1) (1,2,3,0,4)];RAR[(1,3,3,0,1,2,1) (
...
```

Figure 2.5: Partial (raw) view of the AST built from DDG. Nodes are either loops or instructions, and have a unique identifier (ID). PATH describes the node position in the AST (canonical iteration vector). Each loop node contains (e.g. `L2 [0, L0 + -1]{1}`): its induction variables name (e.g. `L2`); its domain limits (e.g. `[0, L0 + -1]`); its induction variable steps (e.g. 1). Instruction nodes describe their type (`ST`, `LD`, +, −, ...) and might have relationships (edges) with other instructions: (e.g. `11->11: RAR...`).

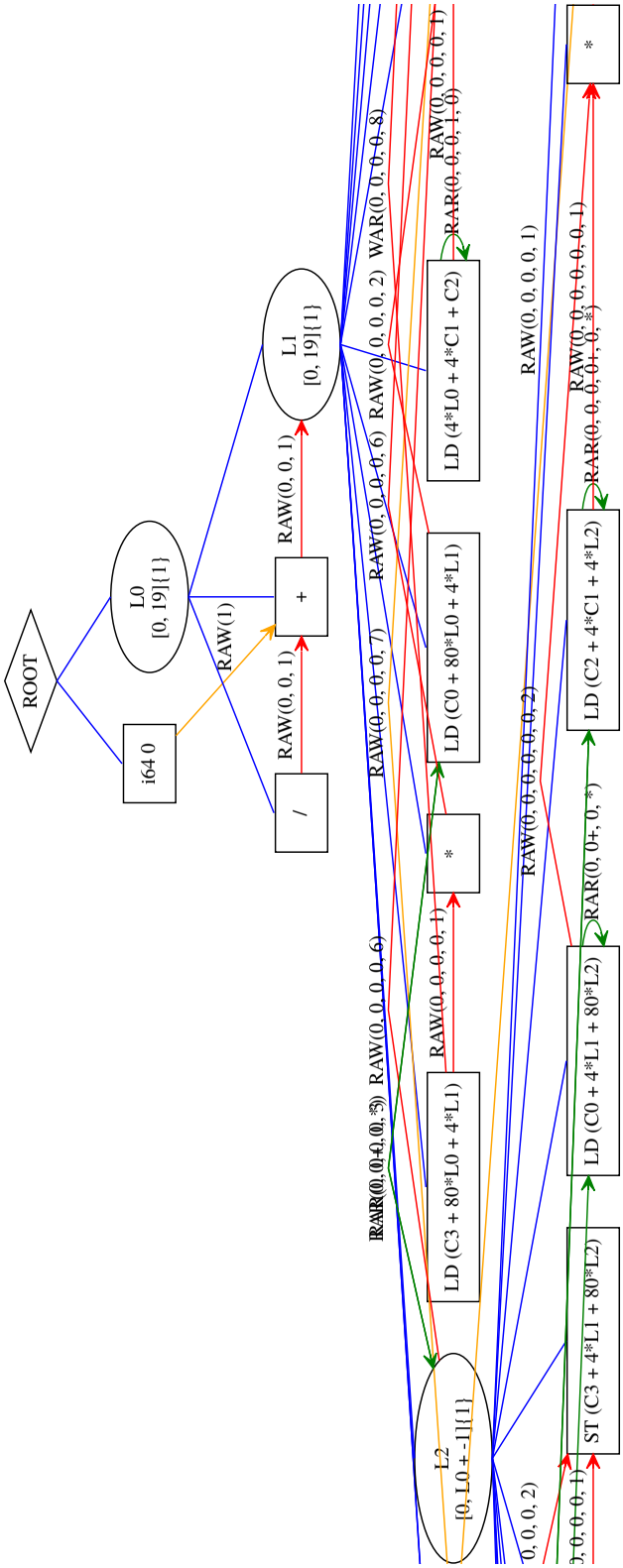

Figure 2.6: Partial view of the graphical representation generated for the `dspmv` example. Elliptical nodes represent loop structures, containing their induction variable counter (e.g. `L0`, `L1`, `L2`), their maximum and minimum values and step size. Squares represent instructions. Red and orange edges represent dependencies between instructions and green edges represent reuses. For easier comprehension, the distance vector directions are inverted to preserve the data flow sense.

```
#pragma parallelize (*)
#pragma schedule S1 (k,i,j)
#pragma tilable () 3
```

Figure 2.7: New schedule for the instructions of `FW-tri`, describing tilling in all three levels and parallelization of the second outermost loop, described in `asteroid` directives.

2.5 Obtaining optimizations

The main goal of the proposed framework is to allow the use of polyhedral loop optimizers over code with poor dependence information, either due to the presence of polynomial memory accesses functions or due to the existence of may-alias between base-pointers. So far we demonstrated how to obtain and use memory profiling to retrieve dependence information, further used to build an affine representation of the loop to retrieve transformations. Next we will present how to ensure that such transformation is safe with regards to data dependencies, for a given program input. However our overall framework is still incomplete as we do not have the ability to feed an optimizing compiler with the DDG yet.

To develop and evaluate our run-time test generation, we use *Polybench 4.1* original applications (with affine access functions and non-aliasing between base pointers). Predefined good transformations for each application, based on the work listed on Park et al. (2013), were performed using PoCC. Both the original and transformed code has then been then linearized. We build our run-time validation tests on the linearized code, for the transformation provided by PoCC from the non-linearized code... For the `FW-tri` and `dspmv` benchmarks that already show polynomial access functions in there original version, we replaced the triangular matrices by conventional square ones with affine memory accesses, so as to be able to get a candidate transformation from PoCC. The hand written version of the first (out of four) loop of FW-tri is displayed in Algorithm 4.

```
void kernel_floyd_warshall(int n, double dist[n][n]){
  unsigned i, j, k;
#pragma scop
 for (k = 0; k < n - 2; k++)
  for (i = k+2; i < n; i++)
   for (j = k+1; j < i; j++)
    S1: dist[i][j] = dist[i][k] + dist[j][k];
#pragma endscop
}
```

Algorithm 4: The **hand written** representation of `FW-tri` used to obtain speculative transformations.

PoCC returns `asteroid` pragma directives to describe candidate transformations, with special directives for parallelization and tiling. Figure 2.7 depicts those directives to describe a thread-level parallelization and full-depth tiling for the `FW-tri` example. Algorithm 5 displays the obtained optimized code, after applying the retrieved transformation to the original program.

```
#pragma scop
 if (n >= 3) {
  for (k = 0; k <= floord(((33 * n)-97), 32); k++) {
   lb1 = ceild((k-29), 33);
   ub1 = min(floord((n-1), 32), k);
#pragma omp parallel for private(j, I, J) firstprivate(lb1, ub1)
   for (i = lb1; i <= ub1; i++) {
    for (j = ceild(((k-i)-30), 32); j <= min(floord((n-2), 32), i); j++) {
     for(I=max(max((32*i),((32*j)+1)),((k-i)+2));I<=min(((32*i)+31),(n-1));I++){
#pragma ivdep
#pragma vector always
#pragma simd
      for (J = max((32 * j), (k-i+1)); J <= min(((32 * j) + 31), (I-1)); J++) {
       if(dist[I*(I-1)/2+J] > dist[I*(I-1)/2+k-i] + dist[J*(J-1)/2+k-i])
        dist[I*(I-1)/2+J] = dist[I*(I-1)/2+k-i] + dist[J*(J-1)/2+k-i]
}}}}}}
#pragma endscop
```

Algorithm 5: `FW-tri` optimized: parallel execution of loop i and tiling of all three loops.

2.6 Building the run-time test

Our run-time tests consist of a set of systems of inequalities over variables and parameters. System variables correspond to loop induction variables; System parameters correspond to program scalar variables unknown at compile time, and constant along the code region being optimized: Such parameters are usually associated to memory pointers and loop boundaries. Each system of inequalities describes the necessary conditions for a given may-dependence between two statements to be violated by the applied transformation. In other words, if this condition is not fulfilled, then it is safe to use the applied transformation.

Our test description heavily depends on polyhedral loop representation concepts. To refresh the terminology consider the simple loop presented below, in Algorithm 6.

```
stencil(int m, int n, int H, int* t, int* a)
for ( j = 1; j < m; j++ )
 for ( i = 1; i < n; i++ )
  S1: t[H*i+j] = ( t[H*(i-1)+j] + t[H*i+j-1] ) * a[j];
```

Algorithm 6: Stencil toy example.

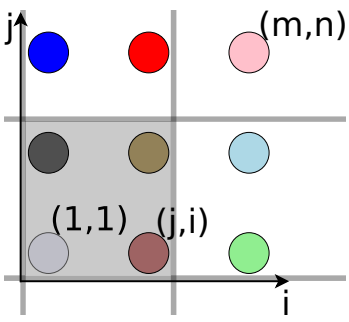

Figure 2.8: Domain space for the statement `S1` from the toy stencil example. A point v is defined by the vector (j, i)

Domain: Each statement A at a loop depth d is mapped to a space $\mathbb{Z}^d$. Each induction variable of the loop nest represents an dimension of this space. Each point in space, described by the induction vector, represents an instance of this statement. As loops do not execute indefinitely, loop bounds define constraints over which points are part of the loop iteration domain. The constrained space defines a polyhedron called the **iteration domain** D_A. In our toy example, the domain space of statement `S1` is repre-

sented by the gray region in Figure 2.8, defined by the constraints:

$$\begin{aligned} & 1 \leq j \\ \wedge\ & j \leq m-1 \\ \wedge\ & 1 \leq i \\ \wedge\ & i \leq n-1. \end{aligned}$$

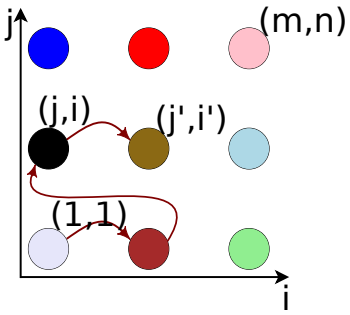

Figure 2.9: Iteration points mapped to the original schedule space (j, i). A single line is completely executed, in the i ascending order. When finished scanning a line, the next line along the j axis is initialized. In this example, iteration (j, i) executes before iteration (j', i').

Original schedule: A schedule maps any point of any statement of the loop to a point in a *lexicographic* ordered space, that defines the execution order of these instructions. For a sequential program, the schedule space has dimensionality the maximum depth of the loop (2 in our example). In the schedule space, points are ordered by sorting their position dimension wise, from a more representative dimension to the lower one. For example, in our toy stencil, all points along the i dimension are scanned before changing to points of higher value of j. For values with same value j, their ordering is defined by the values of i. Thus, to define that $v \prec_{lex} v'$ in the original schedule space we have that:

$$v = (j, i) \prec_{lex} v' = (j', i') \iff (j < j') \vee (j = j' \wedge i < i')$$

Schedules (affine -) are mappings of points of the iteration space to points in the schedule space and can be represented as matrices. In our example, the original schedule, the *identity*, maps every point $v = (j, i) \in D_A$ to the point (j, i) in the schedule space. Figure 2.9 depicts mapped points to the schedule space for our example.

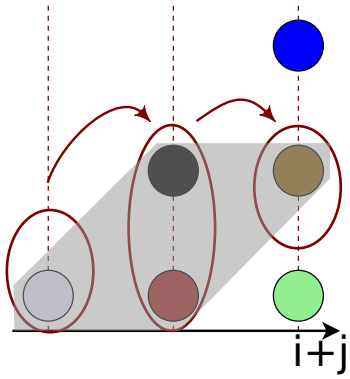

Figure 2.10: Transformed schedule after applying loop skewing and parallelization. Our original 2D loop (j, i) has its instructions scheduled by the 1D vector $(j + i)$.

Transformed schedule: A transformation consists in defining a new schedule space with a mapping different than the *identity* for at least one of the statements of the loop. For our example, an interchange transformation of the two loops would provide a mapping

$$T_I = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

such that

$$(j, i) \times T_I = (i, j).$$

Successive transformations can be applied and the dimensionality of the schedule space might be different from the original one. For example, combining loop skewing and parallelization to our example we might obtain:

$$T_S = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

$$T_P = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$$

$$T_{SP} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix}.$$

For last, applying this transformation to our original representation we obtain:

$$(j, i) \times T_{SP} = (j + i, 0).$$

Here the schedule space has only one dimension, and all instructions with equal values $j+i$ are scheduled to execute at the same time, as shown in Figure 2.10.

Dependence: A pair of instructions (v, v') are in (data) dependence if:

1. They both access the same memory position, that is $M(v) = M(v')$.
2. At least one of them is a write operation.

Usually an instruction that accesses the main memory is either a load or a store instruction. Such instructions define the address to be accessed using expressions over a reference position (the base pointer), over loop induction variables, and over padding values that might only be known at run-time.

In our toy example, an instance of the statement `S1` performs one store and three load operations to memory. To evaluate the dependence between statement instance $v_{S1} = (j, i)$ and $v'_{S1} = (j', i')$ we must compare the addresses of both store operations between each other (one from each instance) and each store against each of the six load operations (three per instance), where each of these comparisons defines one system of constraints.

If two statements have dependencies between them that cannot be resolved statically, we say that those two statements are in *may-dependence* (mD): Our run-time test must, for every instances of those two statements, define if there is an dependence between them and if that dependence is violated by the applied transformation, as we will see below.

2.6.1 Precise dependence violation tests

With the original source code and the transformed schedule, obtained from our loop optimizer, we build a run-time test to validate, between every two statements in may-dependence, if the applied transformation does violate at least one dependence. In other words, the test expresses a necessary condition under which the transformation is invalid. The general formula of the test is:

$$\begin{array}{lll} \bigvee\limits_{(A,A') \in mD} & \exists (v, v') \in \mathbb{Z}^{2s} \text{ s.t.} & \text{Exists pair of instructions} \\ & v \in \mathcal{D}_A \wedge v' \in \mathcal{D}_{A'} & \text{domain} \\ \wedge & v \prec_{lex} v' & \text{original schedule} \\ \wedge & T_A(v) \not\prec_{lex} T_{A'}(v') & \text{transformed schedule} \\ \wedge & M(v) = M(v') & \text{same access location} \end{array}$$

and reads as:

If the transformation is invalid it means that:

$$\begin{array}{rl} \bigvee\limits_{(A,A') \in mD}: & \text{For at least a pair of statements } (A, A') \text{ in may-dependence} \\ \exists (v, v') \in \mathbb{Z}^{2s} \text{ s.t.}: & \text{if exists a single respective pair of instances } (v, v') \\ v \in \mathcal{D}_A \wedge v' \in \mathcal{D}_{A'}: & \text{within the loop domain} \\ v \prec_{lex} v': & \text{where } v \text{ is executed before } v' \text{ in the original code} \\ T_A(v) \not\prec_{lex} T_{A'}(v'): & \text{and } v \text{ is not executed before } v' \text{ in the transformed code} \\ M(v) = M(v'): & \text{and they do access the same memory position, thus they } \textit{really} \text{ are in dependence.} \end{array}$$

Existential instances $\exists(v, v')$ means that all possible statements instances, in may-dependence, must be evaluated, generating a Cartesian product between the sets of instances. A run-time test generated by a direct translation of such systems would have an evaluation complexity quadratic in the computational cost of the original source code. From our toy example, evaluating the violation from the write operation $A = W =$ `t[H*i+j]` against the read operation $A' = R_1 =$ `t[H*(i'-1)+j']` for the applied tilling+parallelism we would obtain:

	$\exists(j, i, j', i') \in \mathbb{Z}^4$ s.t.	Exists pairs (v, v')
	$1 \leq j \leq m \wedge 1 \leq j' \leq m \wedge 1 \leq i \leq n \wedge 1 \leq i' \leq n$	Domain
$\wedge$	$j < j' \vee (j = j' \wedge i < i')$	Original schedule: $v \prec_{lex} v'$
$\wedge$	$(i' + j') \leq (i + j)$	Transformed schedule: $v \not\prec_{lex} v'$
$\wedge$	$t + Hi + j = t + H(i' - 1) + j'$	Same memory position.

Translating it to a run-time test would lead to:

```
error = false;
for ( j = 1; j < m; j++ )                    //Domain v (j)
 for ( i = 1; i < n; i++ )                   //Domain v (i)
  for ( jp = 1; jp < m; jp++ )               //Domain v'(j)
   for ( ip = 1; ip < n; ip++ )              //Domain v'(i)
    error |= (&t[H*i+j]==&t[H*(ip-1)+jp]) //Same memory position
          && (j<jp||(j==jp && i<ip))       //Original schedule v < v'
          && (jp+ip <= j+p)                //Transformed schedule v' <= v
```

Algorithm 7: Stencil toy example test for skewing+parallelism

Our framework can handle any transformation represented by multidimensional time schedules. In a non extensive list, it allows combination of loop parallelization, vectorization, fusion, fission, invariant code motion, skewing, scheduling, software pipelining, splitting, tiling and unrolling. We now illustrate tests systems of equations through different loop transformations to our toy example. We invite the reader to *Optimizing compilers for modern architectures: a dependence-based approach* (Allen and Kennedy, 2002) for detailed description of these different loop transformations.

Permutation (or interchange): In our toy example, every consecutive iteration changes the value of i and keeps variable j constant until i reaches n. The memory accesses to vector t are H elements apart between two consecutive instances, thus locality is dependent on the input value of H. If we perform a permutation of the two loops, that is, we switch the loops execution priority, the innermost loop of the code would iterate over the variable j having consequent instructions to access array t with stride 1, thus showing good spatial locality. Using the permutation transformation for a two loop nest

$$T_I = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix},$$

we obtain the following schedule $\vec{iv}$:

$$(j, i) \times T_I = (i, j)$$

For accesses W and R_1, our test becomes:

$$\begin{aligned} & \exists (j, i, j', i') \in \mathbb{Z}^4 \text{ s.t.} \\ & 1 \leq j \leq m \wedge 1 \leq j' \leq m \wedge 1 \leq i \leq n \wedge 1 \leq i' \leq n \\ \wedge \quad & j < j' \vee (j = j' \wedge i < i') \\ \wedge \quad & \neg (i < i' \vee (i = i' \wedge j \leq j')) \\ \wedge \quad & t + Hi + j = t + H(i' - 1) + j' \end{aligned}$$

Skewing: In our toy example, every consecutive iteration along the i axis depends on the iteration before and every consecutive iteration along j axis also depends on its iteration before: in other words (i, j) depends on $(i-1, j)$ and $(i, j-1)$. Such dependence configuration forbids executing either of these loops in parallel. However, skewing the j axis to the i positive direction, described by the transformation

$$T_S = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix},$$

our obtained schedule holds

$$(j, i) \times T_S = (j, j + i)$$

Such transformation would allow all points with a same $(j+i)$ value to be executed in parallel. Obtained schedule would become $\mathcal{D} : \ (j, i) \rightarrow (j + i)$. For accesses W and R_1, our test for such transformation becomes:

$$\begin{aligned} & \exists (j, i, j', i') \in \mathbb{Z}^4 \text{ s.t.} \\ & 1 \leq j \leq m \wedge 1 \leq j' \leq m \wedge 1 \leq i \leq n \wedge 1 \leq i' \leq n \\ \wedge \quad & j < j' \vee (j = j' \wedge i < i') \\ \wedge \quad & \neg (j < j' \vee (j = j' \wedge j + i < j' + i')) \\ \wedge \quad & t + Hi + j = t + H(i' - 1) + j'. \end{aligned}$$

Loop-invariant code motion: The load operation $R_3 =$ `a[j]` is performed ij times, although deciding the memory address to be loaded only depends on the value of j. Moving this load operation outside the inner-loop, using a local variable to store the value, would reduce the number of times the operation is executed to j times. Such transformation changes the number of statements in our outer loop, thus we must use a virtual dimension to represent both of them. The resulting code becomes:

```
stencil(int m, int n, int H, int* t, int* a)
for ( j = 1; j < m; j++ )
 S2: int tmp = a[j];
 for ( i = 1; i < n; i++ )
  S1: t[H*i+j] = ( t[H*(i-1)+j] + t[H*i+j-1] ) * tmp;
```

Algorithm 8: Stencil toy example after performing loop invariant code motion.

In this case, the W operation $\vec{iv}$ is mapped

$$\text{from } (j, i) \text{ to } (j, 1, i).$$

And the R_3 is mapped

$$\text{from } (j, i) \text{ to } (j, 0, 0).$$

The test to detect broken dependencies writes as:

$$\begin{aligned} & \exists(j, i, j', i') \in \mathbb{Z}^4 \text{ s.t.} \\ & 1 \leq j \leq m \wedge 1 \leq j' \leq m \wedge 1 \leq i \leq n \wedge 1 \leq i' \leq n \\ \wedge \quad & j < j' \vee (j = j' \wedge i < i') \\ \wedge \quad & \neg(j < j') \\ \wedge \quad & t + Hi + j = a + j'. \end{aligned}$$

Each system S of constraints describes violation conditions between a pair of statements in may-alias. Each one of these systems is transformed to a disjunction of conjunctive systems describing all conditions for which at least one dependence is violated. Each of these systems describe required and sufficient conditions to detect dependence violation, meaning that, if a system evaluates to true for a given input, then there is at least one dependence violation and the optimization is incorrect. If all systems evaluates to false, then there are no violations, and the optimized code can be safely used.

The next section describes how quantifier elimination is used to reduce our run-time test evaluation costs, reducing from $O(n^2)$, where n is the number of iterations of our loop, to a $O(1)$ test. Such simplification produces systems that define required conditions for a dependence violation.

2.7 Simplifying the generated test

In our generated run-time tests, quantified variables require loops to explore all their possible values. Eliminating these quantifiers eliminates the need for loops, providing a massive gain in the run-time check speed. We developed a QEA that extends the FME technique. The first step to use FME requires our systems to be in conjunctive form, that is, we rewrite our test as a disjunction of conjunctive systems, given by the formula:

$$S_0 \vee S_1 \vee S_2 \cdots S_{n-1} \vee S_n$$

where each system S_i is in a conjunctive form of inequality constraints given by the formula

$$S_i = \{E_0 \geq 0 \wedge E_1 \geq 0 \wedge E_2 \geq 0 \cdots E_{m-1} \geq 0 \wedge E_m \geq 0\}$$

The FME process was developed to be used with linear systems of constraints in $\mathbb{R}$. Our systems hold multivariate polynomials over $\mathbb{Z}$, thus we face three difficulties: First, FME requires to determine coefficient signs of the variables being eliminated. Retrieving the sign of symbolic coefficient expressions is not a trivial task. The second problem rises from the fact that constraint limits on reals are looser than on integers. Loose limits generate rounding errors that accumulate at every step when eliminating variables. In practice obtained system using QEA on reals is usually so loose that the generated test forbids to chose the otherwise valid optimizing transformation. The third problem rises from the fact that FME was developed to be used with linear systems of inequalities. When applied over linear systems, every iteration of FME eliminates one quantifier. Our polynomial implementation might not eliminate a quantifier at a given step, if it would also be present in it's own coefficient. Naively eliminating constraints of lower degrees might generate unbounded variables leading to precision loss.

To tackle the first problem, our QEA depends on a positiveness expression test (which we call as *Schweighofer tester*), based on (Schweighofer, 2002, theorem 5). The Schweighofer tester is capable to determine if a given system of non-negative inequality constraints implies the positiveness of another expression. The Schweighofer tester is the core of our QEA and is used to detect variables coefficient signs, so that inequalities can be correctly positioned as lower or upper bounds for a the variable being eliminated. It also allows to decide correlations between expressions such that normalization (Pugh,

1991) techniques can be applied, removing rounding errors due to loose system boundaries. For last, we implement our FME in such a form that when a quantifier is not eliminated in a step of FME, we preserve constraints on that variable in a manner that we prevent unbounded variables.

Our proposed QEA processes each disjunctive system separately, eliminating variables and further removing redundant constraints generated by the FME process. Eventually, when all systems have been individually processes, the Schweighofer tester is again used to, between the resulting systems, eliminate those that impose redundant constraints. Our QEA for each system is divided in three main steps:

- **Increase precision**: During this process the Schweighofer tester is built for the given system. Constraints are tightened whenever possible. If new affine constraints can be detected, they are added to the system.
- **Variable elimination**: Say a quantified variable x is selected for elimination. If it is possible to determine the coefficient sign of x in every inequality of the system, then the standard FME step is applied. If not possible to determine the coefficient c of one of the inequality (say $cx \geq 0$), the system S is split into three systems, such that

  $$S \rightarrow S_{c\geq 1} \vee S_{c=0} \vee S_{c\leq -1},$$

  one where $c \geq 1$, one where $c = 0$ and another where $c \leq -1$. The three new systems are inserted to the conjunction of all other systems and processed individually.
- **Redundancy removal**: Once all variables of the simplified system S'_i have been eliminated, the Schweighofer tester is used to detect, withing S'_i, redundant constraints, where the least constraining ones are removed. Hence, once all n systems of

  $$S_0 \vee S_1 \vee S_2 \cdots S_{n-1} \vee S_n$$

  have been individual processed, generating

  $$S'_0 \vee S'_1 \vee S'_2 \cdots S'_{m-1} \vee S'_m$$

  simplified systems, the Schweighofer tester is again used to detect if

  $$S'_i \implies S'_j,$$

  removing S'_i as it defines a subspace of S'_j.

We now describe the most important components of our QEA. For further technical details please refer to §5.

2.7.1 Precision increase

Schweighofer tester

The first component of the proposed QEA is the Schweighofer tester, a tool used in almost every other procedure for different means, such as removing redundant constraints and redundant systems, determining expressions signs or expressions relationships. The core of this utility is the algorithm described by Schweighofer [2] to evaluate expressions signs under a system of constraints. The basic concept of it is the following: Given a system of inequalities

$$S = \{E_1 \geq 0, E_2 \geq 0 \ldots E_n \geq 0\},$$

[2] We actually use the technique in a more general context where we do not enforce variables to live in a compact polytope.

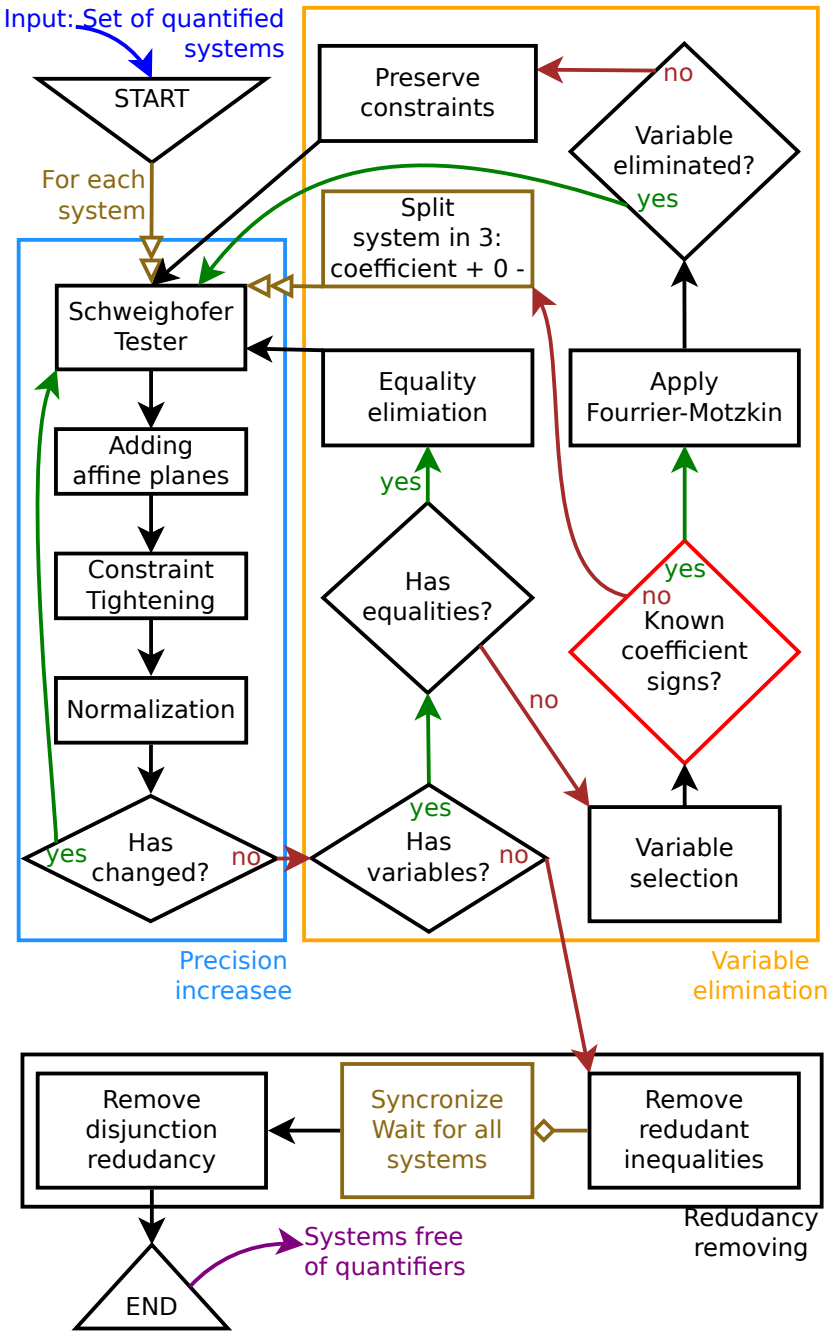

Figure 2.11: Quantifier Elimination Algorithm flow diagram. The main steps are divided in three phases: 1) precision increase 2) variable elimination 3) redundancy removal. The input is a set of quantified systems, and the output is a corresponding quantifier free set of systems.

Schweighofer provides an algorithm to evaluate if

$$S \implies E \geq 0.$$

If E can be written as the sum of the product of any power of the inequalities of S multiplied by non-negative factors, then S implies $E \geq 0$, that is

$$E = \sum \alpha_i \cdot E_i \text{ where } \alpha_i \geq 0, E_i \in S^D, D \in \mathbb{Z}^*.$$

However, applying such algorithm requires to solve a linear system of constraints with size exponential in the size of the expression $|S|^D$, where D is the maximum exponent S is raised to and $|S|$ is the cardinality of S. To tackle this exponential evaluation growth we empirically limited the maximum value of D. Instead of raising the entire system S to the power D, obtaining $|S|^D$ expressions, we use subset constraints $S' \subset S$, and we evaluate if any S'^D can be used to prove if $S' \implies E \geq 0$. We also perform expressions pattern matching to quickly detect obvious signs, such as positive powers that are always non-negative. We also perform expressions factorization and operator recursion to handle bigger expressions, as detailed in §5.

Adding affine planes

The cost of the Schweighofer tester algorithm grows with $|S|^D$, where $|S|$ is the number of inequalities in the system and D the maximum power we raise the system. Due its exponential behavior, we limit the maximum value D used to 2 or 3, depending on the number of constraints of S. Further, instead of combining all constraints of S to try to write the inequality we use distinct subsets $S' \subset S$, limiting the number of constraints of S' to four inequalities. To evaluate if $S \implies E \geq 0$ we evaluate if any of the subsets $S' \implies E \geq 0$.

Such limitations highly limits the capacity of the Schweighofer tester, as it is restricted to only consider a small set of inequalities at a time. To bypass this limitation a set of simple affine constraints (of the form $\pm i \pm j \leq c$), that the Schweighofer tester can prove to be implied by the system, is added to the system. Enriching the system allows to generate more subsystems S', thus increasing the capacity to write more expressions. The decision of increasing the number of constraints, over increasing the maximum exponent is a consequence of empirical observations:

1. Most observed systems have polynomials of absolute degree lower or equals to 3: Raising S to a high degree usually generates many unused inequalities.
2. Adding simple constraints, that are affine and are made up of one or two variables, allows the creation of almost all monomials up to an absolute degree of 4, when D is limited to 3.

For our use cases, it was observed that, the increased capacity of the Schweighofer tester for proving expressions signs reduced the overall execution time of the QEA. This is due to the fact that, as described in §2.7.2, when the coefficient of a variable being eliminated is unknown, the system resolution is split into the resolution of three systems, identical to the original one, but each with an additional constraint that defined the sign of the unknown coefficient. Improving the capacity of detecting signs also increased the quality of the final generated test.

Constraint tightening

This step relies on the Schweighofer tester to detect loose constraints and moves them to a position closer to the system boundaries. Reasoning geometrically, each inequality constraint $(E_n \geq 0) \in S$

defines a plane, or curve, dividing the space and limiting the system boundaries. If a plane is not touching the convex-hull formed by the remaining constraints it is considered loose. When the Schweighofer tester proves that a constraint is implied by other constraints of S, it also determines how loose this constraint is. In fact it gives a distance that can be used to displace this inequality without changing the set defined by the system. As discussed in more details in §5, the "distance" returned by the Schweighofer tester is *not* necessarily the maximum possible one (the tester should be expressed as a more costly linear programming problem, which is not the case in our implementation). As illustrated here, to circumvent this limitation, Schweighofer tester is heuristically invoked with a more constraining test.

In our implementation, verifying if

$$S \implies E_n \geq 0 \text{ where } [E_n \geq 0] \in S$$

will always detect that the distance of this plane has to the convex-hull is zero. To circumvent this problem, we request the Schweighofer tester to verify if

$$S \implies E_n - 1 \geq 0,$$

that is, if S implies a more constraining boundary. To exemplify this technique, let

$$S = \{x + 1 \geq 0 \wedge 2x - 6 \geq 0\},$$

where the first constraint is implied by the second. If the Schweighofer tester is requested to test if

$$S \implies x + 1 \geq 0,$$

it will respond with yes, by returning

$$x + 1 = 1 * (x + 1) + 0,$$

using the first found inequality. However, if requested to verify if

$$S \implies x \geq 0,$$

it would also say yes, but using the second inequality, returning

$$x = 0.5 * (2x - 6) + 3.\ ^{3}$$

This results describes that the system implies $x - 3 \geq 0$. As a consequence, this last inequality can safely replace the original $x + 1 \geq 0$ in the system.

Normalization

When using FME, we are manipulating integer constraints using reals algebra. As $\mathbb{Z} \subset \mathbb{R}$, constraints that do not define integer limits might be a source of precision loss. If this error is to high, it can generate a projection that represent a much bigger space than the actual one, generating quantifier free tests that never allow the use of the desired transformation. To understand the source of these errors, consider the following example:

$$i \leq 100t \leq 100j - 1.$$

The elimination of t, as explained below in §2.7.2, would lead to

$$i \leq 100j - 1.$$

[3] Observe that our systems being in $\mathbb{Z}$ do not restrain the Schweighofer tester to obtain only integer coefficients.

An increase in precision can be obtained through normalization (Pugh and Wonnacott, 1996; Pugh, 1991) of equalities and inequalities. In this example

$$100t \leq 100j - 1$$

can in fact be normalized into

$$t \leq j + \lfloor -1/100 \rfloor$$

becoming

$$t \leq j - 1.$$

After elimination of t, the system becomes

$$i \leq 100(j - 1).$$

Normalization can be generalized to systems of inequalities with multivariate polynomials, and constitutes an important step in making our technique efficient. Using the Schweighofer tester we extend normalization to allow simplification not only over constant, known, values, but over parameter expressions as well. Let $f(p), g(p)$ be two distinct expressions over parameters, such that

$$|f(p)| > |g(p)|.$$

Any inequality of the form

$$f(p) \cdot E \geq g(p)$$

can be normalized, but simply replacing this constraint by

$$E \geq \lceil g(p)/f(p) \rceil$$

could make the system loose the relationship information between $g(p)$ and $f(p)$. Also the sign and rounding direction depends on the signs of those same expressions. The symbolic normalization process reinserts the relationship constraint between $f(p)$ and $g(p)$, and replaces the original constraint, following the rules of Table 2.1.

		$f(p) \cdot E \geq g(p) \wedge \lvert f(p) \rvert > \lvert g(p) \rvert$	
		$f(p) > 0$ $\wedge$	$f(p) < 0$ $\wedge$
$g(p) \geq 0$	$\wedge$	$E \geq 1 \wedge f(p) - g(p) \geq 0$	$E \leq 0 \wedge -f(p) - g(p) \geq 0$
$g(p) < 0$	$\wedge$	$E \geq 0 \wedge f(p) + g(p) \geq 0$	$E \leq -1 \wedge -f(p) + g(p) \geq 0$

Table 2.1: Constraints that are added in the process of normalization of a parametric expression.

The first phase of QEA relies on the ability of the Schweighofer tester to detect loose or new constraints and iteratively increases the expressiveness and tightness of the system. It is executed until a *fixed point* is reached, where no existing constraints are proven loose. As there are no guarantees this process reaches a *fixed point*, it is limited to execute up to three times for each new quantifier being eliminated. By the end of this phase, loose constraints are expected to have been pruned as much as possible, making the system suitable for applying the core functionality of the QEA to lower quantifier degrees.

Case study

Here, we consider our running example so as to illustrate the process of precision tightening performed by our Schweighofer tester. In our `FW-tri` example, verifying the `WAR` dependence violation from v =`dist[j(j-1)/2+k]` and v' =`dist[i'(i'-1)/2+j']` is formulated as:

$$\exists Y \quad \text{s.t.} \quad F(X,Y) = 0 \wedge G(X,Y) \geq 0$$

where

$$\begin{aligned} X = \;& \{n\} \\ Y = \;& \{i,\, i',\, j,\, j',\, k,\, k'\} \\ F = \;& \{i'(i'-1) + 2j' - (j(j-1) + 2k)\} \\ G = \;& \{n-i-1,\, i-i',\, n-i'-1,\, k,\, i-k-2,\, n-k-3, \\ & i'-j'-1,\, j-k-1,\, j-j',\, i-j-1,\, k'-k-1, \\ & k',\, j'-k'-1,\, i'-k'-2,\, i-i'+k-k',\, n-k'-3\} \end{aligned}$$

First phase is to *add affines planes* (Octagon hull) to the system. Here, the Schweighofer tester is capable to decide that:

$$\begin{array}{lll} j' \geq k+2 & j \geq k+2 & i \geq k+4 \\ i' \geq k+3 & n \geq k+4 & j' \geq 2 \\ i \geq j'+2 & n \geq j'+2 & j \geq 1 \\ j \geq k'+1 & n \geq j+2 & k' \geq j \\ i \geq k'+3 & i \geq 2 & i \geq i'+1 \\ i' \geq 3 & n \geq i'+2 & n \geq 4 \end{array}$$

Then, each constraint is tightened when possible. 12 constraints could be tightened here, leading to replacing

$$\begin{array}{lll} k' \geq 0 & \text{by} & k' \geq 1 \\ i \geq i' & \text{by} & i \geq i'+1 \\ j \geq k+1 & \text{by} & j \geq k+2 \\ n \geq i'+1 & \text{by} & n \geq i'+2 \\ i \geq k+2 & \text{by} & i \geq k+4 \\ n \geq k'+3 & \text{by} & n \geq k'+4 \\ n \geq k+3 & \text{by} & n \geq k+5 \\ n \geq k+4 & \text{by} & n \geq k+5 \\ j \geq 1 & \text{by} & j \geq 2 \\ n \geq j'+2 & \text{by} & n \geq j'+3 \\ n \geq 4 & \text{by} & n \geq 5 \\ i \geq 2 & \text{by} & i \geq 4 \end{array}$$

2.7.2 Variable elimination

Equality elimination

In the process of eliminating variables, the first step is to check for the existence of equalities, as it can lead to a Gaussian elimination of quantifiers. In a given system, all inequalities of the form

$E \geq 0$ and $-E \geq 0$ for a same E are combined into an equality $E = 0$. If E has a quantifier without symbolic coefficient, and that exists in other expressions, this quantifier is replaced everywhere else, and the equality is removed.

Fourier-Motzkin elimination

The Fourier-Motzkin elimination is a well known algorithm for removing variables of linear inequality systems on $\mathbb{R}$. Now we demonstrate how this work extends FME to handle multivariate polynomials.

First let us described the classic FME algorithm over linear systems. Consider a system of inequalities over the set of variables

$$V \supset \{t\},$$

where t is a variable that we want to eliminate. Let

$$V' = V - t.$$

For every constraint with variable t, let us isolate it such that,

$$\begin{array}{rrll} & c_i \cdot t & \leq U_i(V') & \text{(upper bounds)} \\ \wedge \quad L_j(V') \leq & c_j \cdot t & & \text{(lower bounds)} \end{array}.$$

The inequalities *lower bounds* define the variables ts minimum value, just as the *upper bounds* define its maximum. c_i and c_j are non-zero positive coefficients of t. t is eliminated from the system by combining each of the lower-bound inequalities with each upper-bound on the form:

$$\forall (i,j),\ c_i \cdot L_j(V') \leq c_j \cdot U_i(V').$$

A simple example would be:

$$\begin{array}{rrl} & 5t & \leq y \\ x \leq & 3t & \end{array}$$

becoming:

$$5x \leq 3y$$

For linear systems, each step of FME eliminates one variable. The first main difference, when being applied over polynomial systems, is that each elimination step may not remove one variable. This can be observed from its formulation:

$$\begin{array}{rrl} & E_i(V) \cdot t & \leq U_i(V') \\ \wedge \quad L_j(V') \leq & E_j(V) \cdot t & \end{array}$$

L_j and U_j do not contain variable t, while the, positive, coefficients E_i, E_j might depend on t, but with one degree lower than the one in the original constraint. Writing inequalities in this form thus requires computing the signs of coefficients of t to correctly classify an inequality as an upper or lower bound. We use the Schweighofer tester to such end.

Our QEA first eliminates quantifiers that have only numeric coefficients, followed by those for which all coefficients can be determined. It eventually ends by eliminating those with unknown coefficient values. The affine constraints are eliminated first to quickly decrease the number of variables of the system and to possibly generate false constraints.

Quantifiers with symbolic coefficients but defined signs are eliminated next, hopping that, as the resulting systems has less variables, it might become possible to detect coefficient signs that were unknown. For last, for each unknown coefficient of the quantifier being eliminated, the system is split into three other ones as detailed above: Observe that for the case the coefficient of an inequality is zero (either from the Schweighofer tester or from the split process), the inequality is directly simplified.

This process always ends up with system(s) with known coefficients: The system can be put in "normal" form where the variable to be "eliminated" is completely isolated (without any coefficient). For our example this would give (assuming $E_i(V) > 0$ and $E_j(V) > 0$):

$$\begin{array}{rrcl} & & t & \leq \frac{U_i(V')}{E_i(V)} \\ \wedge & \frac{L_j(V')}{E_j(V)} & \leq & t \end{array}$$

Obtained system after elimination is simplified (whenever possible) using the greatest (found) common divisor of the coefficients $G(V) = \gcd(E_i(V),\ E_j(V))$. Assuming for convenience $G(V) > 0$ here:

$$L_j(V')\frac{E_i(V)}{G(V)} \leq U_i(V')\frac{E_j(V)}{G(V)}$$

At that point two remarks can be raised. This process generates many inequalities without necessarily eliminating t. It can even create expressions with higher degree than the original expressions.

1. For this reason, there is no guaranty of convergence.
2. There is no good reason to drop-off the original inequalities in t ($E_i(V) \cdot t \leq U_i(V')$ and $L_j(V') \leq E_j(V) \cdot t$ here). Dropping original inequalities would lead to loose of precision as illustrated below.

In the following example, dropping all the original inequalities would lead to a system that would hold unbounded variables, with solely upper or lower bounds. Consider the following system:

$$\begin{array}{rcl} 5 & \leq & t \\ & & t \cdot t \leq 4 \end{array}$$

after applying one step of FME the resulting systems would become:

$$5t \leq 4.$$

Now the variable t is unbounded... To avoid this possible loss of precision, all lower an upper constraints of the "eliminated" variable (say t) that have a degree in t not greater than the maximum degree (in t) of the generated system are re-injected. In our example, the new maximum degree of t in the generated system is one. Constraint $5 \leq t$ is re-injected. Constraint $t^2 \leq 4$ is dropped-off. Our resulting system becomes

$$\begin{array}{rcl} 5 & \leq & t \\ & & 5t \leq 4. \end{array}$$

Applying FME once more, the system results in

$$25 \leq 4,$$

which turns into false.

Observe that restricting to only re-inject expressions of degree not bigger than generated expressions does not solve the convergence problem. Indeed, as already mentioned, expressions generated by combining lower and upper bounds might have higher degree than original expressions. In our test cases, this process turned out to always converge in a few steps. However, in case it would not, one can enforce the degree to always decrease by writing the lower and upper inequalities in a slightly different way:

$$\begin{array}{rll} & E_i(V') \cdot t^{\alpha_i} & \leq U_i(V) \\ \wedge \quad L_j(V) \leq & E_j(V') \cdot t^{\alpha_j} & \end{array}$$

where $E_i(V')$ and $E_j(V')$ are expressions that do not depend upon t, and degree in t of $U_i(V)$ (resp. $U_j(V)$) is strictly less than α_i (resp. α_j). Despite is proven convergence, this scheme for elimination is not the default one. It is used as a backup in case the previous scheme blows out (which turned out to never happen in our experiments).

The overall framework might blow out at different levels:

- the number of systems can exponentially explode because of unknown coefficient signs
- the size of each system can exponentially explode with the number of steps in the elimination scheme

Despite its high cost, Schweighofer tester is always applied so as to avoid blowing out in terms of number of systems. It would have been possible not to test the sign, but eliminate absurd systems at the end of the elimination process. But: 1. it seems more reasonable to prune the exponential growth as soon as possible; 2. proving absurdity of systems with polynomials of high degree (degree increases with the elimination steps) is more difficult than with lower degree.

It is well known that if the size of a system in FME can in theory grow exponentially, in practice it contains a lot of redundancies that can be eliminated as the elimination process goes. Because, proving redundancies has a cost, there is a clear trade-off. In our case, only trivial redundancies (within the octagon domain) are removed during the elimination process. General redundancies, using Schweighofer tester are removed on the final system. Additional techniques, such as sign caching and early detection of false systems (see §5) also help to accelerate the projection process.

Case study

Let us consider the same system again. After eliminating quantifiers k, k', i, j' that held only numeric coefficients, our resulting system is:

$$\exists Y \quad \text{s.t.} \quad G(X,Y) \geq 0$$

where

$$\begin{aligned} X = \;& \{n\} \\ Y = \;& \{i',\ j\} \\ G = \;& \{-3+i',\ -1+j-i',\ -2+n-i', \\ & -10+4n+j-j^2-3i'+i'^2,\ -2+n-j, \\ & -7+2n-i',\ 3j-j^2-i'+i'^2, \\ & -2+j-j^2+i'+i'^2,\ -14+2n-j+j^2+i'-i'^2, \\ & -5+n,\ -10-j+j^2+3i'-i'^2,\ -6+j^2+i'-i'^2, \\ & -6+2n+j-j^2-i'+i'^2,\ -8+j+j^2+i'-i'^2, \\ & -12+2n+j+j^2-i'-i'^2,\ -2+j,\ -4-j+j^2+i'-i'^2, \\ & -4+2n+3j-j^2-3i'+i'^2,\ -7-j+j^2+2i'-i'^2, \\ & -18+4n-j+j^2-i'-i'^2,\ -4+n+j-i',\ -9+n-j+j^2+i'-i'^2\} \end{aligned}$$

When trying to "eliminate" j, the Schweighofer tester is capable to determine the signs of all symbolic coefficients:

$$\begin{aligned} -1+j &> 0 \\ 1+j &> 0 \\ j &> 0 \\ 1-j &< 0 \\ 3-j &< 0 \end{aligned}$$

All lower and upper bounds can be combined without having to split the system. Among all the combinations,

$$\begin{aligned} 6+i'+i'^2 &\leq j \cdot j \\ j \cdot (1-j) &\leq 6-2n-i'+i'^2 \end{aligned}$$

leads to $6+2nj-12j-i'+i'^2 \geq 0$ i.e. $2(n-6)j \geq i'-i'^2-6$. At this point, as j was not eliminated, all constraints that held j with degree 1 are maintained in the new system. At the next iteration of removing j, Schweighofer tester gives that $n \geq 6$. Thus the system is split in two, one where $n = 6$ and another where $n \geq 7$.

2.7.3 Redundancy removal

Simplification

Once all quantifiers have been eliminated, the next phase consists of simplifying, whenever possible, each of the constraints, followed by eliminating the redundancies. The simplification of constraints tries to reduce the degree of parameters on each constraint in two different manners: normalization, just as described in §2.7.1, and factorization.

Given one individual system of inequalities, if a constraint $E \geq 0$ of this system can be factorized as $E = \prod F_i \geq 0$, any factor F_i for which the strict sign can be proven by the remaining constraints of

the system, E is divided by F_i. This process is illustrated in the example below:

$$\begin{array}{rl} x & \leq -1 \\ y & \geq 1 \\ x \cdot E(x,y) & \geq 0 \qquad \rightarrow E(x,y) \leq 0 \end{array}$$

The second step consists of removing any redundant constraint of the system i.e. any constraint that the Schweighofer tester can verify as being implied by the remaining. The reasoning behind this step is:

$$\text{if } b \implies a \text{ then } a \wedge b \equiv b$$

A future simplification, still under development, consists in a statistical evaluation of parameter values collected from multiple executions of the instrumented application. Constraints that in most cases evaluate to true are removed.

Disjunction Simplification

Once all systems are projected we obtain systems solely over parameters. Many of those systems might be redundant, defining subset spaces of each other. Again, the Schweighofer tester plays an important role to remove redundancies and reduce the final test size. The projected systems lay in a disjunction, thus those that define subset systems of other ones, can be eliminated. Given systems S'_1, S'_2,

$$\text{if } S'_1 \subset S'_2 \text{ then } S'_1 \implies S'_2.$$

To detect such implication we verify if the system S'_1 implies all constraints of S'_2, in the form:

$$\left[\forall([E \geq 0] \in S'_2),\ S'_1 \implies E \geq 0\right] \equiv S'_1 \implies S'_2$$

If the implication is proven, then we can remove S'_1.

Comparison (both side of the implication) is done between pairs of simplified systems. A few techniques are applied to avoid comparing between all possible systems or inequalities:

1. If system S'_1 has a parameter not present in S'_2, then S'_2 can not imply S'_1.
2. As soon as it is detected that $S'_1 \implies S'_2$, S'_1 is removed to avoid further comparisons.
3. When testing if $S'_1 \implies S'_2$, if S'_2 contains the inequality $(I_2) : E \geq 0$ and S'_1 contains the inequality $(I_1) : E - c \geq 0$ (with $c \geq 0$), then (I_2) is implied by (I_1), and thus implied by S'_1. For example, $x + 1 \geq 0 \in S'_2$ is implied by $x \geq 0 \in S'_1$.

Test generation

For last our framework generates C code from the projected systems of constraints. The test is made up of inequalities between polynomials in parameters (including base pointer addresses). The overall framework, as described up to now assumes that all pointed memory variables have the same size and are aligned. Under this assumption, base pointers are cast to integers and divided by the type size. The more general case, that is not handled in the current implementation, should replace the equality of memory addresses (in the original system), by a non-overlapping test (two inequalities).

2.8 Current state

The different components of the tool chain were developed in parallel as proofs of concept prototypes, were we experimented: 1. different techniques to overcome precision loss, and processing costs of the quantifier elimination technique. 2. trace analysis and compression in the instrumentation and profiling part. The current state of the framework is still far from being user friendly or applicable to general loops as we intend it to be. The future work section in §7 describes what must be currently addressed such that our tools can be used by the public.

2.9 Summary

The proposed framework does not develop new transformations to better optimize loops, but extends the number of loops that can be safely optimized by existing compiler tools. Formally, our framework allows complex loop transformations even in the presence of may-dependencies. May-dependencies appear from the presence of memory accesses with polynomial expressions, or from memory pointers that may-alias. They impose two problems to optimizing tools: First, deciding what is a good transformation might be impossible statically in the presence of may-dependencies, as locality optimizations require data access patterns. Second, when validating that a transformation preserves semantics, compilers must be conservative and cannot not perform any rescheduling between instructions in may-dependence.

Our solution uses profiling to obtain dependencies observed during the execution of a instrumented version of an application. This information is used as an absolute truth to resolve may-dependencies and obtain profitable transformations from existing loop optimizers. Validity of the obtained transformations are dependent in how representative the observed dependencies are to all existing dependencies.

To ensure the applied transformation is valid, we build constraints that describe required and sufficient conditions for a dependence to be violated. For every pair of instructions in may-dependence, a disjunctive system of conjunctive constraints, inequalities, describe those conditions over loop induction variables and parameters. Those tests contains quantifiers, as many as twice the loop depth of the original code. They can easily (but not very efficiently) be evaluated at run-time with a cost quadratic in the size of the original iteration space.

Using a quantifier elimination technique, we simplify these tests by removing loop variants, such that the tests can be evaluated at run-time with a constant cost. To this end, we developed and implemented a new quantifier elimination algorithm that extends Fourier-Motzkin Elimination scheme so as to handle polynomial constraints over the integers. The Schweighofer tester is the most important component of the QEA, a powerful algorithm that allows to obtain symbolic expressions signs under the space defined by a system of inequality constraints. This technique allows the classification of inequalities as upper or lower bound constraints for a given variable being eliminated, required to apply FME. It is also used to detect relationship between expressions, what allows integer normalization techniques to rewrite loose constraints into tighter ones.

The next chapter, related works, describes other techniques developed to handle polynomial memory expressions. We also mention similar published works, which are focused to consider more restricted transformations. In existing contexts considered systems of constraints mostly consist in conditions that express the detection of memory regions overlapping. Further §4 provides results of our framework, demonstrating that it is capable to generate tighter tests than any other existing tool, allowing valid complex loop transformations to be safely used over code with polynomial memory access expressions and may-alias pointers. Following, *§5: On the details of the quantifier elimination* and *§6: On the details of the DDG* provide technical details of the framework before concluding with §7.

Chapter 3

Related work

3.1 Code optimization using quantifier-elimination

Work has already been done in attempting to use quantifier-elimination techniques to optimize loops with non-linear memory expressions. *The Challenges of Non-linear Parameters and Variables in Automatic Loop Parallelisation* (Grösslinger, 2009, Section 5.1) is the one that resembles the most to this work, as it uses quantifier elimination to obtain dependence validation expressions. In that work the authors state that, in the process of performing the dependence validation test, the most important step is to solve the memory access equality in the integers. From the possible integer solutions they use *QEPCAD-B* (Hong, 1992; Brown, 2003), a software that performs quantifier elimination on $\mathbb{R}$, to generate their test. The work presented here improves the test generation process in many different points such as allowing polynomial expressions as coefficient of existential quantifier. With a customized FME process we constantly realize normalization and affine convex hull detection, which helps in obtaining tighter tests. We also are capable of always eliminating *all* quantifiers, which allows to move our tests outside of the loops. Finally, their work was focused on allowing parametric parallelization, while our work allows any combination of affine transformations with tiling.

More recently, on the work *Fourier-Motzkin with Non-linear Symbolic Constant Coefficients* (Suriana, 2016), an extension to the FME was developed to handle parameter coefficients so as to produce loop boundaries. Limited to systems with a single symbolic coefficient, they perform normalization as described on (Pugh, 1991). In our work, we can also realize normalization over symbolic coefficients, as our Schweighofer tester can prove relationships between symbolic expressions. Without a technique to determine the sign of parametric coefficient they always evaluate all possibilities, except when determined in the environment cache, while our work, many times, can retrieve symbolic coefficient signs from implications of the existing constraints.

3.2 Extended data dependence analysis

There is a large variety of studies on extending data dependence analysis beyond the affine polyhedral model. In *Non-Linear Array Dependence Analysis* by Pugh and Wonnacott (1996), non-affine terms are treated as *uninterpreted function symbols*. A symbolic analysis combines and simplifies these function symbols, sometimes leading to run-time tests. This technique cannot eliminate quantifiers with

polynomial expressions and the generated tests would require loops to validate all possible values. For comparison our work is capable of generating run-time checks constant on the input data size.

In the *Hybrid Analysis: Static and Dynamic Memory Reference Analysis* by Rus et al. (2002), and later the *Logical Inference Techniques for Loop Parallelization* by Oancea and Rauchwerger (2012), a "uniform set representation" is used along with appropriate inference rules to reason about data dependencies. They also present an associated predicate language for efficient run-time tests.

The *A Unified Framework for Nonlinear Dependence Testing and Symbolic Analysis* by van Engelen et al. (2004) represents symbolic information in the form of chains of recurrence, and covers induction variable detection and substitution as well as dependence analysis as a variation of value-range analysis.

As mentioned later, all those works are actually restricted to proving non-aliasing of points-to sets, which is simpler than array dependence analysis.

3.3 Handling polynomials

Polynomials have long been recognized as an important class of functions for loop analysis. In *Symbolic Polynomial Maximization over Convex Sets and its Application to Memory Requirement Estimation*, by Clauss et al. (2009), Bernstein basis decomposition is used to perform maximization of multivariate polynomials in a given integer polyhedron. The authors use this to estimate applications memory usage. As described in *§7.1: Bernstein expansion:*, we intend to use this technique as a preliminary run-time test generation method, by determining if two instructions in may-dependence might access the same memory position inside the space where possible dependence violation might occur. If the results does detect that the distance is always different than zero, there is no need to use the costly QEA.

Recently, *The Power of Polynomials* by Feautrier (2015), has drawn attention to two polynomial positiveness theorems by Handelman and Schweighofer (2002). It also describes potential applications and the exponential cost growth, limiting their use. The process of the *Schweighofer tester* developed in this work is fundamentally based on it, and tries to address the question of usability of such complex algorithms for may-dependency resolution, such as using a random evaluation to define which sub-systems to use and how adding an affine convex-hull helps the positiveness test generation.

3.4 Array delinearization

Another topic that is related to polynomial memory access expressions is delinearization, where a set of loop bounds are used to reconstruct multi-dimensional arrays in order to obtain affine memory accesses from linearized data structures. The oldest approach for delinearization dates back to the work of *Delinearization: An Efficient Way to Break Multiloop Dependence Equations* by Maslov (1992), but recent advances have been made allowing access functions with parametric coefficients, as described in *On recovering multi-dimensional arrays in Polly* by Grosser et al. (2015a) and in *Optimistic Delinearization of Parametrically Sized Arrays* by Grosser et al. (2015c). Whereas delinearization consists of reconstructing affine array accesses to compute or apply a transformation, they work cannot handle cases with non-constant strides, such as linearized triangular matrices. They do also assume no aliasing between the base pointers and their run-time tests only validates the correctness of the predicted data structure. This work generates a run-time test that is specialized by the performed transformation, but does not have to assume that any base-pointers received as parameters do not alias.

3.5 Code versioning with run-time guards

Code versioning is not a new idea, by no means, and is now routinely used by compilers. GCC is capable of performing auto-vectorization (Nuzman et al., 2003) using aliasing and interval overlapping tests, as described by Bik (2004). Regarding aliasing tests like the ones produced by our technique, recent works include *Run-Time Pointer Disambiguation* by Alves et al. (2015), which uses run-time guards derived by static analysis to ensure non-overlapping of arrays. The analysis they use is restricted to affine accesses, but the same paper also presents a purely dynamic approach to disambiguate pointers. However, they cannot resolve dependencies between the same base pointer, which is important when applying automatic loop parallelization. Another point is that they generate strict disambiguation tests to allow the compiler to do more optimizations, but the applied transformation might require a much less restrictive test. The proposed work generates a complete data-dependence safety test, allowing automatic parallelization and vectorization to be applied. The test is particular to a given transformation and might be much less restrictive, thus being more successful in applying the transformation.

3.6 Hybrid, speculative, polyhedral optimization

In the work *Adapting the Polytope Model for Dynamic and Speculative Parallelization* by Jimborean (2012) (later improved by Sukumaran-Rajam (2015); Martinez Caamaño (2016), a predefined set of polyhedral transformations that expose thread level parallelism can be applied, speculatively and just-in-time, once linear memory accesses patterns are observed *during* execution. The idea is to instrument memory accesses during execution; if a linear pattern is observed, then an optimizing transformation is applied, expecting the access pattern to hold. Without guaranties that the memory access prediction would hold, a roll-back to a safe point must be done if a misprediction is detected. The main limitation of this approach is the overhead due to memory instrumentation and backtracking support. Our work based on hybrid analysis allows to avoid speculation as the validity of the transformation is checked before executing the optimized code in an inspector/executor manner.

Our work based on hybrid analysis allows to avoid speculation as the validity of the transformation is checked *before* executing the optimized code in an inspector/executor manner. In the approaches advocated by Venkat et al. (2014b), the inspector corresponds to loops that actually *inspect* the data (i.e. the test complexity is *not* $O(1)$). This approach is motivated as it focuses on indirect indexes through read-only indexing arrays.

In previous paragraphs, we mentioned works that already perform hybrid analysis with $O(1)$ run-time checks Alves et al. (2015); Kaplansky (2006); Rus et al. (2002); Größlinger (2009); Pugh and Wonnacott (1996), but they turn-out to be always restricted to testing non-overlapping of intervals. This is useful in testing if a loop is fully parallel (or could be vectorized) but cannot serve as testing the validity of arbitrary loop transformation (such as loop-interchange). Hence, Alves et al. (2015) only tackles the may-aliasing problem while GCC is capable of performing auto-vectorization Kaplansky (2006) using aliasing and interval overlap tests. Rus et al. Rus et al. (2002), and later Oancea and Rauchwerger Oancea and Rauchwerger (2012) use “uniform set representation” along with appropriate inference rules to reason about data dependence, and an associated predicate language for efficient run-time tests that validate loop parallelization.

Chapter 4

Experimental results

This chapter is dedicated to evaluating the proposed framework, first by comparing the proposed QEA against *QEPCAD Version B 1.69* (Brown, 2003) an publicly available [1] package to perform quantifier elimination. Using applications taken from the Polybench 4.1 benchmark suite, we demonstrate that our framework is precise, generating run-time tests that can actually allow optimizations to be used when valid, and that safely block invalid transformations. To exercise the polynomial capacities of our work, we linearized all evaluated applications, and particularly the *FW-tri* implements Floyd-Warshall minimal distance for symmetric weights, using a packed triangular matrix which generates memory accesses that are squares in the loop induction variables.

4.1 Test-bench

We used two different machine configurations. As every system of constraints can be processed independently of one another, the platform used to benchmark the quantifier elimination processes, i.e. for comparing our QEA against *QEPCAD-B*, has large parallel computation capacities. To measure the overhead of generated tests, a smaller platform was used, for minimizing time measurement variations due to clock frequency scaling and multiple-chips communication. The two experimental environments are:

Quantifier elimination: machine: uv2000.ujf-grenoble.fre (a.k.a idchire), 24 socked, 192 cores Intel Xeon E5-4640 @ 2.4GHz with a total of 768GB of memory running a 64-bit openSUSE Linux with kernel 3.0.101-0.47.71 and gcc 5.3.0. We measure the proposed QEA against *QEPCAD-B*, allowing it to use as much as 1073741822 memory cells, the higher amount we could allow without it failing due to allocation problems. Both solvers are set up with a time-limit of 60 minutes per system. We also provide preliminary results on using the polynomial lower and upper bounds technique implemented in *iscc*, which uses Bernstein expansion to describe polynomials.

Tests overhead: machine: dracula.inrialpes.fr, 4 cores Intel Core i5-4590 @ 3.3GHz with a total of 16GB of memory running a 64-bit Fedora 24 Linux with kernel 4.9. We compare binaries compiled

[1] https://www.usna.edu/CS/qepcadweb/B/QEPCAD.html

with with gcc[2] 6.3.1 and icc [3] Version 17.0.2.174 Build 20170213. We use Polybench 4.1 internal time measuring functions to determine kernels execution times. For a given benchmark we generate two different binaries:

1. Original/Reference: A simple linearized version of the application.
2. Versioned: Combine the generated test with the original and optimized code.

When compiling with *gcc* we use flags

```
-g0 -O3 -march=native -mtune=native -fopenmp.
```

When compiling with *icc* we use flags

```
-g0 -xHost -O3 -parallel -qopenmp.
```

Polybench flags used are:

```
-DPOLYBENCH_TIME -DPOLYBENCH_DUMP_ARRAYS -DDATA_TYPE_IS_DOUBLE
    -DPOLYBENCH_USE_C99_PROTO -UPOLYBENCH_USE_SCALAR_LB.
```

Displayed values are average over 20 executions.

4.2 Quantifier elimination

FW-tri: FW-tri implements a minimal distance algorithm between all nodes. Considering that distances are symmetric, that is, traveling between two positions hold the same cost in both directions, we can represent distances between two nodes only once. The implemented version uses the standard packed representation: a compact upper triangle representation that discards the diagonal, as we consider the distance from a node to itself being always zero.

The algorithm consists in four loop nests of depth 3 each, where one of them, as well the applied transformation to be validated, is presented below in Algorithm 9. The corresponding optimized code is shown further down in Algorithm 10. The applied transformation, that skews the outer loop and parallelizes the middle one, generates 385 dependence violation tests. Our QEA can statically prove that all these systems are false, allowing the transformation of all loops. For this example, as one can see on Table 4.1, *QEPCAD-B* failed to process any of the systems.

One of the systems, verifying the `WAR: dist[j(j-1)/2+k] = dist[i'(i'-1)/2+j']` is formulated as:

$$\exists Y \quad \text{s.t.} \quad F(X,Y) = 0 \wedge G(X,Y) \geq 0$$

where

$$\begin{aligned} X = &\ \{n\} \\ Y = &\ \{i,\ i',\ j,\ j',\ k,\ k'\} \\ F = &\ \{i'(i'-1) + 2j' - (j(j-1) + 2k)\} \\ G = &\ \{n-i-1,\ i-i',\ n-i'-1,\ k,\ i-k-2,\ n-k-3, \\ &\ i'-j'-1,\ j-k-1,\ j-j',\ i-j-1,\ k'-k-1, \\ &\ k',\ j'-k'-1,\ i'-k'-2,\ i-i'+k-k',\ n-k'-3\} \end{aligned}$$

[2] GCC, the GNU Compiler Collection
[3] Intel C/C++ compiler

```
void kernel_floydwarshall ( int n, double *dist ){
int i, j, k;
#pragma scop
for (k = 0; k < n - 2; k++)
 for (i = k+2; i < n; i++)
  for (j = k+1; j < i; j++)
   if(dist[i*(i-1)/2+j] > dist[i*(i-1)/2+k] + dist[j*(j-1)/2+k])
S1: dist[i*(i-1)/2+j] = dist[i*(i-1)/2+k] + dist[j*(j-1)/2+k];
#endscop
#pragma parallelize (k)
#pragma schedule S1 (k+i,i,j)
}
```

Algorithm 9: One of the four loops obtained from the FW-Tri. The `pragma` commands at the end describes the applied transformation. *aster* generates the dependence test from this description. The applied transformation performs an loop skewing of the external loop and parallelization of the second loop.

```
#pragma scop
if (n >= 3) {
 for (k = 2; k <= ((2 * n) + -4); k++) {
  lb1 = ((((k + 2)) > 0)? (1 + (((k + 2)) - 1)/(2)): (((k + 2)) / (2)));
  ub1 = ((k) < ((n + -1))? (k) : ((n + -1)));
#pragma omp parallel for private(j) firstprivate(lb1, ub1)
  for (i = lb1; i <= ub1; i++) {
#pragma ivdep
#pragma vector always
#pragma simd
  for (j = ((k -i) + 1); j <= (i + -1); j++) {
   if(dist[i*(i-1)/2+j] > dist[i*(i-1)/2+k-i)] + dist[j*(j-1)/2+k-i])
    S1:dist[i*(i-1)/2+j] = dist[i*(i-1)/2+k-i)] + dist[j*(j-1)/2+k-i];
  }
 }
}
```

Algorithm 10: The optimized code for FW-Tri.

Observe that the accesses have been multiplied by 2, as our system does not handle fractions, becoming a test for a WAR dependency: `dist[j(j-1)+2k] = dist[i'(i'-1)+2j']`.

Our experimental results for this particular system can be summarized in the following table:

	Time	Result	Result is tight
This work	240.7s	false	true
QEPCAD-B	-	Memory failure	-
iscc	6ms	$4+5n-n^2 \leq 0 \leq 8-5n+n^2 \wedge n \geq 5$	false

EX2: `Sub-stencil`: This stencil toy algorithm can be interpreted as a processing of a chunk of $m \times n$ pixels from an image that is H pixels wide. The original code is presented in Algorithm 11. The internal loop that iterates over i, performs stride-H accesses, and might be optimized by being interchanges with the outermost loop. The optimized code, shown in Algorithm 12, preforms stride-1 accesses, better exploiting spatial locality.

```
void kernel_sub-stencil (int H, int m, int n, double *t) {
 int i, j;
#pragma scop
 for (j = 1; j < m; j++)
  for (i = 1; i < n; i++)
   S1: t[i*H+j] = t[H*i-H+j] + t[i*H+j-1];
#pragma endscop
}
#pragma schedule S1 (i, j)
```

Algorithm 11: Stencil algorithm that works over sub-size chunks of length m of an H-pixels wide image.

```
void kernel_sub-stencil (int H, int m, int n, double *t) {
 int i, j;
 #pragma scop
 for (i = 1; i < n; i++)
  for (j = 1; j < m; j++)
   S1: t[i*H+j] = t[H*i-H+j] + t[i*H+j-1];
 #pragma endscop
}
```

Algorithm 12: Stencil algorithm optimized to better exploit better spatial data locality.

The system that describes the `WAW` may-dependence `t[i*H+j] = t[i'*H+j]` is:

$$\exists Y \quad F(X,Y) = 0 \wedge G(X,Y) \geq 0$$

where

$$\begin{aligned} X = &\ \{n,\ m,\ H\} \\ Y = &\ \{j,\ i,\ j',\ i'\} \\ F = &\ \{H(i-i') - j' + j\} \\ G = &\ \{-1+H,\ i-i',\ -1+j',\ -1+i',\ -1+m-j, \\ &\ -1+j,\ -1+j'-j,\ -1+i-i',\ -1+i, \\ &\ -1-i'+n,\ -1-i+n,\ -1+m-j'\} \end{aligned}$$

Our experimental results for this particular system can be summarized in the following table:

	Time	Result	Result is tight
This work	40.8s	$m \geq H+2 \wedge n \geq 3 \wedge m + 2H(n-3) \geq 3 \wedge m + Hn \geq 4H+2$	true
QEPCAD-B	> 60min	Timed out	-
iscc	7ms	$2 + H - m \leq 0 \leq (n-2)H - 1 \wedge n \geq 3 \wedge m \geq 3$	true

In that case, both our solver and iscc can find that the transformation is valid if $m < H+2$. This test represents a correct limitation, that avoids distinct loop iterations to access overlapping memory regions.

Table 4.1 summarizes a quantitative comparison of the presented QEA to QEPCAD-B for projecting may-dependence systems. Considered systems have been generated based on the transformations applied to Polybench 4.1, as specified on Table 4.2. We observe that our solver manages to process all the systems while QEPCAD-B did not even manage to solve 1% of the systems. Among the processed systems, our solver was tight enough to allow the optimized transformation to be chosen at run-time. Focusing on the 0.5% of the systems that QEPCAD-B managed to process, and generously assuming all the other could have been processed, lack of tightness for QEPCAD-B would lead the generated run-time test not to chose the optimized transformed version for four of the benchmarks (correlation, covariance, FW, gemver).

Application	**Quantified test**				**Presented QE**					**QEPCAD**						
name	#var	#par	#ineq	#sys	#par	#ineq	#sys	Tig	avg.t	#proc	Tig	avg.t	#to	#ie	#fpe	#sf
2mm	4	9	4080	265	9	3435	318	318	224.57	17	17	17.03	61	6	7	174
3mm	6	12	3277	212	12	1892	224	224	176.78	14	14	21.02	64	1	5	128
atax	4	6	382	34	6	73	17	17	39.66	21	21	12.26	1	1	0	11
bicg	4	7	524	45	7	110	27	27	40.10	30	30	13.63	5	0	0	10
cholesky	6	2	408	28	0	1	1	1	5.49	23	23	50.30	2	0	0	3
correlation	6	6	1835	148	6	560	70	70	76.28	59	55	49.33	15	1	7	66
covariance	6	5	1057	77	5	704	41	41	189.64	26	22	234.57	24	0	2	25
doitgen	8	6	287	13	6	11	2	2	257.43	5	5	617.61	3	0	1	4
FW	6	2	475	25	0	1	1	1	66.23	17	9	142.95	1	0	0	7
gemm	6	6	722	45	6	269	34	34	195.89	10	10	21.09	9	1	1	24
gemver	4	10	1886	155	9	89	30	30	31.92	141	137	170.33	3	0	2	9
gesummv	4	6	521	48	6	74	25	25	25.36	48	48	82.31	0	0	0	0
gramschmidt	6	5	91	7	5	49	11	11	52.15	2	2	6.35	1	0	0	4
heat-3d	8	4	208560	8064	0	1	1	1	325.54	-	-	-	-	-	-	-
jacobi-2d	4	3	528	44	3	103	37	37	47.63	5	5	600.96	13	2	0	24
lu	6	2	376	22	0	1	1	1	0.88	7	7	65.07	5	0	0	10
mvt	4	6	534	46	6	79	27	27	28.53	46	46	8.06	0	0	0	0
seidel-2d	4	4	544	40	0	2	2	2	23.90	1	1	492.19	0	0	0	39
symm	6	5	411	26	5	178	28	28	171.34	1	1	6.09	8	0	1	16
syr2k	6	5	1211	74	5	523	89	89	171.51	10	10	6.60	26	1	3	34
syrk	6	4	722	45	4	316	34	34	271.21	8	8	1189.12	14	0	0	23
trisolv	4	4	41	4	0	1	1	1	0.46	4	4	118.96	0	0	0	0
trmm	6	4	628	38	4	2015	111	111	317.44	4	4	45.67	10	1	2	21
FW-tri	6	2	7240	385	0	1	1	1	1088.33	0	0	N/A	90	0	29	266

Table 4.1: Table comparing results tests generation using the implemented QE and QEPCAD-B. From left to right: Application name; On the **Quantified test** section, the **Comp**lexity order of the test (in the input size), the total number of **Ineq**alites in the test, and the number of **Sys**tems (conjunctions) in the test; on the **Presented QE** section also presents how many of the simplified systems we obtain that are **Tight** enough to allow the optimization to be executed. On the section **QEPCAD** section is described the number of systems that: could be **Proc**essed, within those how many are **Tight** and their **average** processing **time**, in seconds; The number of systems that **T**imed **O**ut in a time limit of 1h or that finished with **i**nternal **e**rror, **f**loating **p**oint **e**xception or **s**egmentation **f**ault;

	gcc					icc								
	ori.ginal		opt.imized			ori.ginal			opt.imized					
	time (ms)	GF/s	time (ms)	GF/s	Speed -up	time (ms)	GF/s	icc / gcc	time (ms)	GF/s	Speed -up	icc / gcc	best opt / best ori	Transf
2mm	21.45	2.04	4.38	10.00	4.90	16.07	2.72	1.33	4.39	9.97	3.66	1.00	3.67	psv
3mm	32.95	1.38	4.16	10.93	7.92	23.20	1.96	1.42	2.77	16.43	8.38	1.50	8.38	fFipv
atax	0.35	1.81	0.11	6.00	3.31	0.08	7.85	4.34	0.11	5.89	0.75	0.98	0.76	fipv
bicg	0.53	1.21	0.11	5.92	4.89	0.53	1.22	1.01	0.11	5.83	4.78	0.98	4.86	Fpv
cholesky	15.09	4.24	14.83	4.31	1.02	10.01	6.39	1.51	10.03	6.38	1.00	1.48	1.00	v
correlation	11.93	1.29	6.86	2.25	1.74	10.14	1.52	1.18	6.43	2.40	1.58	1.07	1.58	fipsv
covariance	11.89	1.27	1.23	12.30	9.68	10.10	1.49	1.18	0.90	16.87	11.28	1.37	11.28	fipv
doitgen	8.00	1.80	1.60	9.00	5.00	5.95	2.42	1.34	1.23	11.71	4.84	1.30	4.84	fv
FW	82.53	4.54	36.93	10.14	2.23	82.58	4.54	1.00	52.91	7.08	1.56	0.70	2.23	pt
gemm	3.38	9.32	0.86	36.46	3.91	2.36	13.34	1.43	0.62	50.93	3.82	1.40	3.82	Fipv
gemver	0.60	2.70	0.30	5.44	2.01	0.40	4.05	1.50	0.16	10.39	2.56	1.91	2.56	Fipv
gesummv	0.21	1.18	0.05	4.86	4.12	0.21	1.18	1.00	0.01	19.08	16.19	3.93	16.17	Fpv
gramschmidt	18.98	1.02	6.55	2.95	2.90	17.75	1.09	1.07	6.22	3.11	2.85	1.05	2.85	Fpv
heat-3d	16.28	11.81	6.94	27.72	2.35	19.21	10.01	0.85	4.18	46.00	4.60	1.66	3.89	ipsv
jacobi-2d	7.74	8.03	5.47	11.35	1.41	17.59	3.53	0.44	4.72	13.15	3.72	1.16	1.64	ptv
lu	30.40	2.11	18.46	3.48	1.65	23.41	2.74	1.30	14.48	4.43	1.62	1.27	1.62	v
mvt	0.51	1.17	0.14	4.14	3.54	0.31	1.94	1.65	0.16	3.78	1.95	0.91	2.14	Fpv
seidel-2d	166.13	0.87	136.51	1.06	1.22	180.16	0.80	0.92	147.73	0.98	1.22	0.92	1.22	Fpv
symm	13.28	2.20	13.31	2.19	1.00	10.87	2.69	1.22	10.78	2.71	1.01	1.23	1.01	fFipv
syr2k	19.80	1.74	3.46	9.96	5.72	17.03	2.02	1.16	4.27	8.07	3.99	0.81	4.92	ipv
syrk	4.58	3.77	3.63	4.75	1.26	2.87	6.01	1.59	0.54	32.11	5.35	6.76	5.35	Fpv
trisolv	0.13	1.20	0.11	1.44	1.20	0.10	1.63	1.36	0.13	1.25	0.77	0.87	0.89	t
trmm	6.45	1.79	1.66	6.95	3.88	5.48	2.11	1.18	1.32	8.71	4.14	1.25	4.14	fipsv
FW-tri	93.33	2.20	34.03	6.03	2.74	98.95	2.08	0.94	25.67	8.00	104.91	1.33	3.64	ptv

Table 4.2: Table with applied transformations and speed-up per benchmark. Columns description, from left to right: Application name; **Ori**ginal and **Opt**imized execution times (in ms) and performance (in GF/s) and optimization speed-up when compiling with gcc. To the right we have the same comparisons when compiling with icc, and performance comparison of icc and gcc for each original and optimized binary. The two last columns stands for the best optimized version (gcc/icc) speed-up against the fastest original (gcc/icc) version and the combined transformations to obtain the optimized version, where each letter stands for: Loop **F**ission, Loop **f**usion, Loop **i**nterchange, Thread-level **p**arallelization, Loop **s**kewing, **v**ectorization.

Chapter 5

On the details of the quantifier elimination

Quantifier Elimination is an important class of mathematical algorithms applied in several fields of research, such as physics, logic, computer science and math and "is a standard way of proving the decidability of first order theories" (Cohen and Mahboubi, 2010). A possible interpretation is to represent a boolean formula $F(x, y)$ as a region (set of points) in a space $\mathcal{X} \times \mathcal{Y}$. In the following derived formula with quantifier, $\exists y \in \mathcal{Y} \,/\, F(x, y)$, the quantifier in variable y could be eliminated leading to a formula $F'(x)$ where variable y does not appear anymore. This "simplification" can be viewed as a projection of that region to space $\mathcal{X}$. If, for example, $\{x \in \mathcal{X} \,/\, F'(x)\} = \emptyset$, then $\{(x, y) \in \mathcal{X} \times \mathcal{Y} \,/\, F(x, y)\} = \emptyset$. QE methods can be interpreted as a projection algorithm as depicted in Figure 5.1.

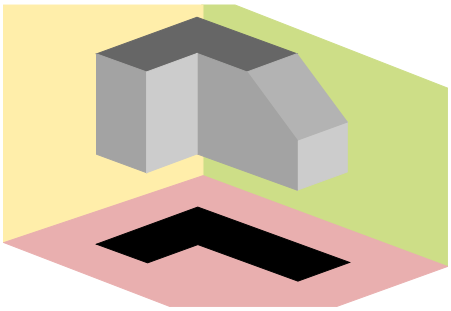

Figure 5.1: Projection of an 3-dimension shape into a 2-dimension one. In the context of this work, the 3D polyhedron represents the space where one dependence is violated, and the 2D shape represents the test we evaluate at run-time to decide if it is safe to use the optimized code, that is, if the current inputs define a point outside the projected shape, it is safe to use the transformed code.

With a resolution cost that is double exponential on the number of variables to eliminate (Davenport and Heintz, 1988), many research has been pursued to efficiently implement quantifier elimination. However, no previous work focus on how to produce tight solutions for integer multivariate polynomial constraints. Applying techniques directed solely to real systems could produce loose constraints due to rounding errors, as described in *§2.7.1: Normalization*.

Let us use an toy example, used by Hong and Din (2012) to demonstrate this error:

$$\exists Y \quad F(X,Y) = 0 \ \wedge \ G(X,Y) > 0$$

where

$$\begin{aligned} X &= \{x\} \\ Y &= \{y_1,\ y_2\} \\ F &= \{y_1^2 + y_2^2 - 1\} \\ G &= \{y_1^2 x - (y_2 - 1)^2\} \end{aligned}$$

After eliminating Y by using an implementation of QEA for $\mathbb{R}$, such as QEPCAD-B (Brown, 2003), the obtained quantified free formula obtained holds that $x > 0$ (which is tight in $\mathbb{R}$). However, reasoning over the $\mathbb{Z}$ one could identify that the original condition F has only four possible solutions of (y_1, y_2), respectively $(0,-1), (0,1), (-1,0)$, and $(1,0)$. Filling G with such values, the three first directly evaluate to false, and the last one turns into $x > 1$ i.e. $x \geq 2$. Although small in this example, such errors can accumulate through each elimination step, generating a very loose final projection. Figure 5.2 depicts a scenario where a single loose constraint differentiates between a system that is empty or not.

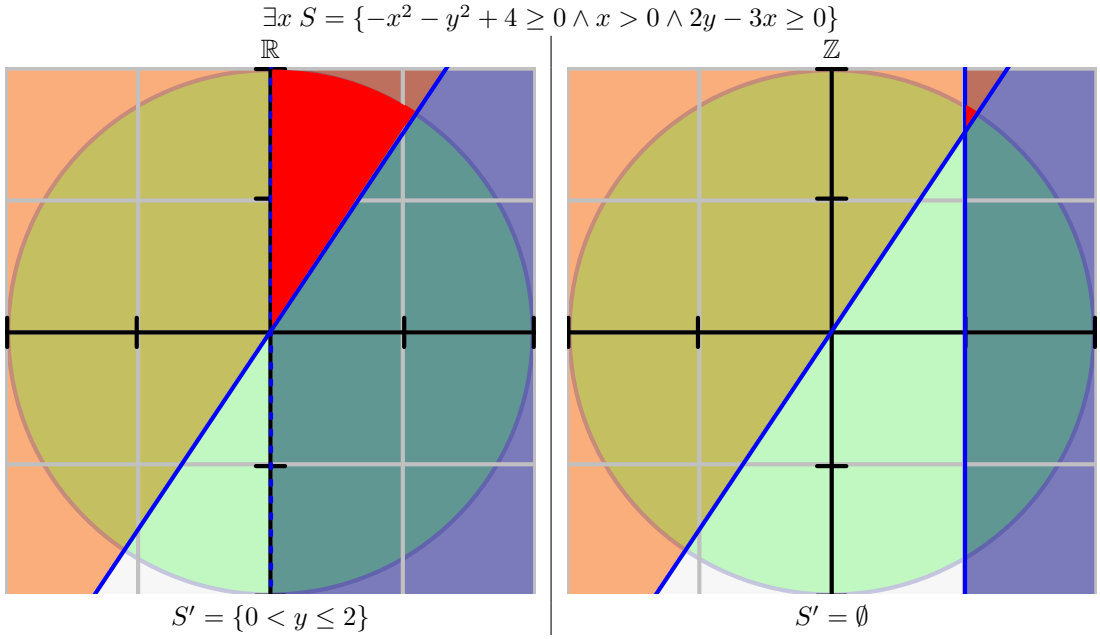

Figure 5.2: Non-inclusive constraints, such as $x > 0$, impose loose constraint in $\mathbb{Z}$. Naively projecting the system as constraints in $\mathbb{R}$, without removing this error would project the red shaded area of the left side, resulting in a non empty S'. However, correcting $x > 0$ to $x \geq 1$, the projected area in red of the rightside holds no integer points, thus the system is empty.

Our Quantifier Elimination Algorithm (QEA) is based on the very well known Fourier-Motzkin elimination (FME) process, enriched with techniques to handle symbolic coefficients, polynomials, and improvement of rounding errors. The core elements require the Schweighofer tester and are described in Section 2.7. Here we provide some additional tricks used to speedup or enhance the Schweighofer tester capacity to determine expression signs and how early detection of false systems is done.

5.1 Sign detection

Performing sign detection can use the Schweighofer tester: Schweighofer tester allows to check whether or not the current system S implies a non-negative inequality $E \geq 0$. Algorithm 13 shows how determination of sign and possible nullity of E is performed, using up to implication tests.

```
sign(ex E, system S){
        bool gtz = gez = lez = ltz = false
        gtz = S.implies(E >= 1)
        gez = gtz || S.implies(E >= 0)
        ltz = S.implies(-E >= 1)
        lez = ltz || S.implies(-E >= 0)
        if(gez && lez){
                if(gtz || ltz)
                        return ABSURD
                return ZERO
        }
        if(gez){
                if(gtz)
                        return GTZ
                return GEZ
        }
        if(lez){
                if(ltz)
                        return LTZ
                return LEZ
        }
        return UNKNOWN
}
```

Algorithm 13: How Schweighofer tester detects the sign of expression E in a system of constraints S. `ZERO`, `GTZ`, `GEZ`, `LTZ`, `LEZ`, `UNKNOWN`, `ABSURD` stand for the detected sign of E being, respectively: zero, strictly positive, non-negative, strictly negative, non-positive, could not resolve, absurd. Both zero and absurd are found by detecting that both positive and negative expressions are implied, but in the absurd one of them is strictly non-zero.

5.2 false detection

Deciding, only after all quantifiers are eliminated, if a given system has no solution might be a difficult task, as generated constraints might be polynomials of high degree. The presented QEA tries to early detect false systems in two different manners:

Proving $-1 \geq 0$: Every time a new Schweighofer tester is built over a system S, it verifies if $S \implies -1 \geq 0$ (by e.g. setting "E" to -1). In case of success, this contradiction proves that S is false, and the system is invalidated.

Proving $E \wedge \neg E$: Every time the Schweighofer tester is used to determine the sign of an expression E under a system S, it persistently tries to detect both if $S \implies E \geq 0$ and if $S \implies E \leq 0$. In the case where both are implied, and at least one is proved to be exclusively non-zero, then E holds an `ABSURD` sign, and the system is detected as false.

5.3 Expression factorization

Whenever the sign of an expression E can't be resolved directly by the Schweighofer tester, a recursive technique tries to factorize E and each operand sign is evaluated separately. To avoid testing the same expressions multiple times, a sign cache is created, and whenever an expression is evaluated once, the returning sign is cached and not evaluated again. When factorized, these algebraic rules are applied:

Even powers are always non-negative ($(E')^{2n} \geq 0$): If the expression being tested is an exponent of even power, it is first verified if the base has known sign. If it does and is not equal to zero, it is returned as strictly positive. Otherwise, it is returned as greater or equals to zero.

Odd powers hold the same sign of the base ($\textbf{sign}(E')^{2n+1} \equiv \textbf{sign}(E')$): If the expression being tested is an exponent of even power and the sign of the base is known, then the same sign is returned.

Sum sign rules If an expression E is a sum of expressions $E' + E''$ the rules when combining the sub-expressions signs are:

1. `ABSURD` prevails over any other sign.
2. `UNKNOWN` prevails over the remaining signs.
3. `ZERO` sign does not change the other sign.
4. If signs are in different directions, the result is `UNKNOWN`.
5. If both E' and E'' hold the same sign, the sum is the same.
6. Strict signs prevail over non-strict.

Multiplication sign rules If an expression E is a multiplication of expressions $E' \cdot E''$ the rules when combining the sub-expressions signs are:

1. `ABSURD` prevails over any other signs.
2. `ZERO` and right after `UNKNOWN` prevail over all other signs.
3. Non-strict signs prevail over strict signs (`GEZ`, `LEZ`).
4. Non-positive and negative inverts the sign (`LEZ`, `LTZ`).

5.4 Equality normalization

The last algebraic card in the sleeve of the QEA is to tighten equalities by applying the same techniques of normalization that we used for inequalities (see *§2.7.1: Normalization*). As an example, let P be the set of parameters of the program, and V the set of variables. If an equality can be written as:

$$f(P) \cdot g(P, V) = r(P, V)$$

If it is possible to prove that:

$$|f(P)| > |r(P, V)|$$

then

$$g(P, V) = r(P, V) = 0$$

5.5 "A little randomness is always good"

Our implementation of the proposed algorithmic approach to *schmüudgen's positivstellensatz* (Schweighofer, 2002), that is highly used for retrieving coefficients signs, imposes important cost challenges. To make this clearer, let us use an example where we want to find if

$$S = \bigwedge \begin{cases} i \geq i' + 1 \\ j' \geq j + 1 \\ m \geq j' + 1 \\ m \geq j + 1 \\ n \geq i' + 1 \\ n \geq i + 1 \\ j' \geq 1 \\ j \geq 1 \\ i' \geq 1 \\ i \geq 1 \\ H \geq 0 \\ H(i - 1 - i') + j - j' = 0 \end{cases} \quad \text{implies } \; i \geq 0.$$

The first step done by the Schweighofer tester is to replace equalities by two equivalent inequalities, such as $E = 0$ becomes $E \geq 0 \wedge E \leq 0$; rewrite all inequalities as $E \geq 0$; and expand all expressions to

obtain sums of monomials, such that the system becomes:

$$S = \bigwedge \begin{cases} i - i' - 1 \\ j' - j - 1 \\ m - j' - 1 \\ m - j - 1 \\ n - i' - 1 \\ n - i - 1 \\ j' - 1 \\ j - 1 \\ i' - 1 \\ i - 1 \\ H \\ Hi - H - Hi' + j - j' \\ -Hi + H + Hi' - j + j' \end{cases} \geq 0.$$

The test now becomes a linear algebra problem, where we want to write the tested inequality $i \geq 0$ as a non-negative sum, of S to any power, that means

$$i = \alpha_0 + \sum_{n=1}^{n=|S^d|} \alpha_n \cdot E'_n \text{ where } E' \in S^d, d \in \mathbb{Z}_{\geq 0}, n = |S^d|, \alpha_{0\cdots n} \in \mathbb{R}_{\geq 0},$$

in words, if i can be written by the sum of any set of inequalities from S raised to any power, then S implies $i \geq \alpha_0$. Let's assume the case where $d = 1$, then we must solve:

$$i = \sum \begin{cases} \alpha_0 \\ \alpha_1(i - i' - 1) \\ \alpha_2(j' - j - 1) \\ \alpha_3(m - j' - 1) \\ \alpha_4(m - j - 1) \\ \alpha_5(n - i' - 1) \\ \alpha_6(n - i - 1) \\ \alpha_7(j' - 1) \\ \alpha_8(j - 1) \\ \alpha_9(i' - 1) \\ \alpha_{10}(i - 1) \\ \alpha_{11}(H) \\ \alpha_{12}(Hi - H - Hi' + j - j') \\ \alpha_{13}(-Hi + H + Hi' - j + j') \end{cases} \quad \text{where } \alpha_{0\cdots 13} \geq 0.$$

Considering each distinct monomial as a variable, this can be solved by a linear solution $\{\alpha\} \times S = i$:

i		α_0	α_1	α_2	α_3	α_4	α_5	α_6	α_7	α_8	α_9	α_{10}	α_{11}	α_{12}	α_{13}
0	1	1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	0	0
0	H	0	0	0	0	0	0	0	0	0	0	0	1	-1	1
0	Hi'	0	0	0	0	0	0	0	0	0	0	0	0	-1	1
1	i	0	1	0	0	0	0	-1	0	0	0	1	0	0	0
0	Hi	0	0	0	0	0	0	0	0	0	0	0	0	1	-1
0	i'	0	-1	0	0	0	-1	0	0	0	1	0	0	0	0
0	j	0	0	-1	0	-1	0	0	0	1	0	0	0	1	-1
0	j'	0	0	1	-1	0	0	0	1	0	0	0	0	-1	1
0	m	0	0	0	1	1	0	0	0	0	0	0	0	0	0
0	n	0	0	0	0	0	1	1	0	0	0	0	0	0	0

and a solution such as $\alpha_{10} = 1,\ \alpha_0 = 1$ (and all other α set to 0) would allow to conclude that $S \Rightarrow i \geq 1$. However, obtaining nice solutions, with all α numerically defined and positive is not easy (see below). For our given example, the non-numerical solution obtained using our underlying math library would be:

$$
\begin{aligned}
\alpha_0 &= 2.0 + \alpha_8 + \alpha_6 - \alpha_{10} - \alpha_4 \\
\alpha_1 &= 1.0 + \alpha_6 - \alpha_{10} \\
\alpha_2 &= \alpha_8 - \alpha_4 \\
\alpha_3 &= -\alpha_4 \\
\alpha_4 &= \alpha_4 \\
\alpha_5 &= -\alpha_6 \\
\alpha_6 &= \alpha_6 \\
\alpha_7 &= -\alpha_8 \\
\alpha_8 &= \alpha_8 \\
\alpha_9 &= 1.0 - \alpha_{10} \\
\alpha_{10} &= \alpha_{10} \\
\alpha_{11} &= 0 \\
\alpha_{12} &= \alpha_{13} \\
\alpha_{13} &= \alpha_{13}
\end{aligned}
$$

which, by interactively replacing unconstrained values (corresponding to taking an arbitrary value of the system and ignoring the $\{\alpha\} \geq 0$ constraints), such as $\alpha_{13} = \alpha_{13}$, by zero, we obtain:

$$
\begin{aligned}
\alpha_0 &= 2.0 \\
\alpha_1 &= 1.0 \\
\alpha_2 &= 0 \\
\alpha_3 &= 0 \\
\alpha_4 &= 0 \\
\alpha_5 &= 0 \\
\alpha_6 &= 0 \\
\alpha_7 &= 0 \\
\alpha_8 &= 0 \\
\alpha_9 &= 1.0 \\
\alpha_{10} &= 0 \\
\alpha_{11} &= 0 \\
\alpha_{12} &= 0 \\
\alpha_{13} &= 0
\end{aligned}
$$

which for last proves it is implied. On the current implementation one problem is that the replacement by zero might invalidate an undetermined answer, as one of the resulting expressions can be

$$\alpha_x = \alpha_y + \alpha_z - 1.0 \wedge \alpha_y = \alpha_y \wedge \alpha_z = \alpha_z$$

and the final solution would be invalidated after replacing α_y, α_z by zero. As briefly discussed in §7, a better (but more expensive) solution would express the problem as a linear programming problem: add the $\{\alpha\} \geq 0$ linear constraints to the system and maximize α_0.

A second problem lies in the fact that systems size to be solved grows exponentially with d. In our example, when d is set to 2, distinct monomials count grows from 10 to 80, and 14 to 170 expressions. Empirically, it was observed that most implications were solved by combination of up to 3 expressions and that most of our problems would not have a polynomial of degree higher than 3. From the latter observation, an empirical limit of the maximum power of S was imposed. From the former observation, it was decided to build subsets of system, with up to three inequalities, and evaluate if at least one of these subsets would imply the inequality being tested. However, the total number of possible subsets is combinatorial, and testing each one of ended taking more time than the original evaluation. The problem now became, how to select those promising subsets to test?

Let $f(X)$ and $g(X)$ be two inequalities where $f(X) \Rightarrow g(X)$. Then, for any value of X, if $g(X)$ is false, then necessarily $f(X)$ is also false. Suppose $g(X)$ corresponds to the positiveness of $E(X)$ and $f(X)$ to a subsystem. Consider a random value to be assigned to the system variables (X here). If g, i.e. the inequality being tested, evaluates to false with that value X, *all* subsystems that evaluate to true are discarded as they *cannot* involve g (thus cannot be used to prove g). The overall process iterates on d with a time-out for each iteration. It starts with all subsystems of S^1 and tries to slim it using many generated random values. Denoting the slimmed version of S^1, S', it then considers S'^2 and tries to slim this new bigger system with the same mechanism. Say we obtain S''. It finally slims $S'' \times S$ and stops.

Chapter 6

On the details of the DDG

Recent works (Fauzia et al., 2013; Elango et al., 2015a,b) use DDG graphs to characterize the data-movement complexity of a program. Performing transformations that preserve dependencies, they reschedule the original program execution order, and try to identify code regions that hold potential for optimizations with respect to data locality. Such works, used as performance debugging, might generate schedules that cannot be converted back into structured loop-oriented code. Different from those, this work uses DDG to resolve may-dependence information, required by loop optimizers to retrieve profitable code changes.

If applied naively using a shadow memory on any scalar/memory variable, the constructed DDG might be polluted by many control/data dependencies stemming from iterators. As an example, enforcing to follow the dependencies related to the scan of the iteration space (that uses loop index increments and jumps) would impose exactly the same execution order as the instrumented program. A simple but too approximate approach used by Fauzia et al. (2013) to get around this problem, consists in instrumenting only floating-point variables and ignoring all dependencies related to integer arithmetic. Our solution is more robust and simply consists in ignoring any instruction for which the `SCEV`/ analysis (of LLVM) provides a closed static formula, dependent solely on canonical induction variables and loop invariant values.

Another possible problem when working with DDG graphs is related with the cost of manipulating such graphs. As each node in the graph represents an dynamic instruction, applications that execute for just a tenth of a second might hold millions of dynamic instructions, making their analyses costly in memory and time. As our work is focused on optimizing loop nests, we exploit the fact that in most cases, dependencies observed in just a chunk of consecutive loop iterations are representative to the entire iteration domain. At that point, one should outline that our approach does not *impose* the DDG to be accurate. As the program behavior (and the observed dependencies) might already depend on the input data, focusing on chunks of consecutive loop iterations, instead of considering the whole loop, is yet another approximation. The challenge is to find tricks (such as the clamping technique described further) that makes the profiling as representative as possible with the important trade-off in mind: tightness versus size/cost. So we limit the number of iterations a loop is tracked, from the time it is entered. Using an environment variable to control the number of traced instructions, we can easily build graphs with representative dependence information and limited number of nodes. This iteration limit can be disable by defining the limit to zero. The applied *cut-off* technique, that just ignores instructions outside the monitoring limits, might too optimistically loose important loop intra-iterations dependencies, that would prevent some transformations, such as *loops interchange*. To still build a small graph but preserving such

constraining dependencies this work proposes a *clamping* technique, where the otherwise ignored memory accesses from consecutive iterations of a loop are instead clamped into a single virtual instruction, representing a single node on the DDG.

Loop optimizers, such as *PLuTo* (Bondhugula et al., 2008), require data producer and consumer information not only to guarantee that a given transformation is valid, but also to improve locality by scheduling close together instructions that use the same data (temporal locality). Our work allows to enhance the DDG with data reuse edges. If one wants to extend the shadow memory so as to online-build `RAR` vectors, this will turn-out to be extremely expensive as it would require to store *all* the consumers of a given storage location in the shadow. Alternatively, we chose to build `RAW` and `WAW` information using the standard technique and offline-compute all the `WAR` and `RAR` hyper-edges by applying a simple transitive closure.

Technical details on how we build the DDG and how we use results is presented below.

6.1 Technical details

It is important to highlight that the implementation of the process of generating the DDG, and using it to retrieve profitable transformations is an ongoing work. It falls into four steps: instrumentation, tracing (or run-time), trace analysis (or simulation) and optimization. The first three steps can be executed in an semi-automated fashion while the modeling is still in preliminary stages. Semi-automated meaning that the user must define the region of code to be instrumented.

Instrumentation: Our instrumentation is done at the LLVM IR level. Function calls to our tracing library are inserted in the program without changing its semantics.

Defining the instrumented region: Currently the instrumented region must be manually defined by the user by inserting a call to

```
ddg_trace_start()
```

at the beginning of the region of interest and, at the end, inserting a call to

```
ddg_trace_stop().
```

During the instrumentation process, each instruction, basic block and loop receive a unique identifier, in a deterministic manner. This is important to map information in the trace file back into the code. List of function calls and their meanings will be explained ahead in this chapter.

Instrumenting the code: The first step of the instrumentation process generates an LLVM IR source code to be used as reference. From this file the instrumented version is generated, using the *opt* command from the LLVM infrastructure:

```
opt -S -load ddg.so -ddg-instrument original.ll -o instrumented.ll
```

Run-Time: Two different C/C++ run-time libraries that have the same interface are presented. The first, called *cut run-time library* (CRL), implements the *cut-off* technique, that stops tracing after a given number of loop iterations. The second one, the *full run-time library* (FLR), traces the entire execution, to either produce a complete or clamped DDG, where the clamping process is done during the simulation step.

Once instrumented, compiled and linked to our run-time library, executing the generated binary will produce an trace file (that can be piped to the next phase) called

```
ddg.trace
```

containing sequences of execution of basic-blocks, induction variables values, and accesses memory positions at instruction level. The environment variable

```
DDG_LIMIT
```

defines size of traced chunks within each loop dimension.

Simulation: The produced trace file contains basic-blocks execution order. To retrieve instruction level dependencies our framework simulates the execution of every basic-block in the order they are found in the trace file: Memory accesses are reported at the instruction level and the correctness of the simulation is guaranteed by matching the instruction identifier in both the reference code and the trace file.

Dependencies between memory accesses are built during this phase, using the shadow memory technique as described in section §2.3. Observe that there would have been no particular difficulties in integrating the construction of the DDG in LLVM itself, thus merging it with the run-time. This design choice (that can be discussed) just allows to decouple all the developments related to the DDG construction (shadowing, clamping, folding, etc.) to the compilation of the program. As for the interface, the simulation phase uses the *opt* command from LLVM to load our analysis:

```
opt -load ddg.so -ddg-cdag .
```

6.2 Traced Events

This section will describe, for each different traced events:

- What is done during instrumentation
- The information stored in the trace file
- How the simulator acts when handling an event of that kind

6.2.1 `ddg_start_trace`

This defines the start of an instrumented region.

Instrumentation: Manually inserted by user, this function call defines a point of entering a given region of interest.

Run-Time: Opens the trace file in append mode and clears the run-time state structures.

Simulation: The iteration vector is cleared, as are any other state structures, including the shadow memory. It is expected that the next operation is either a basic block execution or a loop entry.

6.2.2 ddg_alive_in

Instrumentation: When a call to `ddg_start_trace` is detected, every variable that is alive during this point of the program has its identifier and value stored in the trace file using an `ddg_alive_in` function call.

Run-Time: Add alive-in token to the trace file, followed by variable identifier and value.

Simulation: Obtain live-in information, including loop invariant values, such as memory base pointers.

6.2.3 ddg_basic_block

A basic block defines the smallest sequence of operations that necessarily execute in sequence. Our instrumentation gives, in a deterministic manner, a unique identifier to each basic block of the original program.

Instrumentation: At the beginning of every basic block, that is not a loop header, insert a call to `basic_block` that provides the unique static block identifier.

Run-Time: Add the basic block token to the trace and its corresponding identifier. The CRL only acts if the tracking flag is true.

Simulation: For each basic-block token seen in the trace (identified thanks to its static identifier), its list of instructions is traversed. Special treatment is necessary for ϕ-functions as only one of the operands (the one corresponding to the incoming edge) is used at a time. Identifier of the basic-block is kept so as to find the matching operand. As any ϕ-function has the semantic of a register-to-register copy, simple dynamic copy-propagation is (optionally) performed. As an example, assume the ϕ-function $x_1 = \phi(x_2, x_3)$ is processed and the used operand is identified to be x_3, then the shadow value of x_3 that corresponds to the producers' identifier of x_3 (instruction id & iteration vector) is copied to the shadow of x_1. For instructions that do more than just a copy, a new node corresponding to the produced data is created. The current iteration vector is used to stamp the newly created node. The shadow memory is updated with the id of this new node. Note that memory load/store instructions are treated differently.

6.2.4 ddg_load/ddg_store

Just as with basic-blocks, every instruction of the original program receives a unique, deterministic, identifier that is used to map information from the trace file to the source-code, to the DDG graph and, for last, into the AST model of the program.

Instrumentation: Every load and store instructions is prefixed by a function call with the unique instruction identifier and the memory address variable.

Run-Time: Adds a read/write token and the received/stored memory position.

Simulation: When traversing a basic-block, whenever a `load`/`store` instruction is reached, the simulator reads the trace file to verify the existence of that memory operation, comparing the token type and instruction identifier from the reference source-file and the trace file. The accessed memory position, obtained from the trace file, is used to populate the shadow memory (Nethercote and Seward, 2007) to determine dependencies.

The shadow memory technique consists in a one-to-one map of all accessed memory positions and the last dynamic instruction that wrote to that memory position. Any instruction that loads a mapped memory position is `RAW` dependent on the instruction that wrote it, and an instruction that rewrites an memory position is `WAW` dependent on the previous writer. To also obtain reuse information, that is `RAR` edges between dynamic instructions, our implemented shadow memory also keeps a list of all dynamic instructions that read a memory position before it is rewritten by a store operation.

6.2.5 `ddg_loop_begin/ddg_loop_end`

The $\vec{iv}$, that defines an loop instance, has length equals to the depth of the loop being executed. The `ddg_loop_begin` and `ddg_loop_end` events are responsible to control this depth.

Instrumentation: Every loop header is split in two basic blocks, right at its beginning. The fist executing one defines a loop pre-header, an single entry point to the loop that is outside the loop it self. It receives all possible entry edges, except the fallback edge, that defines an loop iteration, that goes into the second basic block. At the end of the pre-header the `ddg_loop_begin` function call is inserted registering the entrance to the loop. In the same fashion all exit edges of the loop point to a single created footer, holding the `ddg_loop_end` function call.

Run-Time: A new position in the iteration vector is pushed/popped. The CRL only adds the event if in a traced region.

Simulation: It pushes a new position onto the back of the iteration vector.

6.2.6 `ddg_loop_iteration`

Instrumentation: Every loop header has an associated function call inserted at the beginning of the header basic-block. The loop identifier is the same as its header basic-block. LLVM IR loop nest information does not detect irreducible loops, so we do ignore such cases.

Run-Time: Stores the current loop identifier and trip counts. The CRL also verifies if the current trip counts exceeds the tracking limit, and updates the iteration vector as well the active tracking flag.

Simulation: Updates the iteration vector vector values and traverses the loop header basic block just as a `ddg_basic_block`.

6.2.7 `stop_trace`

Defines the end of an instrumented region.

Instrumentation: This function is inserted into the source code of the original program by the user.

Run-Time: Writes finish token, closes trace file and clears iteration vector.

Simulation: Verifies that the iteration vector is zeroed, and generates the output files, if requested.

The resulting DDG from which we extract dependence vectors can be exported for the user to view in Graphviz dot format. Figures 6.2 and 6.3 depicted respectively, graphs build with the *cut-off* and *clamping* techniques, for the `dspmv` application, limiting in four the number of loop iterations to be traced. The same graph can be exported in text format, depicted in Figure 6.1, to be used with the debugging tools provides by Elango et al. (2015a).

```
432
0: 1 #(1,0,1,0,0,0,0);1;/
1: 6 16 26 36 #(1,0,2,0,0,0,0);2;+
2: 8 #(1,0,3,0,0,0,0)
3: 4 #(1,0,3,0,2,0,0);17;LD
4: 10 #(1,0,3,0,3,0,0);18;*
5: 7 #(1,0,3,0,4,0,0);19;LD
6: 7 #(1,0,3,0,5,0,0);20;LD
7: 8 #(1,0,3,0,6,0,0);21;*
8: 9 #(1,0,3,0,7,0,0);22;+
9: 10 #(1,0,3,0,8,0,0);23;*
10: 11 #(1,0,3,0,9,0,0);24;+
11: 49 #(1,0,3,0,10,0,0);25;ST
12: 18 #(1,0,3,0,11,0,0)
13: 14 #(1,0,3,1,2,0,0);17;LD
14: 20 #(1,0,3,1,3,0,0);18;*
15: 17 #(1,0,3,1,4,0,0);19;LD
16: 17 #(1,0,3,1,5,0,0);20;LD
17: 18 #(1,0,3,1,6,0,0);21;*
18: 19 #(1,0,3,1,7,0,0);22;+
19: 20 #(1,0,3,1,8,0,0);23;*
20: 21 #(1,0,3,1,9,0,0);24;+
21: 70 #(1,0,3,1,10,0,0);25;ST
22: 28 #(1,0,3,1,11,0,0)
23: 24 #(1,0,3,2,2,0,0);17;LD
24: 30 #(1,0,3,2,3,0,0);18;*
25: 27 #(1,0,3,2,4,0,0);19;LD
26: 27 #(1,0,3,2,5,0,0);20;LD
27: 28 #(1,0,3,2,6,0,0);21;*
28: 29 #(1,0,3,2,7,0,0);22;+
29: 30 #(1,0,3,2,8,0,0);23;*
30: 31 #(1,0,3,2,9,0,0);24;+
31: 91 #(1,0,3,2,10,0,0);25;ST
32: 38 #(1,0,3,1,11,0,0)
33: 34 #(1,0,3,3,2,0,0);17;LD
34: 40 #(1,0,3,3,3,0,0);18;*
35: 37 #(1,0,3,3,4,0,0);19;LD
36: 37 #(1,0,3,3,5,0,0);20;LD
37: 38 #(1,0,3,3,6,0,0);21;*
38: 39 #(1,0,3,3,7,0,0);22;+
39: 40 #(1,0,3,3,8,0,0);23;*
40: 41 #(1,0,3,3,9,0,0);24;+
41: 112 #(1,0,3,3,10,0,0);25;ST
42: 43 #(1,1,1,0,0,0,0);1;/
43: 59 80 101 122 #(1,1,2,0,0,0,0);2;+
...
```

Figure 6.1: Textual DDG representation to be used as input to performance debugging tool. The first line holds the number of nodes of the graph, and each other line holds: node id: list of consumer nodes#$\vec{iv}$;static instruction identifier;operation type.

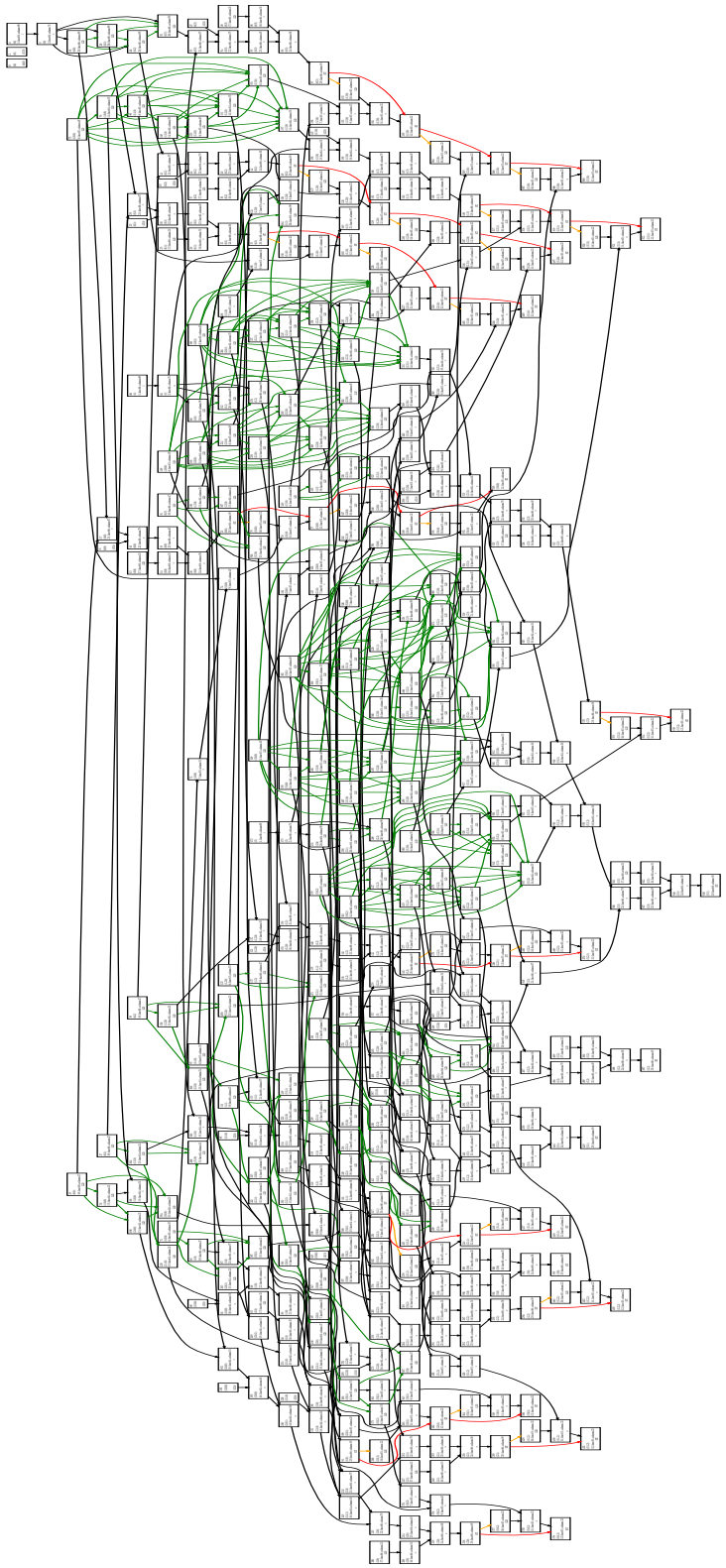

Figure 6.2: Example of complete graph generated with tracing up to 4 iterations per loop, for the `dspmv` example. Observe that many leaf nodes do not generate dependencies.

Figure 6.3: Example of complete graph generated with clamping after 4 iterations per loop, for the `dspmv` example. A virtual node has dependencies with all, previously leaf, nodes.

6.3 Modeling

For last, our framework builds an AST of the instrumented region, mapping observed dynamic dependencies to the static statements of this tree. Figure 2.6 depicts such tree in an graphical manner and Figure 2.5 in text format. Dependencies between two statements can be represented in three different levels of precision:

Iteration precise: Every dynamic dependence observed is preserved, represented by the tuple

$$\{id_s, id_c, \vec{iv}_s, \vec{iv}_d\},$$

where id_s, id_c are the source and destination statement static identifiers, and $\vec{iv}_s, \vec{iv}_d$ are the respective source and destination iteration vectors.

Distinct distance vectors: Preserves every distinct distance vector, represented by the tuple

$$\{id_s, id_c, \vec{iv}_d - \vec{iv}_s\}.$$

Symbolic distance vector: An unique symbolic distance vector describes all dependence vectors observed between two instructions. Represented by the tuple

$$\{id_s, id_c, \Delta(\vec{iv}_d - \vec{iv}_s)\},$$

where $\Delta(\vec{iv}_d - \vec{iv}_s)$ is computed by combining distance vectors in an element wise fashion, following the lattice shown in Figure 6.4.

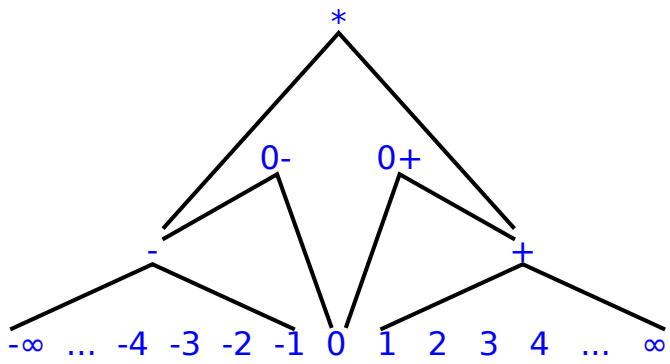

Figure 6.4: Lattice to combine distance vector, in an element wise fashion.

Chapter 7

Conclusion

When dependence analysis fail to provide precise information, it prejudices the compilers to retrieve profitable code transformations or even to ensure that a given, valid, optimization preserves syntax. May-alias base pointers and memory accesses described by polynomial expressions over loop induction variables are two factors that limit existing dependence analysis precision addressed by this work. May-alias conditions appear from unrestrained memory pointers received as function arguments. Memory accesses using polynomial expressions over loop induction variables can appear as result of compilers linearization process of data structures or use of data symmetries to preserve memory, such as symmetric matrices used in LAPACK (www.netlib.org, 1992).

By performing memory accesses profiling the proposed framework uses dependencies observed at run-time to resolve dependencies not solved statically. Using Scalar Evolution (Pop et al., 2005) analyses, spurious dependencies are ignoring computations of values that described statically by a closed formula. To avoid the expensive costs of analyzing all dynamic instructions, techniques such as cut-off and clamping limit the number of loop iterations analyzed. Cut-off, and optimistic approach, ignores dependencies occurred outside the observed space while clamping preserves such dependencies but with a much lighter, and imprecise, method that compresses instructions outside the observed space into single fake instruction. Observed dependencies are used as ground truth information when retrieving loop transformations. Once retrieved a new loop schedule the framework builds a test that defines sufficient, and required, conditions for which the retrieve transformation is **invalid**.

Such test consists in validating, for every pair of distinct dynamic instructions in may-dependence, if they both accesses the same memory position and if their original relative execution order, is not preserved when applied the optimization. Evaluated at run-time the generated test would only return true if at least one dependence is violated. However, the cost of evaluating such test grows quadratically in the number of dynamic instructions in the transformed region. This means that evaluating such test would, probably, take a longer time than just executing the original program without any optimization. To transform such complex test into a much lighter one the proposed work uses a quantifier elimination scheme.

The generated test consists in systems of equalities and inequalities over loop induction variables and memory base pointers. Induction variables, coming from dynamic instruction identifiers (the induction vectors), define existential variables for which all possible values must be evaluated. Using quantifier elimination these existential variables are eliminated generating systems that can be evaluated in $O(1)$, in regard of the number of dynamic instructions, but might be more constraining than the original test. Based on Fourier-Motzking Elimination (FME), a quantifier elimination technique developed for linear systems of inequalities over the $\mathbb{R}$, new extensions were generated for handling multivariate polynomials

on the $\mathbb{Z}$. Writing all expressions as non-negative inequalities, the Schweighofer tester implements an positiveness test algorithm based on Schweighofer's work (Schweighofer, 2002, Theorem 5?) and is the core tool of the proposed simplifier. The Schweighofer tester allows detection of coefficient signs, required to classify if a given inequality defines an upper or lower bound, prerequisite to perform FME. It allows extending normalization (Pugh, 1991) to symbolic expressions, minimizing rounding errors of integer systems. Although the test is evaluated in $O(1)$ in regard of dynamic instructions, the number of expressions inside each system, validating a may-dependence violation among two statements, might grow exponentially with the number of variables eliminated. The number of systems to be evaluated might also grow with the number of statements in may-dependence. Once again the Schweighofer tester plays an important role in simplifying the system, by allowing to detect and remove redundant constraints from within each system, and redundant systems among all systems. Compared to existing quantifier elimination tools, such as QEPCAD-B (Brown, 2003), our quantifier elimination technique demonstrates competitive time and memory costs and yields preciser results, required to successfully allow execution of the optimized code.

For last the simplified test guards the execution of the optimized region. If evaluated, at run-time, to false, the test directs to the execution of the optimized code and directs to the original, not optimized code otherwise. We demonstrated that such tests impose very low overhead, even with very small load works, that take less then a second to execute. Our work allows *any affine rescheduling* of loop regions, as far all memory accesses in the optimized region contain a static closed formula. Different than works that only perform may-alias overlapping tests (Rus et al., 2002; Kaplansky, 2006; Größlinger, 2009; Alves et al., 2015) or speculative systems that impose costly and complex run-time validations (Jimborean, 2012; Sukumaran-Rajam et al., 2014). The soundness and precision of the proposed technique is demonstrated over 24 kernels retrieved from the Polybench 4.1 (Pouchet and Yuki, 2015) and even in code with packed triangular matrices, such as the implemented triangular Floyd-Warshall, for which lightweight and precise tests where generated.

The presented framework demonstrates promising results when it comes to validate complex loop transformations in code with poor dependence information, obtaining less constraining and still valid. Still under development the framework holds many room from improvements in test precision and execution time cost.

7.1 Future work

Although the existing framework presents promising results, some factor currently limit its use: The amount of time to generate run-time tests and to simplify them are in the range of minutes, when dealing with small kernels, making it impossible to use with larger code. The process is not yet automated, requiring human intervention to handle non-affine code. The validation test is simplified regardless if the simplified test is too constraining and would always prevent the optimization to be used. The current implementation cannot handle code with function calls or double indirection. Addressing these flaws are crucial to turn this framework competitive with existing techniques. Some guides on how to do so are provided below:

Bernstein expansion: By using Bernstein expansion it is possible to bound the range of a multivariate polynomial over a parametric polyhedra, as described by Phillipe Clauss on (Clauss and Tchoupaeva, 2004). On that work their motivation was also dependence resolution among memory accesses described by polynomial expressions. Just as in our systems that describe may-dependencies violations, to define the dependence actually exists the memory positions accesses by two instructions in may-dependence

must be the same, such as $f(i) = g(i')$. Rewriting it as $f(i) - g(i') = 0$, the left side expressions, that might be a polynomial, must be equals to zero in order to the dependence exist. Whenever occurs that the $lowerbound(f(i) - g(i')) > 0$ or that the $upperbound(f(i) - g(i')) < 0$, it means that these two instructions do not access the same memory position, thus there is no dependence, much less an dependence violation. If these values cannot be retrieved statically, they can be evaluated during execution, serving as guards to using, or not, the optimized code.

Barvinok (Verdoolaege et al., 2007), a library for counting the number of integer points in parametric and non-parametric polytopes, implements such technique. Using the associated tool, *iscc*, this method demonstrated very promising results: In 385 may-dependence verification systems for the Triangular Floyd-Warshall benchmark it only failed to provide tight results in eight systems.

```
p1:=[n] -> {[k, j', j, k', i, i'] -> -2 * j + 2 * k'+i-i^2-i'+i'^2 :
0 <= k <= -3 + n and j > k and j >= j' and k' > k and
0 <= k' < j' and k' <= -3 + n and i >= 2 + k and
j < i < n and i' > j' and 2 + k' <= i' < i and
i' < k - k' + i and i' < n};
lb(p1);
([n] -> { min((10 + n - n^2)) : n >= 6 }, True)
ub(p1);
([n] -> { max(-16) : n >= 6 }, True)

p2:[n] -> { [k, j', j, k', i, i'] -> -2*k+2*j'+j-j^2-i'+i'^2:
0 <= k <= -3 + n and j > k and k' > k and
0 <= k' < j' and k' <= -3 + n and i >= 2 + k and
j < i < n and i' > j' and 2 + k' <= i' < i and
i' < k - k' + i and i' < n};
lb(p2);
([n] -> { min((4 + 5 * n - n^2)) : n >= 6 }, True)
ub(p2);
([n] -> { max((4 - 5 * n + n^2)) : n >= 6 }, True)
```

Figure 7.1: Use of *iscc* to retrieve conditions for which a dependence violation exists for the Tri-FW benchmark. On the first example the polynomial $-2j + 2k' + i - i^2 - i' + i'^2$ describe the distance of the two memory access positions. It is possible to statically define the system is empty, such that this dependence does not exists, as the maximum value (upper bound) is -16. On the second system, evaluating memory distances described by $-2k+2j'+j-j^2-i'+i'^2$, the lower bound value is negative and the upper bound is positive, meaning that it could exist an integer solution where the polynomial would evaluate to zero, meaning the two instructions accesses the same memory position, thus having a dependence violation. Observe the polyhedron defines the region in space where the relative execution order between two instructions is may dependence is not preserved.

Use of faster tools: Currently the framework always uses the presented FME scheme for whatever system, affine or not, although more fit tools cold be used. Another fact is that empty systems might be detected using simpler approximations of the invalid polyhedron. We propose an sequence of validations and tools, ordered by complexity:

1. **Bounding box / affine convex hull:** False systems might be easily detected if the bounding box or its affine convex hull does not hold any integer solution, what can be quickly detected using the *Integer Set Library* (Verdoolaege, 2010).

2. **Affine systems using existing FME:** Affine systems can use existing FME implementation in the ISL. If the systems holds polynomial constraints FME can be applied to its affine convex hull. The result can be evaluated against profiled values to determinate if it is tight enough.

3. **Bernstein expansion:** For systems defined by affine constraints, and a single polynomial equality, the technique described on the previous item can be used to retrieve parametric conditions to be used as the transformation validations. Systems containing multiple polynomial equalities can be transformed into multiple systems with a single polynomial equality. The obtained conditions can be evaluated using profiled values to determine if they are tight enough.

4. **Faster quantifier elimination:** For systems with polynomial constraints faster quantifier elimination implementations, such as Redlog (Dolzmann and Sturm, 1997; WEISPFENNING, 1997), might be capable to detect systems that are empty even when working over constraints in $\mathbb{R}$.

Improve sign detection: Many can be done to speed-up or increase the capacity of the Schweighofer tester to decide upon expressions signs. Such improvements might increase the number of coefficient signs detected, reducing the number of systems to be evaluated, possible also reducing the final test size and increasing their precision, as normalization highly depends on the Schweighofer tester.

1. **Bernstein expansion:** Using only the affine constraints of an system to build a polyhedron, polynomial expressions that are exclusively positive, or negative, in such polyhedron hold the same sign in the original one. Once again, the Bernstein technique could be used to retrieve polynomial bounds.

2. **Static Schweighofer tester:** Our current heuristic to determinate which sub-set from a system of constraints should be used to build the Schweighofer tester if very costly, requiring to evaluate a huge number of systems. Perhaps using the original system and eliminating constraints that are unique in holding monomials not present in the evaluated expression and that cannot be further generated by higher order combination might be sufficient to avoid memory explosions.

3. **More flexible linear solver:** Our current Schweighofer tester implementation uses the linear solver delivered in the GiNaC library, however this solver does not allow imposing positiveness constraints, and many the algorithm failed to retrieve existing implications due the solver returning solutions with negative factors.

Profile guided QE: This work only focused on complete elimination of quantifiers in test expressions. Perhaps, using run-time information, one could validate that eliminating a certain variable might generate a test that always fails, and try to eliminate other ones, or even generate a test with loops. The problem with these tests containing loops is that validating that the complexity of the generated guard does not outweigh the optimization gains. For last, profiled values can be used to simplify the final test by removing constraints that, in multiple executions traces, always evaluate to true.

Interleaved memory accesses: Can interleaved memory accesses, such as vectors of multi-element structures or two independent sub-images of a bigger picture, be represented in a form that quantifier elimination can preserve, which would help in validating distinct interleaved accesses? Perhaps by determining expressions limited by modular values. For example, an matrix $F[NxN]$, and two sub-matrices $A[MxM]$ and $B[LxL]$. If the accesses to sub-matrices A and B module N would give non-overlapping sets, it means that A and B do not overlap.

Code with function calls: This work does not allow function calls inside the optimized region, since it does not know how to fit them into an affine program representation to be given to a polyhedral optimizer. However, one might think on functions as statements in an outer, "global" loop. Each time a function call or return executes, it is as if a new iteration of this external loop had started. All the functions can then be thought of as residing in a global `if-then-else` chain where each element is assigned a unique time dimension id. In such a representation, all canonical iteration vector would have two more fields in front of them. The first would hold how many times function calls and return instructions where executed. The second holds a unique identifier per function.

Inspector / executor: Currently the framework only supports code with a single layer of indirection, such as `A[`$f(\vec{iv})$`]` += $\cdots$, where $f(\vec{iv})$ is an statically known function over canonical induction variables. It can be extended to handle memory accesses with double indirection, such as `A[B[`$f(\vec{iv})$`]]` += $\cdots$, an inspector / executor loop can verify that `B[`$f(\vec{iv})$`]` defines an one-to-one memory to $\vec{iv}$ map, treating `A[B]` as an common pointer `AB`, constant along the loop and treated as an parameter in our dependence violation systems.

Byte precise / lengthened instrumentation: The current instrumentation and test generation assume data sizes are all equals, such that distinct memory accesses cannot overlap, even if the amount of loaded data is different.

7.2 Closing thoughts

This work demonstrates how to statically optimize some loops with inaccurate dependence information using existing optimizing tools. Run-time behavior, retrieved by profiling, guide the optimizer providing concrete dependence information not resolved statically. A run-time validation test, representing the space where the applied transformation violates at least one dependence, is simplified to a much simpler test using quantifier elimination. We believe this is an comprehensive method to resolve dependencies, including aliasing, allowing more loops to be optimized by polyhedron tools.

Apart from shortcomings that must be addresses, cited on the previous section, the current framework has no guaranties that the chosen transformation is actually profitable or if the run-time test would block valid transformations in most cases. Further research could address these limitations by performing an incremental profiling information, where the success ration of the generate test is accumulated across multiple executions. If the current test usually fails, a more complex instrumentation could retrieve more dependencies, used when retrieving the optimization, replacing the current test and transformed codes. Hardware counters could be used to determine if the applied optimization actually performs better than the original code by using hardware counters to evaluate both codes, and decide if the applied transformation is better than the original code.

Other possibility to address the two limitations could be by combining speculative systems with the current framework, such as Apollo. The first time an application is executed the speculative system runs the its usual way. If it finished the program execution using an optimized code, this optimized code can be exported to generate, statically, an lightweight run-time test, using the techniques here proposed. Just as described above, this could be part of an incremental profiling framework to decide upon the test or transformation quality.

We hope the presented work will be useful to further research.

Bibliography

Allen, R. and Kennedy, K. (2002). *Optimizing compilers for modern architectures: a dependence-based approach*, volume 289. Morgan Kaufmann. On page 35

Alves, P., Gruber, F., Doerfert, J., Lamprineas, A., Grosser, T., Rastello, F., and Pereira, F. M. Q. a. (2015). Runtime pointer disambiguation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 589–606, New York, NY, USA. ACM. On pages 53 and 82

APOLLO (2016). Apollo. http://apollo.gforge.inria.fr. On page 16

Apple (2007). clang: a c language family frontend for llvm. On page 25

Bastoul, C. (2004). Code generation in the polyhedral model is easier than you think. In *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, pages 7–16, Juan-les-Pins, France. On page 17

Benabderrahmane, M.-W., Bastoul, C., Pouchet, L.-N., and Cohen, A. (2008). A Conservative Approach to Handle Full Functions in the Polyhedral Model. Research Report RR-6814, INRIA. On page 17

Bik, A. J. C. (2004). *Software Vectorization Handbook, The: Applying Intel Multimedia Extensions for Maximum Performance*. Intel Press. On page 53

Bondhugula, U., Hartono, A., Ramanujam, J., and Sadayappan, P. (2008). Pluto: A practical and fully automatic polyhedral program optimization system. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008)*. On page 72

Bondhugula, U. K. R. (2008). *Effective Automatic Parallelization and Locality Optimization Using the Polyhedral Model*. PhD thesis, Ohio State University, Columbus, OH, USA. AAI3325799. On page 17

Brown, C. W. (2003). Qepcad b: A program for computing with semi-algebraic sets using cads. *SIGSAM Bull.*, 37(4):97–108. On pages 51, 55, 64, and 82

Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., and Sarkar, V. (2005). X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 519–538, New York, NY, USA. ACM. On page 16

Clauss, P., Fernandez, F., Garbervetsky, D., and Verdoolaege, S. (2009). Symbolic polynomial maximization over convex sets and its application to memory requirement estimation. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 17(8):983–996. On page 52

Clauss, P. and Tchoupaeva, I. (2004). *A Symbolic Approach to Bernstein Expansion for Program Analysis and Optimization*, pages 120–133. Springer Berlin Heidelberg, Berlin, Heidelberg. On page 82

Cohen, C. and Mahboubi, A. (2010). *A Formal Quantifier Elimination for Algebraically Closed Fields*, pages 189–203. Springer Berlin Heidelberg, Berlin, Heidelberg. On page 63

Cousot, P. and Halbwachs, N. (1978). Automatic discovery of linear restraints among variables of a program. In *5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'78)*, pages 84–96, Tucson, Arizona. ACM. On page 17

Davenport, J. H. and Heintz, J. (1988). Real quantifier elimination is doubly exponential. *Journal of Symbolic Computation*, 5(1):29 – 35. On page 63

Dolzmann, A. and Sturm, T. (1997). Redlog: Computer algebra meets computer logic. *SIGSAM Bull.*, 31(2):2–9. On page 84

Elango, V., Rastello, F., Pouchet, L.-N., Ramanujam, J., and Sadayappan, P. (2015a). On characterizing the data access complexity of programs. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 567–580, New York, NY, USA. ACM. On pages 71 and 76

Elango, V., Sedaghati, N., Rastello, F., Pouchet, L.-N., Ramanujam, J., Teodorescu, R., and Sadayappan, P. (2015b). On using the roofline model with lower bounds on data movement. *ACM Trans. Archit. Code Optim.*, 11(4):67:1–67:23. On page 71

Fauzia, N., Elango, V., Ravishankar, M., Ramanujam, J., Rastello, F., Rountev, A., Pouchet, L.-N., and Sadayappan, P. (2013). Beyond reuse distance analysis: Dynamic analysis for characterization of data locality potential. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(4):53. On page 71

Feautrier, P. (1988a). Array expansion. In *2nd International Conference on Supercomputing (ICS'88)*, pages 429–441. ACM. On page 17

Feautrier, P. (1988b). Parametric integer programming. *RAIRO Recherche opérationnelle*, 22(3):243–268. On page 17

Feautrier, P. (1992a). Some efficient solutions to the affine scheduling problem. i. one-dimensional time. *International Journal of Parallel Programming*, 21(5):313–347. On page 17

Feautrier, P. (1992b). Some efficient solutions to the affine scheduling problem. part ii. multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420. On pages 17, 22, and 23

Feautrier, P. (2015). The power of polynomials. In *5th International Workshop on Polyhedral Compilation Techniques*. http://impact.gforge.inria.fr/impact2015/. On page 52

Grösslinger, A. (2009). *The Challenges of Non-linear Parameters and Variables in Automatic Loop Parallelisation*. PhD thesis, Universität Passau. On pages 17, 51, 53, and 82

Grosser, T., Groesllinger, A., and Lengauer, C. (2012). Polly - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 04(22):X. On pages 17 and 24

Grosser, T., Pop, S., Ramanujam, J., and Sadayappan, P. (2015a). On recovering multi-dimensional arrays in polly. In *5th International Workshop on Polyhedral Compilation Techniques*. http://impact.gforge.inria.fr/impact2015/. On page 52

Grosser, T., Ramanujam, J., Pouchet, L.-N., Sadayappan, P., and Pop, S. (2015b). Optimistic delinearization of parametrically sized arrays. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 351–360, New York, NY, USA. ACM. On page 17

Grosser, T., Ramanujam, J., Pouchet, L.-N., Sadayappan, P., and Pop, S. (2015c). Optimistic delinearization of parametrically sized arrays. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 351–360, New York, NY, USA. ACM. On page 52

Henretty, T., Veras, R., Franchetti, F., Pouchet, L.-N., Ramanujam, J., and Sadayappan, P. (2013). A stencil compiler for short-vector simd architectures. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 13–24, New York, NY, USA. ACM. On page 16

Hind, M. (2001). Pointer analysis: haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 54–61. ACM. On page 16

Hong, H. (1992). Simple solution formula construction in cylindrical algebraic decomposition based quantifier elimination. In *Papers from the International Symposium on Symbolic and Algebraic Computation*, ISSAC '92, pages 177–188, New York, NY, USA. ACM. On page 51

Hong, H. and Din, M. S. E. (2012). Variant quantifier elimination. *Journal of Symbolic Computation*, 47(7):883 – 901. On page 64

Irigoin, F. and Triolet, R. (1988). Supernode partitioning. In *Symposium on Principles of Programming Languages (POPL'88)*, pages 319–328, San Diego, CA. On page 17

ISO (1999). The ANSI C standard (C99). Technical Report WG14 N1124, ISO/IEC. On page 16

Jimborean, A. (2012). *Adapting the Polytope Model for Dynamic and Speculative Parallelization*. Theses, Université de Strasbourg. On pages 17, 53, and 82

Kaplansky, V. (2006). Gcc patch: implementation of alias versioning in vectorizer. https://gcc.gnu.org/ml/gcc-patches/2006-05/msg00266.html. On pages 53 and 82

Karp, R. M., Miller, R. E., and Winograd, S. (1967). The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14(3):563–590. On page 17

Lamport, L. (1974). The parallel execution of DO loops. *Communications of the ACM*, 17(2):83–93. On page 17

Martinez Caamaño, J. M. (2016). *Fast and Flexible Compilation Techniques for Effective Speculative Polyhedral Parallelization*. Theses, Université de Strasbourg. On page 53

Maslov, V. (1992). Delinearization: An efficient way to break multiloop dependence equations. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, pages 152–161, New York, NY, USA. ACM. On page 52

Nethercote, N. and Seward, J. (2007). How to shadow every byte of memory used by a program. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, pages 65–74, New York, NY, USA. ACM. On pages 27 and 75

Nuzman, D., Eres, R., Guenther, R., Jelinek, J., Matz, M., Sandiford, R., and Rosen, I. (2003). Auto-vectorization in gcc. https://gcc.gnu.org/projects/tree-ssa/vectorization.html. On page 53

NVIDIA (2016). NVIDIA CUDA C programming guide. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf. Version 8.0. On page 16

Oancea, C. E. and Rauchwerger, L. (2012). Logical inference techniques for loop parallelization. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 509–520, New York, NY, USA. ACM. On pages 52 and 53

Park, E., Cavazos, J., Pouchet, L.-N., Bastoul, C., Cohen, A., and Sadayappan, P. (2013). Predictive modeling in a polyhedral optimization space. *International journal of parallel programming*, 41(5):704–750. On page 31

Pirk, H., Petraki, E., Idreos, S., Manegold, S., and Kersten, M. L. (2014). Database cracking: fancy scan, not poor man's sort! In *Tenth International Workshop on Data Management on New Hardware, DaMoN 2014, Snowbird, UT, USA, June 23, 2014*, pages 4:1–4:8. On page 16

Pop, S., Cohen, A., Bastoul, C., Girbal, S., andré Silber, G., and Vasilache, N. (2006). Graphite: Polyhedral analyses and optimizations for gcc. In *In Proceedings of the 2006 GCC Developers Summit*, page 2006. On pages 17 and 24

Pop, S., Cohen, A., and Silber, G.-A. (2005). Induction variable analysis with delayed abstractions. In *High Performance Embedded Architectures and Compilers*, pages 218–232. Springer. On pages 18 and 81

Pouchet, L.-N. (2010). PoCC, the Polyhedral Compiler Collection, version 1.3. http://pocc.sourceforge.net. On page 17

Pouchet, L.-N. and Yuki, T. (2015). PolyBench/C 4.1. http://polybench.sourceforge.net. On page 82

Pugh, W. (1991). 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, pages 4–13, New York, NY, USA. ACM. On pages 17, 37, 42, 51, and 82

Pugh, W. and Wonnacott, D. (1996). Non-linear array dependence analysis. In Szymanski, B. and Sinharoy, B., editors, *Languages, Compilers and Run-Time Systems for Scalable Computers*, pages 1–14. Springer US. On pages 42, 51, and 53

Rus, S., Rauchwerger, L., and Hoeflinger, J. (2002). Hybrid analysis: Static and dynamic memory reference analysis. In *ICS*, pages 251–283. IEEE Computer Society. On pages 17, 52, 53, and 82

Schweighofer, M. (2002). An algorithmic approach to schmüdgen's positivstellensatz. *Journal of Pure and Applied Algebra*, 166(3):307 – 319. On pages 19, 37, 38, 52, 67, and 82

Smaragdakis, Y. and Balatsouras, G. (2015). Pointer analysis. *Found. Trends Program. Lang.*, 2(1):1–69. On page 16

Sukumaran-Rajam, A. (2015). *Beyond the Realm of the Polyhedral Model: Combining Speculative Program Parallelization with Polyhedral Compilation*. Theses, Université de Strasbourg. On page 53

Sukumaran-Rajam, A., Martinez, J. M., Wolff, W., Jimborean, A., and Clauss, P. (2014). Speculative Program Parallelization with Scalable and Decentralized Runtime Verification. In Bonakdarpour, B. and Smolka, S. A., editors, *Runtime Verification*, volume 8734, pages 124–139, Toronto, Canada. Springer. On pages 16, 17, and 82

Suriana, P. A. (2016). Fourier-motzkin with non-linear symbolic constant coefficients. Master's thesis, Massachusetts Institute of Technology. On page 51

Sutter, H. (2005). The free lunch is over: A fundamental turn toward concurrency in software. http://www.gotw.ca/publications/concurrency-ddj.htm. On page 15

Trifunovic, K., Cohen, A., Edelsohn, D., Li, F., Grosser, T., Jagasia, H., Ladelsky, R., Pop, S., Sjödin, J., and Upadrasta, R. (2010). Graphite two years after: First lessons learned from real-world polyhedral compilation. In *GCC Research Opportunities Workshop (GROW'10)*. On page 17

van Engelen, R. A., Birch, J., Shou, Y., Walsh, B., and Gallivan, K. A. (2004). A unified framework for nonlinear dependence testing and symbolic analysis. In *Proceedings of the 18th Annual International Conference on Supercomputing*, ICS '04, pages 106–115, New York, NY, USA. ACM. On page 52

Venkat, A., Shantharam, M., Hall, M., and Strout, M. M. (2014a). Non-affine extensions to polyhedral code generation. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 185:185–185:194, New York, NY, USA. ACM. On page 17

Venkat, A., Shantharam, M., Hall, M., and Strout, M. M. (2014b). Non-affine extensions to polyhedral code generation. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 185:185–185:194, New York, NY, USA. ACM. On page 53

Verdoolaege, S. (2010). isl: An integer set library for the polyhedral model. In *Mathematical Software–ICMS 2010*, pages 299–302. Springer. On pages 17 and 83

Verdoolaege, S., Seghir, R., Beyls, K., Loechner, V., and Bruynooghe, M. (2007). Counting integer points in parametric polytopes using barvinok's rational functions. *Algorithmica*, 48(1):37–66. On page 83

Vijay, S., Bard, B., Igor, P., Olivier, T., and David, G. (2016). X10 language specification. http://x10.sourceforge.net/documentation/languagespec/x10-latest.pdf. Version 2.6. On page 16

WEISPFENNING, V. (1997). Simulation and optimization by quantifier elimination. *Journal of Symbolic Computation*, 24(2):189 – 208. On page 84

Williams, S., Waterman, A., and Patterson, D. A. (2009). Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76. On page 16

Wulf, W. A. and McKee, S. A. (1995). Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24. On page 16

www.netlib.org (1992). Linear algebra package. http://www.netlib.org/lapack/. On pages 24 and 81