Adapting the Polytope Model for Dynamic and Speculative Parallelization

Alexandra Jimborean

UNIVERSITÉ DE STRASBOURG

N° d'ordre :

École Doctorale Mathématiques, Sciences de l'Information et de l'Ingénieur

UdS – INSA – ENGEES

THÈSE

présentée pour obtenir le grade de

Docteur de l'Université de Strasbourg

Discipline : Informatique

par

Alexandra Jimborean

Adapting the Polytope Model for Dynamic and Speculative Parallelization

Soutenue publiquement le 14 Septembre 2012

Membres du jury:

Directeur de thèse :	Philippe Clauss, Professeur, UdS, Strasbourg
Co-encadrant :	Vincent Loechner, Maître de conférences , UdS, Strasbourg
Rapporteur :	Albert Cohen, Directeur de Recherche, INRIA - ENS, Paris
Rapporteur :	André Seznec, Directeur de Recherche, INRIA - IRISA, Rennes
Rapporteur :	John Cavazos, Professeur, University of Delaware, USA
Examinateur :	François Bodin, Directeur technique, CAPS Enterprise, Rennes
Examinateur :	Jean Christophe Beyler, Ingénieur logiciel senior, Intel, Versailles

Laboratoire des Sciences de l'Image, de l'Informatique, et de la Télédétection **UMR 7005**

Per aspera ad astra

Contents

List of Figures

Chapter 1

Introduction

1.1 Context of the work

The advent of multicore processors imposes new strategies for reaching good software performance and exploiting advantageously the provided hardware. The key to success is now radically related to parallelism and applications have to be run in phases where many computations are performed simultaneously on all the available processor cores. There are several possible options to accomplish this: programs can be written by explicitly describing what can be run in parallel and what cannot, the compiler can extract parallel computations from a serial code by performing advanced code analyses and then generate parallel code, or the software can be run on top of a runtime system, or virtual machine, performing on-the-fly analyses and parallelizations. Nevertheless, although each option has been intensely studied, they all have some inherent limitations.

Even if many parallel programming languages are available, such as OpenMP, MPI, Cilk, TBB, HMPP, OpenCL or CUDA, parallel programming is in general difficult, because the programmer is required to handle complex issues, such as selecting a convenient algorithm for parallelization, analyzing the dependences between parts of the code, ensuring correct semantics, being aware of the underlying hardware characteristics and using a suitable programming model. Moreover, performance portability is difficult to be ensured due to hardware heterogeneity. Such issues significantly increase the software time-to-market. To aid the programmer in delivering parallel code, building compilers that perform automatic parallelization became an active research area.

Compilers dedicated to automatic parallelization have a rich history, particularly focusing on scientific computing applications. Examples of such compilers are SUIF [132], Polaris [13] and PIPS [35], that are able to automatically parallelize sequential programs without the programmer's intervention. They mostly focus on *for*-loops accessing multi-dimensional arrays and referencing array elements through linear functions of the loop indices. Thanks to precise data dependence analysis, such loops can take advantage of advanced parallelizing transformations as tiling, loop splitting or fusion, loop interchange, loop skewing and more generally linear loop transformations. The theory concerning loop analyses and transformations is unified in a well-known mathematical

framework called the polytope model [44, 45]. However, its applicability is limited to array-based scientific applications exhibiting explicit linearity of their loop bounds and array accesses. Loops exhibiting complex exit conditions and memory accesses through pointers or indirect arrays cannot be handled at compile-time using these techniques, since crucial information can only be known at execution-time. Moreover, it is generally difficult for a compiler to select the parallelizing and optimizing transformations that would perform well under various circumstances (in different execution contexts or on distinct processors).

The third solution is to run the targeted program in the frame of a runtime system whose role is to use advantageously the available dynamic information and automatically parallelize on-the-fly some code parts. One main advantage is that the effectiveness of a code transformation can immediately be evaluated and the course of execution can be adjusted accordingly by the runtime system in real time. In particular, *speculative* parallelizing techniques are possible since an online verification can consecutively launch recovery actions, in case previously speculated information is invalidated, such as canceling wrong computations and restarting them from the last correct state. This approach does not have, a priori, any limitation on the type of targeted code, however it is strongly constrained by the time overhead inevitably introduced. Hence, it is impossible to perform complete analyses and optimizations at runtime, as done by a compiler. On the other hand, generating *efficient* code is mandatory.

In this current context, speculative parallelization is an essential strategy to handle the parallelization of general-purpose codes. A well-researched direction in speculative parallelization is thread-level speculation (TLS) [19, 66, 70, 76, 103, 105, 106, 114, 119, 137]. A TLS framework allows optimistic execution of parallel code regions before all dependences between instructions are known. Hardware or software mechanisms track register and memory accesses to determine if any dependence violation occurs. In such cases, register and memory state are rolled back to a previous correct state and sequential re-execution is initiated. Our proposal focuses on enhancing previous works on speculative parallelism, by providing advancements in the quality of the generated parallel code. The highlights of this thesis are:

1. *Optimize and parallelize* the code at runtime, by applying a polyhedral transformation prior to parallelization. This has twofold consequences: first, boosting the performance of the generated parallel code, and second, exhibiting parallelism in codes which could not be parallelized in the original form. Chapter 2 introduces the mathematical background of the polyhedral model and a general overview of TLS systems, followed by our proposal on how the polyhedral model can be applied speculatively at runtime, in Chapter 3.

2. *Fully exploit parallelism* in codes that exhibit different phases during one execution, by adapting to the current behavior of the code. Our strategy relies on slicing the execution in intervals, where each interval represents a program's phase. For each phase, we prepare a custom code version, in accordance with the properties exhibited by the code during the certain phase. Chapter 4 details the multi-versioning and the chunking techniques we employ for this purpose.

From the point of view of the implementation techniques, this thesis provides an

insight on:

1. optimizing and parallelizing transformations performed at runtime, which consist of (1) a linear re-scheduling of the target loop nest's iterations in order to reveal parallelism and (2) the parallelization of the *outermost parallel* loop in the transformed code;

2. adaptation of the polytope model to dynamic and speculative parallelization;

   - speculation based on the representation of the memory accesses as predicting linear functions of the loop indices, obtained by interpolating addresses accessed during execution samples;
   - dynamic dependence analysis by computing the dependence distance vectors from memory addresses accessed during execution samples;
   - dynamic generation of speculatively parallel code, using binary patterns patched at runtime;
   - rollback in case of misprediction, by canceling the last executed chunk.

3. global orchestration of the loop nest execution, by running successive slices, or chunks, of the outermost loop that are executed serially, but intra-chunk iterations are run in parallel;

4. software implementation as a collaborative static-dynamic framework consisting of extensions of the LLVM compiler and an x86-64 runtime system.

We implemented the entire system in a platform called VMAD, which stands for ***V**irtual **M**achine for **A**dvanced, **D**ynamic Code Instrumentation and Optimization*. The technical details of the framework are presented in Chapter 5, which describes the code preparation at compile-time and the actions taken by the runtime system during execution. Next, the experimental evaluations are included in Chapter 6.

Moreover, we illustrate VMAD as a generic framework, which in addition to performing speculative parallelization, is suitable to accomplish various types of complex code instrumentation, analysis and optimization. Such applications are described in Chapter 7.

Last but not least, the conclusions, introspections and perspectives of our work are presented in the end of the thesis.

The next section is dedicated to setting the context of our research, by reviewing the outstanding previous proposals concerning each important aspect of our work.

1.2 Static-dynamic frameworks for analysis and optimization

Our framework, VMAD, is a platform for dynamic code instrumentation and optimization, targeting automatic parallelization on-the-fly. In what follows we present some of the outstanding related works concerning the main characteristics and contributions

of VMAD, followed by a more detailed survey of the state of the art in the subsequent chapters. Chapter 2, Section 2.3 sets the frame of our work as a TLS system, compared to previous techniques. Once all related notions are introduced, together with the background of our work, Chapter 4, Section 4.2 presents previous approaches for multiversioning and its applications, while Section 4.3 reviews works that performed chunking on loops. Chapter 5, Section 6.1.1 gives an overview of various works targeting code instrumentation and profiling; and finally Chapter 7, Section 7.2 describes several works on runtime selection among several optimized versions.

To outline the context of our work, we review in what follows the closest and most representative tools and methods designed to perform:

- Code instrumentation : PIN [79], DynamoRIO [18],
- Polyhedral code optimization : Pluto [15],
- Speculative code parallelization : LRPD [106] and R-LRPD [33] tests.

PIN: Pin [79] is a dynamic binary instrumentation engine for the Intel architectures (x86 and x86-64 instruction sets) which allows the programmer to build customized instrumentation tools, called *Pintools*. It has already been included in many Intel commercial tools, such as Intel Parallel Inspector, Intel Parallel Amplifier and Intel Parallel Advisor. PIN inserts snippets of code in a running program to collect runtime information. Thus, PIN is suitable both for program analysis, useful in performance profiling, error detection, capture and replay; as well as for an architectural study, offering support for processor and cache simulation or trace collection. Thanks to its flexibility, various tools can be generated, for emulation, security and parallel program analysis. Since PIN is a dynamic engine, it uses Just-in-time compilation to compile (and optimize) the instrumentation code just before it is executed. The pintools can be attached to the binary code or even to the running process, without the need to recompile or re-link the code. Using this mechanism, it can also handle dynamically generated code. A pintool consists of three types of routines: instrumentation, analysis and callback routines.

1. Instrumentation routines: once the program is loaded in memory, PIN's strategy is to take control of the program and to JIT some small pieces of code, thus inserting in the original code calls to the analysis routines.
2. Analysis routines: they are executed when the code to which they are associated begins its executions and are aimed to collect the required dynamic information.
3. Callback routines: these are special callbacks invoked when certain conditions are met, or when specific events occur.

PIN provides an explicit API, very flexible and easy to use for building tools dedicated to a large palette of instrumentation types. However, due to its overhead, PIN is not yet ready to be used as an online profiler, as part of a dynamic optimization phase. According to Bach *et al.* [6], the inherent overhead of PIN revolves around 30%, when no pintools are executed. Depending on the purpose of instrumentation, extra overhead

is added. For instance, for basic block counting, the overhead of PIN is up to 2000%, according to Hazelwood *et al.* [79]. Unlike VMAD, one cannot instrument by sampling, as the instrumentation inserted with PIN cannot be disabled during the execution of the code, leading to a high overhead. Thus it cannot be used in a dynamic optimizer.

DynamoRIO: Very similar to PIN, DynamoRIO [18] is a framework designed for dynamic instrumentation of codes, by allowing the programmer to write their own tool for instrumentation, analysis, profiling, optimization or any other code transformation. Thus, unlike other similar tools, DynamoRIO does not only monitor the code, but it can also perform any kind of code modifications, as the application is running, as specified by the programmer. From a performance perspective, the base overhead of DynamoRIO is 45% in average, however, an optimized version outperforms PIN for basic block counting [18]. Still, its high overhead makes this tool unsuitable for being embedded in a dynamic optimizer.

Pluto: Pluto [15] is a framework for automatic parallelization and optimization of affine loop nests. It is a source-to-source compiler that builds an abstract representation of the loop nest, using the polyhedral model, for further loop transformations. All loop optimizations, such as tiling, fusion, unrolling, are manipulated in the polyhedral representation, from which the new loop nest is generated in the transformed form. The goal is to optimize for data locality and to expose parallelism, simultaneously. Thus, the compiler defines a cost function, aimed to evaluate distinct transformations and to select the best. The transformation is defined as a hyperplane, evaluated by the cost function, based on the data reuse distance. The transformation aims to minimize the reuse distance in order to limit the volume of communicated data. For instance, for a tiling hyperplane, the cost function can provide an upper bound on the data communicated between the tiles, depending on the tile size. Pluto is designed to detect synchronization-free parallelism, permutable loops or pipelined parallelism at various levels and to automatically transform the loops in the corresponding form. The experimental results demonstrate that codes automatically optimized and parallelized with Pluto perform very well, considerably outperforming other research or commercial compilers [14, 15, 16]. Despite its very promising performance, Pluto is limited to statically analyzable codes embedding affine loop nests, very frequent in scientific codes. On the other hand, general purpose codes cannot benefit from the great advantages of the polytope model, as they cannot be fully disambiguated at compile time. It is the goal of our work to dynamically model the loops in the polyhedral representation and to perform automatic optimizations and parallelization.

The LRPD test: Among the pioneers of speculative parallelization are the authors of the LRPD test [106], who noticed the frequency of statically undetectable parallel loops that occur in general purpose codes. Their proposal is to speculatively execute all loops in parallel and to perform a dependence analysis at runtime, to ensure that the semantics of the code is preserved. To enhance parallelism, they apply privatization and reduction parallelization methods dynamically and check their validity during execution. As soon as a dependence violation is detected, the loop is re-executed sequentially.

Although this might lead to significant performance losses in case of non-parallelizable loops, this work set the premises of speculative parallelization.

In contrast to previous works, they relax the constraints on the dependences, consequently only flow dependences lead to a rollback and to a sequential re-execution of the loop. To ensure correctness in the presence of output and anti dependences, they apply privatization and reduction methods, thus eliminating those dependences. Additionally, they develop the Lazy Reduction Privatizing Doall (LRPD) test to check dependences dynamically. The compiler is forced to take conservative decisions, however during execution, one can verify the exact dependences that occur. For example, if a variable is read, but its use is under a conditional, the compiler must consider a dependence. However, at runtime the correct decision is taken, whether the variable has been used or not and whether a private copy of the value could remove the dependence. The test is also applicable to detect cross-processor dependences.

All in all, the LRPD test is able to handle any type of loops, without restrictions, nevertheless, as soon as a dependence is detected, the whole loop is re-executed sequentially.

The R-LRPD test: To overcome the limits of the LRPD test, the R-LRPD [33] test aims to exploit partial parallelism of loops. It transforms a partially parallel loop into a sequence of fully parallel loops and speculatively executes each of the newly obtained loop with the LRPD test. The execution in phases of the loops shows some similarities to our work, for detecting different behaviors of one loop during one execution. But, in addition to parallelization, VMAD targets also loop optimizations, by applying a polyhedral transformation.

Chapter 2

Polyhedral transformations and speculative parallelism: an overview

This chapter introduces the theoretical notions of the polytope model in Section 2.1 and a general overview of the traditional TLS systems in Section 2.2. An overview of previous approaches dedicated to speculative automatic parallelization of loops is given in Section 2.3, which presents also detailed works on components of TLS systems, such as mechanisms for predictions, dependence analysis, transactional memory; as well as previous approaches applying the polyhedral model, however statically. Finally, we conclude the chapter by underlining the limits of current TLS systems and proposing solutions to improve them.

2.1 The polytope model

The polytope model, also called the polyhedral model, is a very powerful mathematical abstraction used in loop optimizations. In geometry, a polytope represents a geometric object with flat sides, which can exist in any number of dimensions. In particular, we are interested in a special class of polytopes, the *convex polytopes*, having the property that they are convex sets of points in an n-dimensional space $\mathbb{K}^n$. This class finds its application in linear programming, as any linear transformation preserves the properties of the points building the polytope. Additionally, a convex polytope may be defined in a number of ways, depending on what is more suitable for the problem at hand. One definition is in terms of a convex set of points in space, but among other important definitions are: as the intersection of half-spaces (half-space representation) or as the convex hull of a set of points (vertex representation).

Abstracting a loop in the polyhedral representation is equivalent to associating to each dynamic instance (iteration) of each statement an integer point in space, contained in the *statement's polyhedron*. We introduce the mathematical notions required for modeling the loop nest. An exhaustive presentation of applications of the polytope model in program optimizations are given by Feautrier in multiple works [41, 42, 44, 45, 46] and has been addressed in many others dedicated to compile-time [15, 16, 55] or runtime loop optimizations [101]. For a throughout presentation of the mathematical apparatus building the underlying background of the polytope model, the reader is re-

ferred to the monograph of Schrijver [109]. In what follows we present only an overview, detailing the aspects required to understand the background of this dissertation.

2.1.1 Notations and definitions

We denote by $\vec{v}$ a vector, by $|\vec{v}|$ its dimension, and by $\vec{v}$[i] the i^{th} element of $\vec{v}$.

Definition 1: (*Affine function*) A function f: $\mathbb{K}^m \rightarrow \mathbb{K}^n$ is said to be affine iff $\exists$ a matrix A $\in \mathbb{K}^{n \times m}$ and a vector $\vec{b} \in \mathbb{K}^n$ such that:

$$\forall \vec{x} \in \mathbb{K}^m, \text{f}(\vec{x}) = \text{A}\vec{x} + \vec{b}.$$

Definition 2: (*Affine hyperplane*) An *affine hyperplane* of an n-dimensional affine space V is a subspace of dimension $n-1$, defined by a linear equation in $\vec{x} \in \mathbb{K}^n$ of the form:

$$\vec{a} \cdot \vec{x} = b,$$

where $\vec{a} \in \mathbb{K}^n$ (at least one element $\vec{a}$[i] $\neq 0$) and b is a scalar from $\mathbb{K}$.

Definition 3: (*Affine half-space*) An affine hyperplane divides the space into two *half-spaces*, defined by the inequalities:

$$\vec{a} \cdot \vec{x} \geq \text{b}$$

and

$$\vec{a} \cdot \vec{x} \leq \text{b},$$

where $\vec{a} \in \mathbb{K}^n$ (at least one element $\vec{a}$[i] $\neq 0$) and b $\in \mathbb{K}$.

Definition 4: (*Convex polyhedron*) The intersection of a finite number of *affine* half-spaces defines a *convex polyhedron*, each half-space providing a face of the polyhedron. Formally, the polyhedron $P \subset \mathbb{K}^n$ can be expressed as a set of m affine constraints in $A \in \mathbb{K}^{m \times n}$ and $\vec{b} \in \mathbb{K}^m$:

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

Definition 5: (*Parametric polyhedron*) A polyhedron P may be parametrized by a vector of parameters $\vec{p}$ and is denoted by $P(\vec{p})$. It can be defined by a matrix $A \in \mathbb{K}^{m \times n}$, a matrix of symbolic coefficients $B \in \mathbb{K}^{m \times p}$, where p is the dimension of the vector of parameters $|\vec{p}| = p$ and a vector of constants $\vec{b} \in \mathbb{K}^m$ as:

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

Definition 6: (*Polytope*) A bounded polyhedron is called a *polytope*.

Libraries such as Polylib [96], PPL [99], isl [127], Barvinok [128], PIP [41] or Cloog [9] provide a range of functions implementing operations on unions of polyhedra in $\mathbb{Q}^n$, such as:

- set operations (union, difference, intersection...),
- image and inverse image (preimage) with respect to an affine function (transformation),
- existence of an integer value ("point") inside the polyhedra,
- counting the number of points included in a parametric polytope (Ehrhart polynomial [30]),
- lexicographic minimum/maximum (parametrically),
- scanning integer points.

2.1.2 The polyhedral representation of a loop nest

The compact polyhedral representation allows to model the dynamic instances of each statement of a loop nest in a precise manner, in the view of computing the dependences between the statements and performing code transformations. Given the loop nest:

Listing 2.1: Affine loop nest

```
for i = 1,N
    for j = 1,N
        S : A[i,j] = A[i-1][j+1]+1
    endfor
endfor
```

the statement S can be either one instruction, or a set of consecutive instructions. A dynamic instance of S is given by each execution of S, in different iterations: $S_{1,1}$, $S_{1,2}$, ..., $S_{1,N}$, ..., $S_{N,N}$. Since the dynamic instances are defined in terms of the values of the enclosing loop iterators, this leads us to the next definition:

Definition 7: (*Iteration vector*) The *iteration vector* of a statement S, denoted by $\vec{x_S}$, is the n-dimensional vector of values of the iterators of the n loops enclosing S.

Definition 8: (*Domain, index set*) The set of all iterations vectors of a statement S is called the *domain* or the *index set* of the statement, denoted by $\mathcal{D}^S$.

```
for (i = 1; i ≤ N; i++)
    for (j = 1; j ≤ N; j++)
        if (i + j ≤ 9)
            S
```

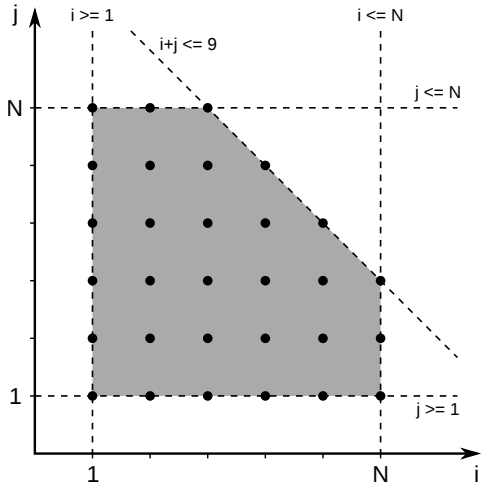


Figure 2.1: Sample loop nest (left) and the iteration domain of S for N = 6(right) (courtesy: B. Pradelle)

$$\begin{cases} i-1 \geq 0 \\ j-1 \geq 0 \\ i+N \geq 0 \\ j+N \geq 0 \\ -i-j+9 \geq 0 \end{cases} \qquad \mathcal{D}^S = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \in \mathbb{Z}^2 \middle| \begin{pmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ -1 & -1 & 0 & 9 \end{pmatrix} \begin{pmatrix} i \\ j \\ N \\ 1 \end{pmatrix} \geq 0 \right\}$$

Figure 2.2: Domain of statement S from Figure 2.1

Consider the affine loop nest in Figure 2.1. Note that statement S is under a conditional, hence not all the points in the iteration space are actually executed. The domain of S, $\mathcal{D}^S$, is depicted on the right. The iteration domain is expressed as a set of linear inequalities, as shown in Figure 2.2

If the loop bounds and data accesses can be expressed as affine functions of the enclosing loop indices and other parameters, and all conditionals in the loop are statically predictable, then the domain of every statement in the loop can be defined as a polyhedron, where the iteration vector is defined in $\mathbb{Z}^d$ and d is the depth of the innermost loop enclosing the statement S:

$$\mathcal{D} = \{\vec{x} \in \mathbb{Z}^d | A\vec{x} \geq \vec{b}\}$$

Considering also the parameters, the complete definition of a parametric polyhedron is:

$$\mathcal{D} = \left\{\vec{x} \in \mathbb{Z}^d | A\vec{x} + B\vec{p} + \vec{b} \geq 0\right\}$$

Equivalently, this represents the intersection of several half-spaces. The regions of code containing only affine loop nests and conditionals that can be evaluated statically, are known as *Static Control Part (SCoP)*. In a SCoP, the loop bounds, the conditionals and

the data access functions are defined as affine functions on the enclosing loop indices and some parameters. The values of the parameters must not necessarily be known at compile-time, however, they must remain fixed during the execution of the SCoP. Additionally, there are no constraints on the structure of the loop nests, both perfectly and imperfectly nested loops being accepted.

2.1.3 Data access functions

Given a statement S executed inside a loop nest, not only the iteration domain of the statement is representative, but also the memory accesses it performs. For a precise analysis of the code in the polytope model, these accesses must be modeled as affine functions of the enclosing loop indices. Informally, each memory access can be seen as an array access with the subscripts defined as affine functions of the outer loop indices. For computing the memory location actually accessed, one uses a function of the form:

$$f(\vec{x}) = F\vec{x} + \vec{f} \qquad (1)$$

where F is matrix $\mathcal{M}_{s\times d}$ and $\vec{f}$ a vector of size s, where s is the dimension of the array and d the depth of the statement. As an example consider the loop nest:

Listing 2.2: Affine loop nest

```
for i = 1,N
    for j = 1,N
        S : A[i,j] = A[i-1][j+1]+B[2·i+j+5]
    endfor
endfor
```

The memory accesses performed by S are two read operations, R^A: A[i-1][j+1] and R^B: B[2·i+j+5]; and one write: W^A: A[i,j]. One obtains the following access functions, with $\vec{x} = \begin{pmatrix} i \\ j \end{pmatrix}$:

$$f_{R^A}(\vec{x}) = \begin{pmatrix} i-1 \\ j+1 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} -1 \\ 1 \end{pmatrix}$$

$$f_{R^B}(\vec{x}) = 2 * i + j + 5 = \begin{pmatrix} 2 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + 5$$

$$f_{W^A}(\vec{x}) = \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

Considering also the parameter N of the loop nest, the functions are rewritten in a more compact form as:

$$f_{R^A}(\vec{x}) = \begin{pmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \\ N \\ 1 \end{pmatrix}$$

$$f_{R^B}(\vec{x}) = \begin{pmatrix} 2 & 1 & 0 & 5 \end{pmatrix} \begin{pmatrix} i \\ j \\ N \\ 1 \end{pmatrix}$$

$$f_{W^A}(\vec{x}) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ N \\ 1 \end{pmatrix}$$

Data accesses to scalars are treated similarly, as if each scalar was an array with only one element.

2.1.4 Scheduling the statements

By applying a polyhedral transformation, one transforms a sequential loop nest into a semantically equivalent, optimized or parallel loop nest. Such transformations must express the execution order of the statements, must be instruction-wise and should handle loops in the presence of parameters. Since neither the domain, nor the data access functions of a statement can provide information concerning the execution order, one must add an extra piece of information, associated to each statement. Thus, each dynamic instance of a statement is associated a *logical date*, defining the space and time when that particular instance should be executed. For a precise and uniform representation, the logical dates are multidimensional, i.e. intuitively can be seen as days, hours, minutes, seconds ..., starting with the most significant components on the first positions, until the least significant ones (lexicographic order). Assigning a logical date is achieved by means of a function, called *scheduling function*.

Definition 9: (*Scheduling function*) The *scheduling function* of a statement S, known also as the *schedule* of S, is a function that maps each dynamic instance of S to a logical date, expressing the execution order between statements:

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

The associated timestamps allows one to order the instructions according to the lexicographic order, denoted as $\ll$, as component-wise comparison of vectors:

$$(a_1, \ldots, a_n) \ll (b_1, \ldots, b_n) \Leftrightarrow \exists i : 1 \leq i \leq n, \forall m : 1 \leq m < i,\ a_m = b_m \wedge a_i < b_i$$

Intuitively, two dynamic instances having the same timestamps can be executed in parallel. More precisely, the scheduling function is a mirror of the sequential execution

order of the dynamic instances. Note that, unlike the iteration vectors, the scheduling function can express also the textual ordering of the statements. For example, consider the following loop nest with several statements enclosed in the innermost loop:

Listing 2.3: 2-depth affine loop nest

```
for ( i = 0; i < N; i++ )
    for ( j = 0; j < N; j++ )
        S1
        S2
    endfor
endfor
```

Both statement $S1$ and $S2$ have the same iteration vectors, since they only depend on the enclosing loop indices. They cannot capture the execution order of different statements, defined in the same loop. On the other hand, the scheduling function is able to maintain this information, by interleaving constants between each loop level, which are aimed to express the textual order:

$\theta^{S_1}((\mathtt{i},\mathtt{j})) = (0,\mathtt{i},0,\mathtt{j},0)$
$\theta^{S_2}((\mathtt{i},\mathtt{j})) = (0,\mathtt{i},0,\mathtt{j},1)$

In this representation, one can easily deduce the order of the statements $S1$ and $S2$. The first two constants, 0 and 0, indicate that they are executed at the same loop level, but $S1$ comes before $S2$ in the textual order, according to the last constant of each vector.

More formally, applying the schedule $\theta(\vec{x}) = T\vec{x} + \vec{t}$ to the integer points of the iteration domain $\mathcal{D} = \left\{\vec{x} | A\vec{x} \geq \vec{b}\right\}$, is expressed as a polyhedron of the form:

$$\mathcal{D}' : \left[\begin{array}{c|c} I & -T \\ \hline 0 & A \end{array}\right] \left(\begin{array}{c} \theta(\vec{x}) \\ \hline \vec{x} \end{array}\right) \begin{array}{c} = \\ \hline \geq \end{array} \left(\begin{array}{c} \vec{t} \\ \hline \vec{a} \end{array}\right)$$

For example, given the loop in Fig 2.1 one obtains the polyhedron:

$$\mathcal{D}^S : \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ -1 & -1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} \geq \begin{pmatrix} 1 \\ -N \\ 1 \\ -N \\ -9 \end{pmatrix}$$

and for a scheduling

$$\theta \begin{pmatrix} i \\ j \end{pmatrix} = i + j$$

one obtains:

$$\Theta^S : \left[\begin{array}{c|cc} 1 & -1 & -1 \\ \hline 0 & 1 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & -1 \\ 0 & -1 & -1 \end{array}\right] \left[\begin{array}{c} i' \\ \hline \\ i \\ j \\ \\ \\ \end{array}\right] \begin{array}{c} = \\ \hline \\ \\ \geq \\ \\ \\ \end{array} \left[\begin{array}{c} 0 \\ \hline 1 \\ -N \\ 1 \\ -N \\ -9 \end{array}\right] \tag{2}$$

This is commonly referred to as the *generalized change of basis*.

Scheduling matrices By applying an affine scheduling function $\theta(\vec{x}) = T\vec{x} + \vec{t}$ on the iteration domain $\mathcal{D}^S$ of a statement S, one obtains the *scheduling matrix* Θ^S of S, where $\Theta^S \in \mathbb{Z}^{d^t \times (d+p+1)}$, with $d^t = |\vec{t}|$:

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

For a throughout understanding, we continue with a few examples showing the purpose of the scheduling matrices.

Listing 2.4: Matrix multiply

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

Identity schedules: By applying the identity schedules on each of the statements in Listing 2.4:

$$\Theta_{S_1}(\vec{x}_{S_1}) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ N \\ 1 \end{pmatrix} = \begin{pmatrix} i' \\ j' \end{pmatrix}$$

and

$$\Theta_{S_2}(\vec{x}_{S_2}) = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ k \\ N \\ 1 \end{pmatrix} = \begin{pmatrix} i' \\ j' \\ k' \end{pmatrix}$$

one obtains the same code as listed in Listing 2.4. On the other hand, a slight change of the schedule for statement S_2:

$$\Theta_{S_1}(\vec{x}_{S_1}) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ N \\ 1 \end{pmatrix} = \begin{pmatrix} i' \\ j' \end{pmatrix}$$

and

$$\Theta_{S_2}(\vec{x}_{S_2}) = \begin{pmatrix} 1 & 0 & 0 & \mathbf{1} & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ k \\ N \\ 1 \end{pmatrix} = \begin{pmatrix} i' + N \\ j' \\ k' \end{pmatrix}$$

results in the code in Listing 2.5.

Listing 2.5: Matrix multiply with a new schedule

```
for ( i=0; i< N; i ++)
    for ( j=0; j< N; j ++)
      S1: C[i][j] = 0;

for ( i=N; i< 2*N; i ++)
    for ( j=0; j< N; j ++) {
      for ( k=0; k< N; k ++)
          S2: C[i-N][j]+=A[i-N][k] * B[k][j];
```

Which generates a loop distribution, because the loop nest had to be split in two different nests to preserve the semantics of the scheduling matrices.

Finally, one can perform a loop interchange with the following schedules:

$$\Theta_{S_1}(\vec{x}_{S_1}) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ N \\ 1 \end{pmatrix} = \begin{pmatrix} i' \\ j' \end{pmatrix}$$

and

$$\Theta_{S_2}(\vec{x}_{S_2}) = \begin{pmatrix} 1 & 0 & 0 & \mathbf{1} & 0 \\ 0 & \mathbf{0} & \mathbf{1} & 0 & 0 \\ 0 & \mathbf{1} & \mathbf{0} & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ k \\ N \\ 1 \end{pmatrix} = \begin{pmatrix} i' + N \\ k' \\ j' \end{pmatrix}$$

which is equivalent to the code in Listing 2.6.

Listing 2.6: Matrix multiply with interchange

```
for ( i=0; i< N; i ++)
    for ( j=0; j< N; j ++)
      S1: C[i][j] = 0;

for ( i=N; i< 2*N; i ++)
    for ( k=0; k< N; k ++) {
      for ( j=0; j< N; j ++)
          S2: C[i-N][k]+=A[i-N][j] * B[j][k];
```

Canonical form of the scheduling matrices: To make the scheduling matrix more meaningful, Cohen *et al.* [10, 31] propose a normalized representation, by decorating it both with static and dynamic information. Thus the scheduling matrix Θ^S of a statement S contains one extra row for each loop level, to define the relative order of the statements within each iteration, plus one row for the loop at depth 0, which yields $2d^S+1$ rows (d^S is the depth of the innermost loop enclosing S).

This encoding is suitable for expressing compositions of transformations, as it is decomposable in three sub-matrices:

1. *The iteration ordering matrix* $A^S \in \mathcal{M}_{d^S,d^S}(\mathbb{Z})$ representing the iteration vectors;
2. *The matrix of parameters* $\Gamma^S \in \mathcal{M}_{d^S,d_g p+1}(\mathbb{Z})$, whre $d_g p$ denotes the number of global parameters;
3. *The statement ordering vector* $\beta \in \mathbb{N}^{d^S+1}$, which specifies the order of S among the other statements executed at the same iteration.

The structure of the canonical schedule matrix is:

$$\Theta^S = \left[\begin{array}{ccc|ccc|c} 0 & \cdots & 0 & 0 & \cdots & 0 & \beta_0^S \\ A_{1,1}^S & \cdots & A_{1,d}^S & \Gamma_{1,1}^S & \cdots & \Gamma_{1,p}^S & \Gamma_{1,p+1}^S \\ 0 & \cdots & 0 & 0 & \cdots & 0 & \beta_1^S \\ A_{2,1}^S & \cdots & A_{2,d}^S & \Gamma_{2,1}^S & \cdots & \Gamma_{2,p}^S & \Gamma_{2,p+1}^S \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ A_{d,1}^S & \cdots & A_{d,d}^S & \Gamma_{d,1}^S & \cdots & \Gamma_{d,p}^S & \Gamma_{d,p+1}^S \\ 0 & \cdots & 0 & 0 & \cdots & 0 & \beta_d^S \end{array}\right]$$

In this form, various transformations can easily be expressed: by affecting A^S one defines a loop interchange (2.6), skewing or loop reversal; by altering Γ^S one generates shifting transformations, as in Listing 2.5; and by modifying β^S one can redefine the execution order of the instructions, equivalent to performing loop fission or loop fusion.

For the matrix multiplication example from Listing 2.4 one obtains:

$$A^{S_1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad \Gamma^{S_1} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \quad \beta^{S_1} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

$$A^{S_2} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \Gamma^{S_2} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \beta^{S_2} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Considering two statements at the same loop level as in Listing 2.3, the three matrices are for each statement respectively:

$$A^{S_1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad \Gamma^{S_1} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \quad \beta^{S_1} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

$$A^{S_2} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad \Gamma^{S_2} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \quad \beta^{S_2} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

This unified representation can describe a wide set of polyhedral transformations as well as compositions of such transformations in the view of generating efficient parallel code.

2.1.5 Static dependence analysis

Such loop transformations need to be validated according to the semantics of the sequential execution, to ensure that the resulting code preserves its correctness. The mechanism of dependence analysis was introduced for this purpose, studied extensively by Feautrier [42].

Definition 9: (*Dependence of Statement Instances*) Two statements S and R are said to be dependent, if there exist two instances $S(\vec{x_S})$ and $R(\vec{x_R})$, where $\vec{x_S}$ and $\vec{x_R}$ belong to the iteration domains of S and R, respectively such that $S(\vec{x_S})$ and $R(\vec{x_R})$ access the same memory location and at least one is a write.

If in the original sequential order $S(\vec{x_S})$ is executed before $R(\vec{x_R})$, R is said to be dependent on S. Under these circumstances, S is called the *source* and R is the *destination* of the dependence. Reversely S depends on R, if R is executed first in the sequential order.

To preserve the semantics, the execution of two dependent statements must be the same in the original sequential and in the transformed parallel order. On the other hand, two independent statement can be executed in arbitrary order.

Dependences are classified in three categories, depending on the order of read and write operations:

- RAW: read-after-write, or *flow dependence*;
- WAR: write-after-read, or *anti-dependence*;
- WAW: write-after-write, or *output dependence*;

Note that RAR, read-after-read, is not considered a dependence, since the memory is not altered, hence the order of execution of the two read operations is arbitrary. However, these accesses are considered by some optimizations to improve data locality.

Various algorithms have been designed to rewrite *anti-* and *output dependences*, which alter the same memory location. The most common strategies are privatization [106], renaming or expansion [43], such that the constraints are relaxed and more optimizing techniques become legal.

The formal representation of the dependences occurring in a loop nest is in the form of a *dependence polyhedron*, $\mathcal{P}_e$. Intuitively, if two dynamic instances $S(\vec{x_S})$ and $R(\vec{x_R})$ are dependent, there exists a relation $\{\vec{x_S} \rightarrow \vec{x_R}\}$. If $|\vec{x_S}| = |\vec{x_R}|$ there exist an edge in the polyhedron $\mathcal{P}_e$, having as vertices $S(\vec{x_S})$ and $R(\vec{x_R})$. Note that, the statements S and R need not be distinct, only the dynamic instances $S(\vec{x_S})$ and $R(\vec{x_R})$ must be different.

2.1.6 Dependence vectors

If $R(\vec{x_R})$ depends on $S(\vec{x_S})$, or simply denoted $R(\vec{j})$ depends on $S(\vec{i})$, one has that sequentially $S(i)$ is executed before $R(j)$. In *loop dependence analysis* [8], this is equivalent to saying that iteration j of loop L depends on iteration i, where L contains the statements S and R. Then, R depends on S with:

- the *distance vector* $\vec{d} = \vec{j} - \vec{i}$;
- the *direction vector* $\sigma = \mathbf{sig}(\vec{d})$;

  The sign of an integer i, denoted σ is

  $$sig(i) = \begin{cases} 1, \ if \ i \ > \ 0, \\ -1, \ if \ i \ < \ 0, \\ 0, \ if \ i \ = \ 0. \end{cases}$$

  The sign of a vector $\vec{d} = (d_1, d_2, ..., d_m)$ is sig($\vec{d}$) = (sig(d_1), sig(d_2), ..., sig(d_m));
- at level $\mathbf{l} = \mathbf{lev}(\vec{d})$.

  Given that m is the depth of the loop L, for a distance vector $\vec{d} = (d_1, d_2, ..., d_m)$, the *leading element* is the first non-zero element. If this is d_l, then l represents the *level* of $\vec{d}$ and $1 \leq l \leq m$. Also, the level of the zero vector is defined to be $m + 1$. The vector d is said to be *lexicographically positive* or *negative* if its leading element is positive or negative, respectively.

A distance vector or a direction vector of a dependence must always be lexicographically non-negative. Thus, considering the dependence vector $\vec{d}$ at level l, of statements S and R, $l \in \{1, 2, ..., m + 1\}$. If $1 \leq l \leq m$ we say that the dependence of R on S is carried by the loop of depth l. The dependence of R on S is loop independent (not carried by any loop) if $l = m + 1$, equivalent to $\vec{d} = \vec{0}$.

Only loop-carried dependences between statements contribute to the computation of dependences between iterations.

Using the distance vectors, one can compute the *dependence matrix* of the loop L, whose rows are the distance vectors of all the dependences in L, in any order.

2.1.7 Polyhedral transformations

Once the dependences between iterations are computed, one can apply polyhedral transformations (schedules) such that no dependence violations occur. Such transformations are applied in the view of improving the performance, for instance via better data locality, or to exhibit parallelism. Transformations represent a reordering of the execution of instructions and are defined as scheduling matrices Θ^S associated to statements.

Frequently, aggressive optimizations inhibit parallelism, due to loops fusion, strength reduction, replacing arrays with scalars etc. To overcome this problem, one requires to apply optimizations such as scalar expansion, induction variable detection, loop splitting and skewing, to recover the parallelism opportunities. Selecting the optimal sequence of transformations opens an entire area of research, notorious for being a difficult combinatorial problem. We refer the reader to the works of Kennedy and Allen [2], Wolf and Lam [133] proposing loop transformations to maximize parallelism. In the geometrical frame of the polytope model [44, 45], one can generalize and handle all affine transformations in a unitary manner.

To validate a schedule θ one computes the scalar product between each of the transformation matrices Θ_S and the dependence matrix. If in the resulting matrices, the first non-null component of each row is positive, the schedule is valid. The outermost parallel loop level is given by the first column in the resulting matrix in which all elements are null.

2.1.8 Concluding remarks

The polytope model is a robust and efficient framework for statically analyzable, affine codes. However, these constraints restrict its applicability to scientific codes mostly. To fight against the limitations of the polytope model, several research directions revolve around addressing the problem of scalability, as a trade-off that reduces optimality by splitting the program and introducing approximations to increase scalability [126]. Another very well researched area targets vectorization: programs are transformed in order to generate efficient SIMD code [55]. The works of Cavazos *et al.* [23, 92] are dedicated to find the optimal sequence of transformations to be applied in an iterative compilation framework. Extending the polytope model to general purpose codes, non-statically analyzable, is a challenge, as computing dependences online, validating a transformation and generating parallel, optimized code at runtime might incur considerable overhead. Still, this outlines our goal of dynamically optimizing general purpose codes, by applying polyhedral transformations in combination with speculative parallelization.

2.2 General overview of traditional TLS systems

As dynamic transformations rely on predictions in case of non-statically analyzable codes, significant research efforts have been devoted to building speculative frameworks, which allow optimistic optimization and parallelization of codes. Known under the

name of *TLS systems (Thread Level Speculation)*, they provide a mechanism to execute speculatively transformed code and a recovery strategy, that guarantees the correctness of the results, should a misprediction occur.

The overview of a traditional TLS system is depicted in Figure 2.3. It consists of two main phases: a first phase performed at compile-time (static), and a second phase performed at runtime (dynamic). The first phase consists of analyzing the code in order to extract general and relevant properties, as data dependences that can be disambiguated without requiring runtime information. It is also the place where, for some TLS systems, the targeted code snippets are prepared for runtime parallelization. Some systems also include a preliminary profiling step in order to be helped by some dynamic information at compile-time, provided that the same behavior will occur during the real execution.

At runtime, often the target code is profiled on some execution samples in order to capture dynamic information relevant for parallelization. Classically, dynamic dependence analysis is performed, as well as some modeling of values taken by scalar variables, or values of memory addresses that are accessed. This modeling is usually dedicated to building a prediction mechanism to remove dependencies and predict the starting values used by the parallel speculative threads. In our proposal, we use multivariate linear interpolation to model the behavior observed during online profiling.

Information collected and computed from this online profiling can then be used to apply code transformations supposed to improve the forthcoming parallelization. In most TLS systems, this step is reduced to straightforward transformations, as simply cutting the outermost loop in slices that will be run in parallel. Our proposal distinguishes from previous proposals by providing elaborated scheduling transformations, as it will be shown in the rest of this thesis.

Then, during the parallel run of the code, speculation is verified in order to ensure correct semantics. Such verification usually consists of detecting conflicting accesses to the referenced memory locations. In our proposal, this is achieved by comparing each actual memory access to the predicted one. In case of misspeculation, the incorrect computations have to be canceled and the memory restored to a correct state. Usually, this is achieved by re-executing the faulty thread for which a conflicting memory access has been detected. In our proposal, the global orchestration, that executes the target loop nest by successive chunks, implements the rollback as the cancelation of the last executed chunk. Then, a chunk containing the original sequential version of the code is run to overcome the misprediction point.

In the next section, we reference some of the most representative works on static-dynamic platforms for code analysis and optimization and for speculative parallelization.

2.3 State of the art

There has been a considerable amount of research in TLS systems which propose speculative execution of threads, sometimes requiring architectural support, relying on hypothetical hardware mechanisms and simulators [63,76,90,103,105]. Since our proposal is a *software-only* framework, our reviews focus mostly on works of this type.

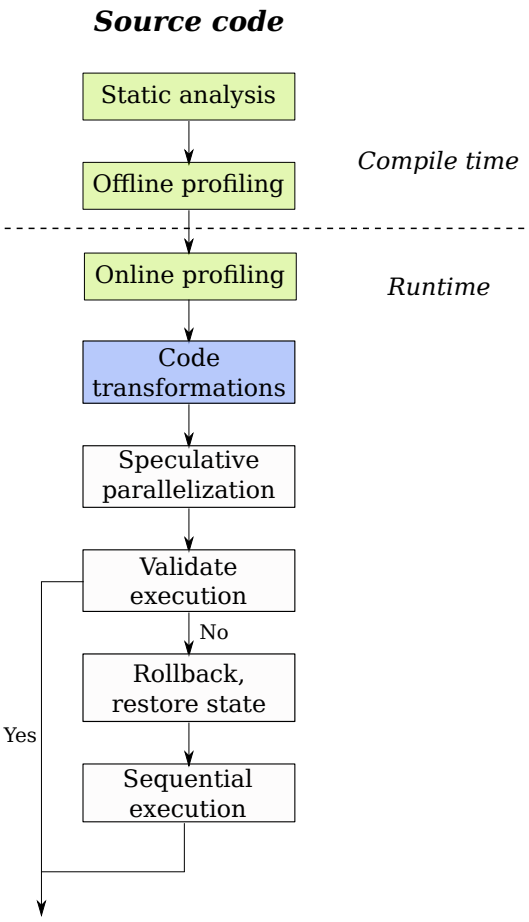


Figure 2.3: Overview of a TLS system

Automatic parallelization One of the early proposals exploring TLS systems is [114] targeting an approach that scales on various shared-memory architectures. Since TLS requires to detect dependences violations at runtime, they propose to leverage the invalidation-based cache coherence. According to this method, the processor must first invalidate all copies of a cache line, prior to modifying it. Steffan *et al.* propose to extend this strategy and adapt it for TLS systems, by invalidating copies that were speculatively loaded in the past, in case the modification request comes from an iteration that executes before the ones that use the copies, in the sequential order.

POSH [76] is a compilation framework for transforming the program code into a TLS compatible version, by using profile information to improve speculation choices. A similar approach is presented in [63]. The Mitosis compiler [103] generates speculative threads as well as pre-computation slices (p-slices) dedicated to compute in advance values required for initiating the threads. The LRPD test [106] speculatively parallelizes forall loops that access arrays and performs runtime detection of memory dependences. Such technique is applicable only when the array bounds are known at compile time. Tian *et al.* [119] focus on the efficient exploitation of pipeline parallelism using a data speculation runtime system which creates copies of statically, as well as dynamically allocated data, on-demand. Similar to [104], this study handles only single-level loops.

SPICE [105] is a technique using selective value prediction to convert loops into data parallel form. A similar approach is proposed in [117]. In [28], a speculative parallelization in chunks of the outermost loop is proposed, using a sliding window for reducing the impact of load imbalance. However this last proposal is limited to array-only applications.

Softspec [19] is a technique whose concepts represent preliminary ideas of our approach. Linear memory accesses and scalar values sequences are detected, resembling our strategy, but only for innermost loops. Hence one-variable interpolating functions are built and used for simple dynamic dependence analysis via the *gcd* test. Thus, only the innermost loop can be parallelized. However, their initialization and verification mechanisms are similar to ours.

Zhong *et al.* present in [137] several code transformation techniques to uncover the hidden parallelism and improve TLS efficiency, as speculative loop fission, infrequent dependence isolation or speculative prematerialization.

Kulkarni *et al.* propose the system Galois [70] to perform speculative auto-parallelization of codes with irregular data access patterns. Inspired from database applications, in which parallelism is fully exploited, while the implementation details are transparent to the user, they define a new abstraction for the sequential programs. Galois provides new classes, such as the Galois sets and the Galois iterators, which programmers can use to highlight opportunities to exploit parallelism. Parallelization relies on a form of "scientific commutativity", i.e. non-deterministic results, yet still correct. The underlying operations of the speculative system (commutativity, rollback) are defined by descriptions given by the developer. For commutativity, the programmer annotates the code to emphasize whether the execution of different methods can be interleaved; while the rollback mechanism performs a copy of the memory area from which it restores the state in case of a misspeculation. The Galois system consists in the classes

with different manners to acquire the lock, *catch-keep* and *catch-release* classes and a runtime system. In the case of *catch-keep* class, the lock is acquired in the beginning of each iteration of a loop and released in the end. However, the drawback is that if an iteration is unsuccessful in obtaining a lock, because another one owns it, one of the two iterations must rollback. To overcome this, the catch-release class acquires the lock in the beginning of the iterations, per method, such that the lock is released as soon as the method finishes, allowing interleaved execution of different methods. The system is evaluated on two real-world applications, well known for their difficulty to be auto-parallelized: a Delaunay mesh refinement application and a graphics application that performs agglomerative clustering, showing promising results. We target similar codes, which cannot be statically analyzed, yet we identify linear patterns, such that the entire process of auto-parallelization is transparent to the user.

The work of Tian *et al.* [117, 118, 119, 120, 121] explores speculative parallelization and focuses on the *copy or discard* execution model. It relies on a main, non-speculative thread plus several additional threads that run in parallel, but commit their results in order. As soon as a thread is detected as incorrect, its output is simply discarded, hence no rollback mechanism is required. A particularity of their work is the partitioning of a loop iteration in three sections – the prologue, the speculative body, and the epilogue. The prologue and the epilogue contain statements dependent on the preceding iteration, whereas the speculative body consists of statements with a high probability of being independent, thus can be executed in parallel. Additionally, they developed algorithms to optimize the communication between the threads. The entire system relies on PIN [79] as an offline profiler, whilst the code is prepared at compile time with LLVM [77]. The results show speed-ups between 3.7 to 7.8 on 8 cores.

Another proposal for handling irregular codes is given by Paralax [125], an infrastructure for automatic paralellization relying on annotations from the programmer. The compiler is helped to extract thread-level parallelism and uses the annotations for a simplified verification model. These annotations are proposed to the programmer by a dedicated tool.

An example of work targeting automatic speculative parallelization, called ParExC [115], refers to code that has been optimized at compile time, but it abounds in runtime checks. This is specific for codes that cannot be disambiguated statically, thus the compiler generates self-healing code, which adds some tests and adapts to dynamic contexts. However, even optimized code can register a slowdown, due to numerous checks. The solution they propose is to parallelize solely the runtime checks, rather than the entire application. Moreover, their technique relies on the strategies employed by traditional TLS systems and in particular, they use a transactional memory-based alternative to guarantee the correctness of the code. ParExC speculates on a failure free execution and aborts as soon as a misspeculation is encountered. Similar to TLS systems, they use speculative variables to decouple executors that access a shared state.

Recent works of Kim *et al.* [66] describe the fragility of static analysis, pleading for speculative parallelization. They present a speculative DOALL system, targeting automatic parallelization on clusters, by speculating on some memory or control dependences. Statically, a PDG (program dependence graph) is built, from which some dependences are speculatively removed. Regarding the control edges, the system counts

how frequently an edge is taken, with respect to the number of iterations of the loop. Given a certain threshold, all edges that are executed less frequently are speculatively removed from the PDG. Similarly, for memory dependences, the flow between the load and store instructions is computed and the same strategy is applied as in the case of control edges. Once the PDG is speculatively simplified based on the threshold, the code is automatically parallelized. Verification relies on transactional logs and is supported by rollback and recovery mechanisms. The SPEC DOALL system executes a master process, non-speculative, and several speculative worker processes. To improve communication between the nodes, they make use of the Copy on Access mechanism, which loads the memory pages at request. The page fault handles must be rewritten to load a page which was not already mapped. All in all, the tool targets automatic parallelization for clusters, optimizes communication between the cores and focuses on scalability. Nevertheless, SPEC DOALL is highly representative for our work since is shows how sensitive static analysis is to the implementation style. For this purpose, they evaluate the unmodified PolyBench benchmarks [95] and compare it to a version in which the static arrays have been replaced with dynamically allocated arrays. This prevents many compile-time optimizations and leads to many missed parallelization opportunities, emphasizing the urge for speculative parallelization. However, although the Polybench benchmarks are suitable for advanced loop optimizations, the SPEC DOALL tool performs simple, straightforward parallelization only.

Dynamic loop optimizations Similar to speculative parallelization, the purpose of dynamic optimizations is to allow transformations adapted to the current execution behavior. Hence it relies on a phase of dynamic profiling, followed by code transformations applied on-the-fly. The most common approach for performing dynamic optimizations is to use JIT compilation [7, 47, 69]. While dynamic transformations offers a solution to codes which cannot be statically optimized, applying software dynamic translation could be too costly, especially on codes with a changing behavior. Until now, limited progress has been reported on complex, dynamic optimization of loop nests, due to heavy processes that must be performed at runtime, such as data dependence analysis and code generation. Nevertheless, it is our goal to address these problems.

An interesting proposal targeting dynamic optimization is the work of Luo *et al.* [81], which promote dynamic performance tuning in TLS systems. Not only extracting efficient speculative threads is made difficult by complex control flows and ambiguous data dependences, but the performance of speculatively parallel code is often sensitive to the characteristics of the underlying architecture, to the input data or the code may even exhibit different phases during one execution. To address these problems, they propose a mechanism able to adapt dynamically to the execution context, by changing the optimizing strategy at runtime to generate better performing code. This proposal is particularly of interest in the context of our work, as it targets code that exhibit several execution phases and adapts to each of them dynamically. In contrast, they rely on hardware support to monitor the threads and estimate their performance.

One of the main difficulties in performing dynamic loop optimizations consists of validating the transformation, especially when aggressive loop optimizations are ap-

plied, such as loop fusion or fission, tiling, which alter the loop structure. The work of Zuck *et al.* [139] perform a formal proof after every run of the compiler, verifying the semantics of the transformed code. However, their model is limited to loops accessing memory through statically analyzable arrays. Some preliminary work on validating speculative optimizations of loops is presented briefly. They insert runtime tests whose results are used either to modify the behavior of the optimized code, such that correctness is preserved while maintaining the performance benefits of the optimizations; or to jump to an unoptimized version. However the work is still preliminary and provides some examples only, rather than a formal proof.

Transactional memory To ensure data race free code, a common approach in TLS systems is using transactional memory, which allows a group of read and write operations to execute atomically. Nevertheless, TM systems are notorious for the high overhead they introduce.

Thus, the work of Adl-Tabatabai *et al.* [1] develops compiler and runtime optimizations for transactional memory constructs, using JIT. Static optimizations are employed to expose safe operations, such that redundant STM operations can be removed, while the STM library interface is tailored to handle JIT-compiled and optimized code.

STMlite [85] is a tool for light-weight software transactional memory, dedicated to automatic parallelization of loops, guided by a profiling step. One of the contribution of STMlite is that it reduces the overhead of traditional TM models, by decoupling the commit phase from the main transaction execution.

Raman *et al.* [104] propose software multi-threaded transactions (SMTXs), which enable combining speculative work and pipeline transformations. SMTXs use memory versioning and separate the speculative and non-speculative states in different processes. Thus, all processes share the same view of the virtual address space, but each process can alter only its privatized memory area. This is transparently handled by the system using copy-on-write semantics. Misspeculations are managed by remapping the virtual address space to the committed memory state. STMX has a centralized transaction commit manager and conflict detection is decoupled from the main execution.

However, although TM is considered to be a promising solution for multi-threaded programming, in [22], Cascaval *et al.* present the STM model as a research toy, due to its high overhead. They explore the performance of a highly optimized STM system, but show that at low levels of parallelism, performance is significantly reduced. By evaluating independently the overhead of different components, the findings show that the elimination of dynamically unnecessary read and write barriers could bring considerable benefits. Yet, optimizing STM for reducing its overhead is still a challenging problem.

Software value prediction Value prediction is required by TLS systems to minimize sharing of data and to start ahead-of-time execution of threads. An example of work revolving around the aspect of software value prediction (SVP) for TLS systems is [75]. Li *et al.* spawn threads using the fork utility for loop parallelization, execute the current iteration in the main thread and speculatively execute the next one

in the forked process. They try to predict the values required by the spawn thread, initialize the variables such that no dependence violations occur and periodically check the speculations to detect dependences. Their contribution consists in the predictor, which is optimized to profile only critical variables, detected by the compiler. Additionally, if more than one such variable exists, they propose a *prediction plan* because the variables might be dependent one on another. Their work is compared to the classical predictors such as "constant-stride" predictors, "last value", "indirect-predictor" or "code reordering" predictors and evaluated from the point of view of input data sensitivity. The finding is that, most of the times, the critical variables are represented by scalars or pointers which can either be deduced based on a certain pattern or can be predicted using the last value predictor. Since SVP is a typical instrument for TLS systems, several other works developed strategies to predict the values of the shared variables [37, 100, 117].

Dependence analysis An important aspect in speculative systems concerns detecting incorrect executions, which typically reflects in violations of dependences. Since loops represent the main candidates for speculative parallelization, a well-researched direction targets dependence analysis on loops, and particularly proposing algorithms for *dynamic* dependence analysis [68, 82, 89, 123, 129, 134, 136].

As a dependence profiler tool, Alchemist [136] is designed to identify dependences across loop iterations, loop boundaries and methods. It can be used offline by speculative systems, as it provides a very precise dependence analysis, analyzing complex data. Nevertheless, it induces a large overhead and it is not aimed for a runtime usage. It is specifically designed for the *future* language constructor, managing the results of asynchronous computations.

SD^3 [68] is a proposal for performing dynamic dependence analysis, being the first to target reducing both the time overhead and the memory footprint. Thus, the profiling phase is parallelized to incur a minimal time overhead, whereas several compression algorithms are used to reduce the stored data. For computing the set of dependences at runtime, Kim *et al.* define a *history table*, storing all memory accesses until the previous iteration; and a *pending table*, which contains the memory accesses performed in the current iteration. Next, all dependences between the pending and the history accesses are computed, and the algorithm is refined to distinguish between loop carried and loop independent dependences. The output of SD^3 is that it provides suggestions to the developer on which modifications are desirable, such that the code becomes suitable for parallelization. The tool is built on top of PIN [79], but it is parallelized to reduce the overhead. Nevertheless, SD^3 shows a 70× slowdown on average, 29× and 181× in the best and worst cases, respectively, when running on 8 cores. The tool targets a precise dependence analysis, without sampling. Due to its high runtime overhead, SD^3 cannot be embedded in a TLS system. To address the input sensitivity problem, the authors provide a measurement of similarity between dependences discovered when executing the code with different inputs. Tested on the OmpSCR benchmark suite [91], the tool proved to be rather stable, as dependences do not vary much.

In [89], Oancea and Mycroft propose a dynamic analysis based on congruences of

sets. The approach starts by building a pattern to predict the memory accesses and to compute dependences. Next, it generates a dependence-pattern represented as a directed acyclic dependency graph (addg), in which nodes are iteration numbers, and edges are directed from dependence's source to sink and are annotated with the type of the dependence (WAR, RAW, WAW). The goal is to map dependent iterations on the same thread, such that no dependence violations occur. Mapping is based on congruences of sets, computed from the dependence pattern.

Cost models In [134], Wu *et al.* provide a cost model to guide the compiler in selecting tasks for TLS systems. The model is based on analyzing *may-dependences* and estimating the probability of successful speculations. The compiler uses an auxiliary profiling tool, Dprof, for detecting *may-dependences* at runtime and for mapping the dynamic dependences on the source code. One of the metrics used in evaluating the cost is the distance between two consecutive dependent iterations of a loop, which is named *independence window*. The name comes from the fact that if this value is larger than the speculation window, no dependence violation occurs, thus the effectiveness of the TLS system is not influenced. At compile time, Dprof decides which *may-dependences* to instrument. For a minimal time overhead, no profiling is attempted on induction, reduction or private variables, since dependences carried by these type of references can be eliminated by suitable compiler transformations. Additionally, loop carried dependences which can be represented as dependence distances do not require instrumentation. At runtime, Dprof logs all read and write accesses and computes the dependences. The output takes the form of a feedback given to the compiler to select transformations which do not violate dependences, or to the programmer to re-structure the code to exhibit parallelism.

Static polyhedral optimizations Since the polyhedral model is suitable for modelling loops and shows promising results for loop optimizations, many tools applied it for performing automatic optimizations and parallelization. Among them, the most notable are Pluto [15], Polly [55] and Chill [108]. However, in its original form, the polyhedral model requires a linear description of the memory accessing pattern of the code, hence, it is limited to codes exhibiting this behavior. Additionally, since the transformations might incur a considerable overhead, the model is particularly suitable for statically analyzable programs.

Pluto [15], as described in Section 1.2, is an end-to-end transformation tool, that automatically selects good transformations to improve both coarse-grained parallelism and cache locality. It uses a powerful cost function to find affine transformations that enable tiling, fusion or other optimizations of nested loops.

Polly is a tool for automatic optimization and parallelization of loops, improving data locality and performing automatic parallelization for thread-level and SIMD parallelism. It handles code in the LLVM intermediate representation, therefore it can be seen as an internal optimizer of LLVM. Polly relies on external tools for performing the main steps towards code optimization: ISL [127] for computing dependences, Pluto/PoCC [15] for selecting a valid transformation and Cloog [9] for code generation. Additionally, it can retrieve the transformations to be applied by taking as input

manually edited files. Hence, Polly is able to overcome the limitations of the external tools, using more complex polyhedral descriptions defined as JSCoP files. The originality of the framework stands in including the well-known polyhedral optimizations into the intermediate representation of a compiler, making Polly independent of the programming language. The advantage is that Polly is easy to use and transparent to the user.

Chill [108] is a source-to-source compiler targeting C/C++ code, similar to Pluto. The user must provide a script with the list of polyhedral transformations to be applied, following a specific API. The scripting language proposed by Chill allows composable compiler transformations, and outputs readable code. From the parallelism perspective, Chill transforms sequential loops into high performance GPU code, outperforming hand-tuned examples.

All in all, currently, these tools offer support only for statically analyzable codes, whereas our target is to apply the polyhedral model on general-purpose codes.

2.4 Limits of current TLS systems

Consistent research has been devoted to improve various components of the traditional TLS systems: offline and online profilers [68, 82, 89, 123, 129, 134, 136]; value predictors [19, 37, 75, 100, 105, 117]; verification and rollback mechanisms [117, 118, 119, 120, 121]. Although each component contributes to the overall performance of the system, we consider equally important to improve the performance of the generated code. We emphasize the main aspect that limits the performance gains which can be obtained following the traditional TLS approach: *reduced code transformations*, leading to:

- inefficient parallel code and
- missed parallelization opportunities.

Improving the support for more complex code transformations is a challenge, due to the characteristics of the TLS systems.

Any speculative system requires to verify all along the execution of the parallelized code that it stays semantically correct, in order to validate or invalidate its execution. In TLS systems, a given thread can be responsible of a suspicious memory or register access that has to be canceled by performing a rollback to a previous safe state.

When the parallelization strategy consists of slicing a loop in contiguous parallel chunks, each chunk being run by a different thread, verification is achieved by directly comparing the memory behaviors of the parallel and sequential versions, the latter, obviously, being the baseline. It consists of monitoring the memory and the register accesses of the speculative threads, in order to verify if accesses made by different threads to the same memory locations occur in an order which is different than the original sequential one. Since the i-th thread executes code that would be run before the code of the $(i+n)$-th thread in the sequential version, any write access from the $(i+n)$-th thread to a common memory location and occurring before any access from the i-th thread generates a rollback of the $(i+n)$-th thread in the parallel version. This mechanism is illustrated in figure 2.4.

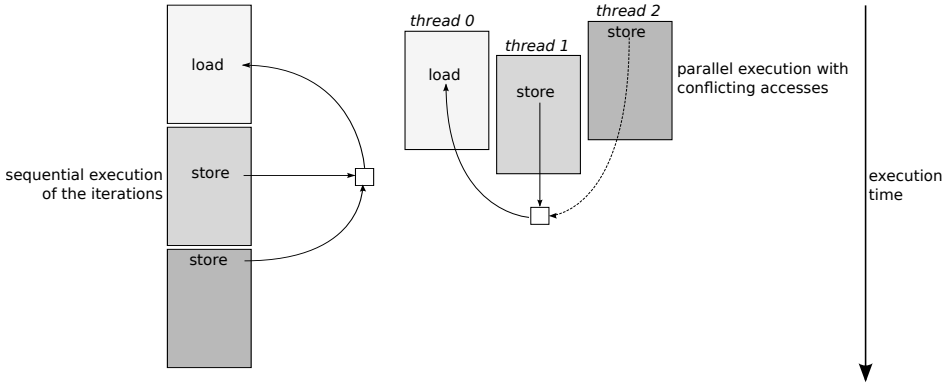


Figure 2.4: Illustration of the usual TLS parallelization by chunks and conflict detection (courtesy: Ph. Clauss)

This verification scheme is straightforward, since the parallel execution can be directly mapped on the original sequential order.

Significant improvements have been reported for this parallelizing strategy, in order to apply to a larger set of codes. For instance, value prediction is used when a dependency occurs between threads, such that each thread has a better chance to be validated at the end of its run [105, 117]. However, even with such improvements, it is always the same parallelization strategy which is applied: contiguous chunks executed in parallel.

This is mostly due to the fact that if the original code is transformed by significantly re-scheduling its instructions, it becomes extremely difficult to identify any conflicting memory or register accesses, since the parallel execution order can no longer be easily mapped on the original sequential order. Additionally, to perform advanced code transformation, one requires to perform *dynamic dependence analysis* to ensure the semantics of the transformed code remains correct.

Program analysis theory for code parallelization provides the well-known notion of dependences between instructions. In parallel programming or compile-time automatic parallelization, any code transformation is semantically correct if all dependences are respected. Namely, two dependent instructions executed successively in the sequential code must preserve their execution order in the transformed code.

In speculative parallelization, dependences are unknown before the code has been run. However, any parallelization is considered as "valid", since any dependence violation is recovered thanks to the rollback. But too many dependence violations induce a huge overhead from the numerous rollbacks. Hence it is essential, if possible, to get dependence information as soon as possible to generate efficient parallel code.

A previously proposed approach is to run in advance a subset of the code which performs just the memory address computations, in order to get information regarding all dependences. This idea is used in the inspector/executor model, where sequential programs are divided in two components. The first one, called the inspector, is in charge of extracting the program dependences, usually between loop iterations. Then, an executor runs the tasks as soon as all their dependences have been computed. This model has been first proposed by Zhu and Yew in [138], and has been later extended

in many directions [25, 86, 106].

In general, this model is efficient if the address computation is clearly separated from the actual computation, allowing the creation of an efficient inspector. Moreover, to capture the dependences with no restriction, some control bits are commonly associated to every array element during the inspector phase. This often restricts those methods to array accesses, and can lead to major memory overheads. Moreover, pointer references can strongly disturb the automatic inspector creation, limiting the applicability of this method.

Hence, very often it is impossible to validate a code transformation in advance. Following the general idea of verification in speculative systems, the transformation should be verified while the transformed code is executed.

Code transformations are usually guided by sufficient knowledge of the code properties and by objectives like data locality optimization, energy saving or load balancing. In the speculative approach, transformations should also be guided by predictions on initially unknown code properties, and particularly dependences between instructions. Thus, the open question for speculative parallelization is: how to predict, with the best possible success rate, dependences that cannot be determined at compile time?

Any prediction process has to be based on some observations on how the code behaves, at least during a sample of its execution. Moreover, the observed behavior has to be used to feed the prediction mechanism. A representation model is required for this purpose. There is an important literature about prediction mechanisms, and many of these proposals have been, or could be, experimented to model the memory behavior of a code. However, in the case of advanced speculative parallelizing and optimizing transformations, the employed model has to provide enough information to guide the transformations. Moreover, it must provide the means to build the required verification system, preventing dependence violations, and the associated rollback mechanism.

To enhance the traditional strategy and target high-performance parallel code, we propose a challenging and promising new paradigm for speculative parallelization.

Chapter 3

From polyhedral transformations to speculative parallelism

This chapter emphasizes one of the key contributions of our proposal, focusing on applying polyhedral transformations at runtime, in the view of speculatively parallelizing a loop nest.

We propose to use the polytope model [15, 44, 45] which we adapted for a dynamic usage. Known also as the *polyhedral model*, it provides powerful mathematical theory to represent loop nests and to reason about the correctness of complex loop transformations. The polyhedral model is dedicated to loop nests in which the memory access functions and the loop bounds are affine functions of the enclosing loop iterators. Particularly, it is applicable on statically analyzable codes (static control structures, affine bounds and memory references), as it requires an exact characterization of data dependences. A detailed description on how the model is applied in a research compiler is given in [16]. It consists of three main phases: (1) static analysis of the code; (2) transformation in the polyhedral model; (3) code generation for the transformed program. The static analysis relies on mathematical mechanisms from linear algebra and solving of integer equations. Applying transformations relies on geometry, while code generation represents the translation of the polyhedral abstraction of programs to an equivalent code. Empirical tests show that polyhedral optimizations only, without parallelization, can lead to speed-ups of up to 14x [98], solely by improving data locality. However, applying the traditional methodology at runtime would be inefficient because it is extremely time-consuming. To benefit from such a powerful optimization model, we propose a light-weight system, able to fill-in the missing information at runtime, to perform polyhedral transformations dynamically and to automatically parallelize loops. This chapter introduces several motivating code examples, followed by an in-depth presentation of the adaptation of the polytope model for dynamic analysis, dynamic transformations and finally for speculative parallelization.

3.1 Can general-purpose codes be modeled in the polyhedral abstraction?

It is well known that many general-purpose code are suitable for parallelization, although these opportunities are hidden to the compiler due to dynamic memory allocation, presence of parameters that cannot be evaluated at compile-time, dynamic control structures, indirect references, pointers etc. Similarly, many of such codes are eligible to be modeled in the polyhedral abstraction, however, the required information is partially missing at compile-time. Before revealing our techniques for building the polyhedral representation of such programs, we present several kernels from the realm of general-purpose codes, whose performance would be greatly improved by a polyhedral transformation.

3.1.1 Indirect references

Consider the sequential loop nest in Figure 3.1(a). Array `a` is write-accessed using an indirect reference through array `b`. Hence it cannot be decided at compile-time what output dependence is induced by this memory reference. If values contained in array `b` are all different, then no dependence occur since distinct elements of array `a` are updated. In this case, the outermost loop, loop `i`, can be parallelized and no rollback is required. Therefore, the compiler may generate a parallel code guarded by tests on the values in `b`. However, if several elements of `b` are equal, this parallel solution may not be efficient due to many rollbacks. For example, consider the following contents of array `b` (in row-major order): `b[N][N] = {{0..N-1}{0..N-1}{0..N-1} ...}`. In this case, for each value of index `i`, all elements of array `a` are updated. Hence an output dependence occurs, carried by the outermost loop, loop `i`. Parallelizing this loop by contiguous chunks would induce a high number of rollbacks, one for each iteration (except in the thread computing the first iteration of the loop `i`). Nevertheless, notice that for a fixed value of index `i`, loop `j` updates distinct elements, consequently it does not carry any dependences. Thus, a better strategy would be to parallelize the innermost loop. However, this solution may induce too many synchronization barriers leading to poor performance.

A more suitable solution is to exchange both loops as shown in Figure 3.1(b). It allows parallelizing the outermost loop, which is now loop `j`. However, the drawback is that array `b` is now accessed in column-major order. This induces poor data locality and many cache misses. An improved solution would be to tile the loop as shown in Figure 3.1(c), such that the data accessed inside the tile fits in the cache. Notice that the tile size, `tile_size`, is also a parameter that may be adjusted at runtime to reach the lowest execution time.

But even more complex program transformations may be useful. Consider now the following content of array `b`: {{0..N-1}{N-1 0..N-2}{N-2 N-1 0..N-3}...}. In this case, neither loop `i` nor loop `j` should be parallelized due to the numerous induced rollbacks. A better solution is to transform the loop nest as illustrated in figure 3.1(d). In this new version, distinct iterations `i` access distinct elements of `a` and therefore the outermost loop can be parallelized without inducing any rollbacks. Moreover, array `b`

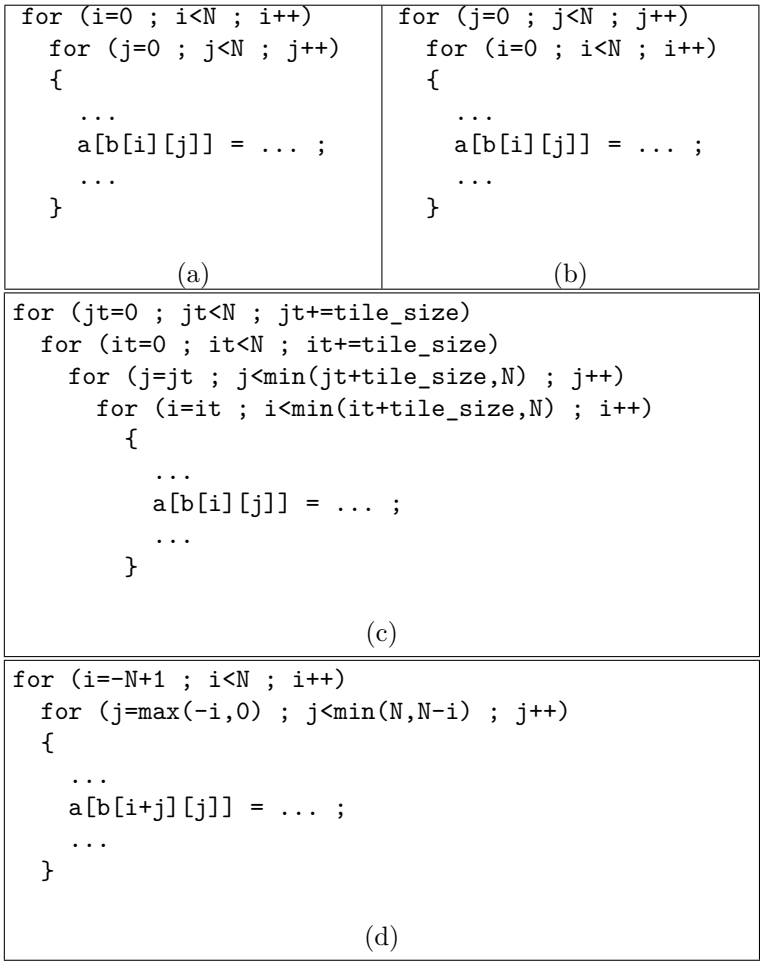


Figure 3.1: Examples of loop nests with an indirect memory reference

is accessed in row-major order in the innermost loop which implies good data locality.

All these considerations argue for more advanced parallelizing transformations strategies and more flexibility, in order to adapt at runtime to the current memory access behavior. Since indirect memory accesses can only be resolved at runtime, the parallel code can only be speculative. But it will be better adapted to the current context if it is generated at runtime, using profiling information collected on some initial execution samples. In most of the above examples, the memory access behavior can be fully captured by profiling the first few iterations of the loop nest, and by performing linear interpolation of the accessed memory addresses.

However, rollbacks still occur if the behavior of the memory accesses observed during the profiling phase changes at a given point of the loop nest execution. Nevertheless, any rollback can trigger another profiling phase in order to capture the new memory access behavior and generate a newly adapted parallel code.

This accounts for our strategy of applying the polyhedral model at runtime, on general-purpose code, as illustrated in what follows on several representative examples.

```
(1) start list with spaced chunks
******___******_******_____******____****** ...

(2) continue list with equally spaced elements
*___*___*___*___*___*___*___*___*___*___*__ ...

(3) continue with random spacing between elements
_**___*_*_*__*___**__**___***__**___****_** ...
```

Figure 3.2: Memory allocation for the linked list

3.1.2 Sparsely allocated linked list

This example handles a large linked list allocated in memory in chunks and searches for an element of a given value. The memory areas reserved for the linked list, marked with '*', follow the pattern shown in Figure 3.2.

Listing 3.1: Code parsing the list in Figure 3.2

```
curr = head;
while(curr){
    computation on the value of the current element
    curr = curr->next;
}
```

The first pattern may be recognized as a consequence of eliminating some elements from a list allocated contiguously. Parallelization can bring high benefits particularly when the processing of elements is computational intensive. Each of the chunks may be executed by a set of parallel threads, provided that the dependency analysis allows it.

The memory allocation exhibited by the second part of the list may appear when several data structures are allocated consecutively in the same loop. The elements, although not in contiguous memory areas but equally spaced, follow a new linear function, thus the chunk can be parallelized accordingly.

Finally, the third part of the list contains elements which are randomly distributed, possibly after many eliminations. This does not allow the parallelization of the chunk, nevertheless, it does not prohibit the parallelization of the previous chunks either.

The purpose of this example is to exhibit the need of a mechanism that supports parallelization of parts of the loop nest, when the loop cannot be completely parallelized. In contrast to previous TLS approaches which would apply a rollback as soon as the memory access behavior changes, our framework is designed to be more adaptive and to alternate sequential parts and *several parallel parts* of the loop, during one execution of the loop. Alternating several parallel schedules for different loop slices, during the same execution is a technique explained in Section 3.3.1 and in Chapter 4, Section 4.3.

3.1.3 Transpose matrix with parameters

This benchmark is extracted from the OmpSCR benchmark suite [91] (fft6) and it computes the transpose of a matrix. Both input and output matrices are dynamically allocated and the accesses are linearized, which prohibits automatic parallelization at compile time. The source code is listed in Listing 3.2.

The presence of the parameter N hinders compile-time optimizations. In contrast, the main advantage of capturing the behavior of the loop at runtime is that the parameter N is identified as being constant. This information allows one to safely apply polyhedral transformations.

Listing 3.2: Transpose matrix with parameters

```
for (i = 0; i < N; ++i) {
  for (j = i; j < N; ++j) {
    b[i * N + j] = a[j * N + i];
    b[j * N + i] = a[i * N + j];
  }
}
```

A straightforward parallelization of the outermost loop would yield a speed-up, as handled by most TLS systems. In addition, the contribution of our proposal is that not only it identifies the parallelizable loops, but it also applies code transformations to enhance performance. On this particular example, a suitable and promising transformation is tiling.

3.1.4 Banded matrix with indirect references

This example illustrates a two dimensional matrix B encoding the element occurrences of a banded matrix, linearized in a one-dimensional array A. These occurrences can be represented as a linear function of two variables. Next, the elements of array A are processed inside a nested loop of depth three, by using indirect references A[B[i][j]]. The code snippet is shown in Listing 3.3.

Depending on the elements of array B, one can note that if the elements of A are accessed by following a linear function, the loop nest can be analyzed, but it cannot be parallelized in its original form, since each loop carries a dependence between consecutive iterations. By applying a polyhedral transformation one can generate a dependence free outermost loop. The optimized code is displayed in Listing 3.4. Not only this transformation fully exploits the parallelism exhibited by the code, but it optimizes the temporal locality of the accessed elements. In the sequential version, the element B[k][k] accesses the diagonal of matrix B, based on the index of the innermost loop, which yields very low spatial and temporal locality. Similarly for the element B[k][j], which accesses matrix B in column-major order. In the optimized version, as a consequence of the polyhedral transformation, element B[k][k] becomes B[z][z], depending on the index of the outermost loop, which results in very good temporal locality. Likewise for element B[z][y], depending on the outerloops z and y.

Listing 3.3: Indirect references to a banded matrix (sequential version)

```
for(i=0; i<N; i++)
 for(j=0; j<N; j++)
  for(k=0; k<N; k++)
   A[B[i][k]] += A[B[k][j]] * A[B[k][k]];
```

Listing 3.4: Indirect references to a banded matrix (parallel version)

```
for(z=0; z<=N-1; z++)
 for (y=0; y<=N-1; y++)
  for (x=0; x<=N-1; x++)
   A[B[x][z]] += A[B[z][y]] * A[B[z][z]];
```

Besides parallelization, our goal is also to apply a polyhedral transformation to improve the performance of loops which are already parallelizable in their original form, but could benefit of a better spatial and temporal locality.

We provided this example to demonstrate our strategy of applying polyhedral transformations at runtime, when speculative parallelization is not recommended otherwise, due to numerous rollbacks.

Among other codes accessing memory via indirections, one can focus on sparse matrices operations. Matrix multiplication exhibits parallelism on multiple levels, yielding very good performance improvements. The same benefits can be obtained by parallelizing a block-diagonal sparse matrix multiplication. However, modern compilers cannot automatically detect this opportunity, due to accesses performed via indirections. As noted by Bruening *et al.* in Softspec [19], the Non-Hermitian Eigenvalue Problem Collection [40] contains numerous examples of sparse matrices with non-zero values organized in stripes or in blocks along the diagonal. Computations performed on such matrices are parallelizable, as each stripe or block displays a linear memory accessing behavior. By speculatively executing the loop in parallel, a rollback occurs in the first iteration of each new block. Nevertheless, the benefits of executing each block in parallel overcome this overhead.

3.1.5 Dynamic control structures

Finally, the last example initializes a linked list and a two-dimensional matrix. The linked list is parsed in a loop nest of depth two which processes the elements of the matrix A differently, based on the values taken by the elements of the list. There are eight different cases, as illustrated in Listing 3.5.

The list is initialized such that chunks of consecutive elements of the list have the same property and the memory accesses performed in each case follow a different pattern. Thus, the first *if* branch is executed for a number of consecutive iterations. The code corresponding to the first case displays a set of dependences that allow parallelization by applying a suitable polyhedral transformation. Next, the second *if* branch is executed and a rollback occurs, since the memory accessing behavior

changed. By applying a new polyhedral transformation, another part of the loop can be successfully parallelized. One continues this strategy until the loop execution completes. This code example requires eight different parallelization phases for one single loop nest execution, to exploit fully the parallelization opportunities.

Listing 3.5: Loop with pointers and dynamic control structures

```
    curr = head;
    i = 0;
    while(curr ≠ NULL){
        for(j=0; j<M; j++){

            if (hasProperty1(curr->val))
                    A[i+1][j] += curr->val + A[i][j];

            else if (hasProperty2(curr->val))
                    A[i+1][j+1] += curr->val + A[i][j];

            else if (hasProperty3(curr->val))
                    A[i+1][j] += curr->val + A[i][j+1];

            else if (hasProperty4(curr->val))
                    A[i][j+1] += curr->val + A[i][j];

            else if (hasProperty5(curr->val))
                    A[i][j] += curr->val + A[i][j+1];

            else if (hasProperty6(curr->val))
                    A[i][j+1] += curr->val + A[i+1][j];

            else if (hasProperty7(curr->val))
                    A[i][j] += curr->val + A[i+1][j];

            else if (hasProperty8(curr->val))
                    A[i][j] += curr->val + A[i+1][j+1];

             curr = curr->next;
        }
        i = i++;
}
```

Overall, this example emphasizes both the necessity of parallelizing parts of the loop and of applying different polyhedral transformations on each part, which comprises the main contribution of our proposal.

All the above code snippets are representative of codes whose performance could greatly be boosted by applying polyhedral transformations. However, in the original form, they are not suitable to be modeled in the polyhedral abstraction. The next

sections are dedicated to explaining our technique for building the polyhedral representation of non-statically analyzable codes.

3.2 Adapted polyhedral model as an analysis model: dynamic dependence analysis

The first phase is devoted to the analysis and modeling of the target program. Typically, the polyhedral model relies on a static analysis of loop nests in the following form:

Listing 3.6: Affine loop nest

```
for ( i_0 = low_0; i_0 ≤ high_0; i_0++)
    for ( i_1 = low_1(i_0); i_1 ≤ high_1(i_0); i_1++)
        for ( i_2 = low_2(i_0, i_1); i_2 ≤ high_2(i_0, i_1); i_2++)
            loop body
```

in which the data access patterns and the loop bounds can be described in terms of affine functions of the enclosing loop indices. We extend this model and adapt it for codes which cannot be fully analyzed at compile time. These codes are defined as loop nests where:

- each loop can be either a *for-*, *while-* or *do-while* loop;
- may contain pointers and indirections;
- some particular scalars, called *basic scalars,* take values which can be expressed as affine functions of enclosing loop indices;
- the loop bounds can be written as affine functions of enclosing loop indices;
- the memory access patterns can be described by affine functions on some large enough phases of execution.

Our proposal divides the analysis phase into one *static* and one *dynamic* part. The static analysis provides the linear modeling from the static invariants of the code, whereas the dynamic analysis fills in the missing information, with data discovered at runtime. Non-statically available information is captured by means of an online profiling phase, aimed to monitor and to instrument the data accesses and the loop bounds. To limit the runtime overhead of the profiling, a suitable solution is to perform *instrumentation by sampling.* Thus, only the first few iterations of the loops are instrumented.

During the instrumentation phase, we monitor the values taken by the basic scalars, the accessed memory locations and the number of iterations executed by the subloops. We distinguish two main types of memory instructions:

1. Instructions which do not require to be instrumented in each iteration, because they are almost entirely statically analyzable. They access memory locations following the pattern of an explicit linear function that can be determined at compile-time. Typically, the function can be expressed as $A \cdot I + base\_address$, where A is a vector of integer values (known at compile time), whose size is equal to the number of the enclosing loops, and I represents the vector of the loop indices. As an example, most array accesses can be represented using linear functions. In this case, it suffices to insert instrumentation code for acquiring the *base_address* outside of the loop, which reduces significantly the overhead.

2. Instructions which require full instrumentation, because they access memory locations that cannot be disambiguated statically, by using pointers or indirect references. The accessed memory locations depend on values only known at execution-time. This type of memory instructions require to be instrumented at each iteration.

As mentioned previously, the goal is to align to the polyhedral representation of a loop, i.e. to represent all data accesses and loop bounds as affine functions on the enclosing loop indices, when possible. Since instrumentation is performed during a sample of the total execution, the generated functions interpolate the monitored values and provide the means to *describe* and to *predict* the behavior of the loop.

To be able to handle all types of loops in the same manner, being them *for*-, *while*- or *do-while* loops, we introduce the notion of "*virtual iterators*". They are canonical iterators inserted in the loops, starting from 0 and incremented with a step of 1. Such iterators allow us to handle loops that originally did not have any iterators in the code and to apply speculative polyhedral transformations, based on predictions.

Expressing data accesses and loop trip counts as affine functions make it possible to perform an analysis of dependences between the instructions inside the loop nest.

3.2.1 Dependence analysis

Studying dependences is of utmost importance, as a polyhedral transformation is considered to be valid if and only if all dependences are satisfied, i.e. the original execution order is preserved among the dependent instructions.

Static dependence analysis Given that all data accesses and loop bounds are described by affine functions which can be computed statically, a precise dependence analysis can be carried out at compile time. A step by step introduction and algorithms for dependence analysis are presented in [34]. Using the polyhedral description of the loop, one computes the dependences by applying the mathematical support to solve linear equations. ISL [127] is a dedicated library which provides an implementation for operations on integer sets, required by the polyhedral model. This work is further extended by Grosser et al. [55], to perform automatic polyhedral transformations in the LLVM compiler suite [77], targeting statically analyzable codes.

Dynamic dependence analysis Performing dependence analysis at compile time allows one to benefit from the full mathematical support for solving linear equations, since the time cost is not of major concern. In contrast, at runtime, one must develop

very fast but precise enough algorithms to carry out the dependence analysis during the execution, with an acceptable time overhead.

We tackle this problem by designing an algorithm for the runtime computation of distance vectors. During a sample of the execution (i.e. a few consecutive iterations of each loop), all memory accesses are monitored. They are collected in a table, as the one displayed in Figure 3.3, which maintains information regarding:

- the values of the enclosing loop indices;
- the ID of the instruction performing the memory access (static instance of the instruction);
- the address in memory being accessed;
- the type of access (read (R) / write (W)).

Since the number of memory accesses can be very high, we compute the distance vectors as soon as the information becomes available. The table is updated accordingly, by removing vectors that become redundant for the computation of the distance vectors. Thus, we only store the minimum amount of data. The algorithm is given below:

- every first access to a memory address is stored in the table;
- if a read (R_0) is performed to a location that exists in the table:
  - if $\exists$ a write (W_0), compute the distance vector between R_0 and W_0 (flow dependence);
  - if $\neg\exists$ a write, store R_0;
- if a write (W_1) is performed:
  - if $\exists$ a write (W_0), compute the distance vector between W_0 and W_1; delete W_0; keep W_1 (output dependence);
  - if $\exists$ more R_0, R_1 ... compute all distance vectors between (R_0 , W_1), (R_1 , W_1), ...; delete all R_i; keep W_1 (anti dependence);

For each pair of instructions, with at least one write, if no distance vector was computed for the given pair, we perform the *value range analysis* and the *gcd* test, using the interpolating linear functions of the instructions. For the *value range analysis*, the lower and the upper bound of the iterators are used, as explained in Section 5.2.2. Taking a conservative approach, we consider as independent instructions the ones for which one of these tests guarantees no overlapping, otherwise a dependence must be considered.

We consider a pair of instructions to be *solved* if either one or more distance vectors have been computed, or if the instructions are identified as independent. Finally, if there are no unsolved pairs of instructions remaining, the set of distance vectors suffices to validate a schedule. Given a transformation T, we compute the scalar product between T and the distance vectors $\vec{d_i}$:

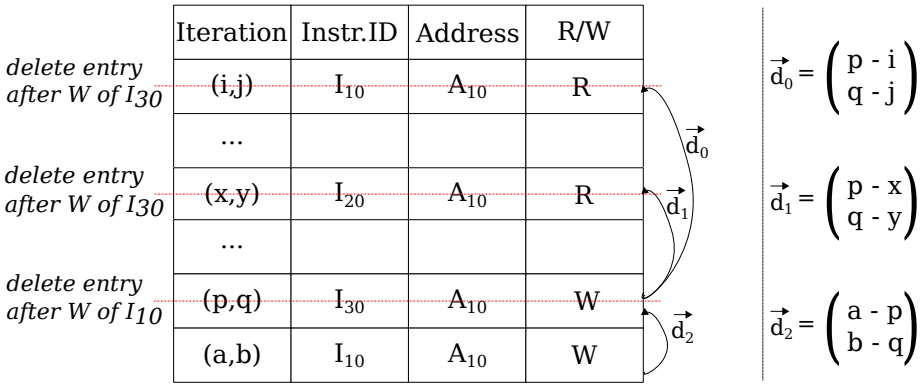


Figure 3.3: Table for computing the distance vectors dynamically (2-depth loop nest)

$$T \cdot \vec{d_i} = \vec{r_i}$$

The schedule is valid if the first non-null component is strictly positive, for all resulting vectors $\vec{r_i}$. The parallel loop level is the outermost level not carrying dependence.

3.3 Adapted polyhedral model as a transformation model: code patterns

The polyhedral model is both an analysis and a transformation model by itself. Since our purpose is to use it dynamically, the model must be adapted to avoid costly computations and to stay in the limits of a reasonable runtime overhead. Polyhedral transformations represent changing the execution order of the iterations of the loop, or of the statements in the body of the loop. Moreover, the loop structure can be altered. Among traditional transformations one could mention loop skewing, interchange, fusion or loop fission. Polyhedral transformations are mostly affine. The underlying reason is that they preserve the collinearity and the convexity of points, thus maintaining the representation of the loop in the polyhedral form. The validity of a transformation is evaluated with respect to the polyhedral dependences, as presented in Section 2.1.5.

State-of-the-art compilers, such as Pluto [15] or Polly [55], already bring the benefits of the polyhedral model to transform loops, generating a parallel schedule optimized to improve data locality. Nevertheless, since the time required to select such a transformation and to generate the transformed code can be very high, one must adapt and improve the classic method, for a dynamic use. The remaining of this section presents our technique for a fast and efficient dynamic code generation using *code patterns* and compares it to other approaches.

3.3.1 Parallelizing transformations with code patterns

Loop nests whose control or data dependences cannot be fully determined at compile-time, cannot be readily modeled in the polyhedral representation. One solution would

be to speculatively transform the code, but surrounded by guards for resolving dependences at runtime, and for verifying the validity or relevance of the transformed, parallel code. Such approach is proposed by Patel and Rauchwerger in [93]. However, this solution constrains the way the code is parallelized, by setting the instruction schedule in advance. Moreover, although the guards invalidate the pre-defined parallel code, other parallel schedules could be suitable. Another important aspect is that a parallel solution may yield low performance, for instance because of the poor data locality induced by the memory accesses which cannot be predicted at compile-time. To overcome these drawbacks we propose a more flexible solution based on code patterns.

Code patterns have already a long history of addressing runtime code specialization and simple optimizations. Noël *et al.* [88] use templates with "holes" declared as external global variables that are filled at runtime. They perform optimizations based on constant propagation, strength reduction and loop unrolling. This proposal sets the premises for our work of performing more advanced loop optimizations dynamically. More recent works [65] extend this approach and use value prediction based on affine functions to build their templates. They are specialized by statically precomputing values of the affine functions at some specific points. Runtime instantiation represents selecting one of the precomputed values, thus the overhead is negligible. This proposal invites to more advanced loop optimizations, however, in this form, template instantiation relies on statically known values. We extend this by performing re-scheduling of loop iterations, on codes for which the parameters cannot be pre-computed statically. In another proposal [111], patterns are seen as pre-compiled fragments of code stitched together dynamically.

We propose a mixed static-dynamic approach, in the form of code patterns. Namely, at compile time, several parallel code patterns are prepared, from which different code versions can be dynamically generated. Since the underlying concept of our work is the polyhedral model, the parallel code patterns are built to support the generation of distinct code versions, by applying various polyhedral transformations. They can be seen as "polyhedral masks", tailored such that the result of a large set of parallelizing transformations can be implemented at execution-time, by patching some pre-defined areas of code. This simple and fast code generation makes it possible to generate different versions of the code (each version obtained by applying a new transformation), from the same pattern, during one single execution of the loop. This is a key concept in our proposal, adding support for *partial parallelism* in loops and for *adapting* to the current execution behavior. More details are given in Chapter 4, Section 4.3, which explains how the newly generated versions are alternated to complete the loop execution.

To better argument the use of code patterns, we start with a pedagogical example in which the code can be statically analyzed, and we detail in what follows the implications of non-statically analyzable code and our solutions to handle it.

First, consider the simple two-loop nest, with indices i, j in Listing 3.7.

Performing the affine transformation (x, y) = (i, i+j) on the original loops, one obtains a new version. We can then rewrite the code to loop on x and y instead of i and j, obtaining the skewed routine from Listing 3.8, equivalent to the one from Listing 3.9.

Listing 3.7: Original Code

```
do i = 1,6
    do j = 1,5
        A(i,j) = A(i-1,j+1)+1
    enddo
enddo
```

Listing 3.8: Transformed Code T1

```
do x = 1,6
    do y = 1+x, 5+x
        A(x,y-x) = A(x-1,y-x+1)+1
    enddo
enddo
```

Listing 3.9: Equivalent transformed Code T1

```
do x = 1,6
    do y = 1+x, 5+x
        i = x
        j = y-x
        A(i,j) = A(i-1,j+1)+1
    enddo
enddo
```

Listing 3.10: Transformed Code T2

```
do x = 2,11
    do y = max(x-5,1), min(6,x-1)
        A(y,x-y) = A(y-1,x-y+1)+1
    enddo
enddo
```

Listing 3.11: Equivalent transformed Code T2

```
do x = 2,11
    do y = max(x-5,1), min(6,x-1)
        i = y
        j = x-y
        A(i,j) = A(i-1,j+1)+1
    enddo
enddo
```

A more complex transformation is (x, y) = (i+j, i), which leads to the skewed and interchanged loops in Listing 3.10. equivalent to:

As one can notice, the loop structure remained the same (except the loop bounds and the initialization code), despite the different affine transformations that have been applied. By rewriting the loop bounds and the memory accesses as generic affine functions of the enclosing loop indices, we can build a simplified pattern from which an infinite number of versions can be generated, provided that the loop structure and the order of the statements remains unchanged.

Listing 3.12: Simplified pattern

```
do x = low_x, upp_x
    low_y = max(a*x+b, cst)
    upp_y = min(c*x+d, cst)
    do y = low_y, upp_y
        i = e*x+f*y+g
        j = h*x+k*y+l
        A(i,j) = A(i-1,j+1)+1
    enddo
enddo
```

At runtime, the coefficients of the linear functions computing the loop bounds and the original iterators are assigned values according to the affine transformation to be applied.

As presented in Section 3.2, we insert virtual iterators in all loops, which makes it possible to represent *while-* and *do-while* loops in the polyhedral abstraction. As an example, consider the loop nest:

Listing 3.13: Original code: 2-depth while loop nest

```
while (p != NULL)
    q = q0
    while(q != NULL)
        p->val += q->val
        q = q->next
    endwhile
    p=p->next
endwhile
```

mapped onto the pattern, becomes

Listing 3.14: Transformed code: 2-depth for loop nest

```
do i = 0 , upp_i
    do j = 0, upp_j
      /**initialization code**/
      p = a*i+b*j+c
      /**guarding code**/
      if (p != NULL)
          q = q0
        /**initialization code**/
          q = d*i+e*j+f
          /**guarding code**/
          if(q != NULL)
              p->val += q->val
              q = q->next
          endif
          p=p->next
      endif
    enddo
enddo
```

In order to preserve the correct semantics of the original code, the patterns must include:

1. *Guarding code*: to control when the body of each loop is executed;
2. *Initialization code*: to initialize particular scalars called *basic scalars* in the beginning of each iteration;
3. *Verification code*: which verifies the speculations.

The next subsections provide in-depth details on each of these aspects.

1. Guarding code

As shown previously, we insert virtual iterators i, j and transform the *while* loops into *for* loops. The computation of the new loop bounds is done automatically at runtime,

for any loop nest depth, using the Fourier-Motzkin elimination algorithm [109]. An implementation of the algorithm is available in the software library FM [97]. Note that the conditions of the original while loops are preserved in the code, ensuring that we do not execute unpredicted iterations. Similarly, we check that all iterations have been executed, by verifying that the exiting iteration, with respect to the sequential order, executes when predicted (see Section 3.4). We call this *guarding code* and it is aimed to verify our speculation on the loop bounds. The *guarding code* is inserted in the innermost loop, thus allowing various affine transformations, such as loop interchange.

To argument our choice of design, we show a pattern where the guarding code, namely the conditions of the original loops, is preserved in the original position, and we emphasize the consequences when one tries to apply a polyhedral transformation. Consider the simple 2-depth loop nest in Listing 3.15.

Listing 3.15: Original code of a 2-depth loop nest

```
while (cond1)
   while(cond2)
     ....
   endwhile
endwhile
```

For the purpose of example, in the following pattern we proceed by:

1. inserting the i and j virtual iterators,
2. transforming the *while* loops into *for* loops,
3. preserving the original conditions *cond*1 and *cond*2 associated to the corresponding loops (namely *cond*1 of the original outermost loop, corresponds to the outermost transformed loop; likewise for *cond*2).

One obtains:

Listing 3.16: Sequential code with virtual iterators

```
for i = 0 , N
   if (cond1)

      for j = 0, M
         if (cond2)
            ....
         endif
      endfor

   endif
endfor
```

Given that the values of N and M are large enough to allow the execution of all original iterations, the two loops are equivalent. However, should one perform a loop interchange, the generated code would be:

Listing 3.17: Loop interchange with imperfectly nested guarding code

```
for x = FM_low_x, FM_upp_x
    if (cond1)

        for y = FM_low_y, FM_upp_y

            i = y
            j = x

            if (cond2)
                ....
            endif
        endfor

    endif
endfor
```

which is invalid, because $cond1$, originally corresponding to the outermost loop i, should now be associated to the inner loop y, since due to the loop interchange, $i = y$.

Thus, to ensure that all parts of code are reachable and are evaluated at the correct point of execution, we include the guarding code inside the innermost transformed loop:

Listing 3.18: Loop interchange with perfectly nested guarding code

```
for x = FM_low_x, FM_upp_x

        for y = FM_low_y, FM_upp_y

            i = y
            j = x

            if (cond1)
                if (cond2)
                    ....
                endif
             endif
        endfor
endfor
```

which makes both $cond1$ and $cond2$ reachable and correctly evaluated.

Note that this is only a simplified version of the pattern, of a perfectly nested original loop. Chapter 7, Section 3.3.1 provides examples of codes and their associated patterns for imperfectly nested original loops.

2. Initialization code

We use the linear functions obtained from the profiling phase, to initialize some particular scalars at runtime. This value prediction mechanism is similar to the ones presented in [37, 75, 100, 117].

Note that we initialize only the scalars whose values depend on a previous iteration, i.e., which are read before being written in the current iteration. In the SSA form [32], these correspond to the ϕ nodes [1]. We promote the registers to memory and initialize each scalar originating from a ϕ node, just before its first use in the loop. This practice ensures that the correct starting value is assigned in the beginning of each iteration. We call them *basic scalars*, because the values of all other variables can be deduced from them. Similarly, the computation of the memory addresses to be accessed is derived from the *basic scalars*. The initialization code is equivalent to *privatization*, since all values that depend on other iterations are re-declared locally in each thread and initialized with the interpolating linear functions.

Thus the new shape of the loop nest is in compliance with the polyhedral model. In this form, the loops can be further transformed as in the case of statically analyzable code, by applying an affine and unimodular transformation T:

$$T \cdot \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} \Leftrightarrow \begin{pmatrix} i \\ j \end{pmatrix} = T^{-1} \cdot \begin{pmatrix} x \\ y \end{pmatrix}$$

Thus we obtain a new loop version in x and y, as in Figure 3.4. The bounds of the loops and the coefficients of the linear functions (shown in red) are assigned values dynamically.

The coefficients of the linear functions are computed by applying the transformation matrix T on the predicting linear functions obtained from the profiling phase. In gray it is displayed the initialization and verification code that is inserted in the pattern, in addition to the original code.

3. Verification code

The verification code is required to validate or invalidate the premises of the speculations. Since both the memory state and the control flow has been modified, different types of instances must be verified:

- Basic scalars
- Memory accesses
- Loop bounds

More details on the verification code are presented in Section 3.4.

[1] *phi* nodes track the different paths from which a live variable can come.

```
forall (x = 0 ; x <= cx*chunk_size+constx ; x++){
  for (y = 0 ; y <= cy1*x+cy2*chunk_size+consty ; y++){
       //initialization code
       i=trans_inv_i(x,y); j=trans_inv_j(x,y);
       p = s1*i+s2*j+s3;
       //guarding code
       if (p!=NULL){
           q = q0
          //initialization code
          q = s4*i+s5*j+s6;
          //guarding code
          if (q!=NULL){
           //addresses predictions verifications
           ...
           if (&q!=a2*x+b2*y+c2) rollback();
           S2:store &q;
           ...
          endif
          if (&p!=a1*x+b1*y+c1) rollback();
          S1:store &p;
           //scalars values predictions verifications
           i=trans_inv_i_next(x,y); j=trans_inv_j_next(x,y);
           if (p!=s1*i+s2*j+s3) rollback();
           if (q!=s4*i+s5*j+s6) rollback();
          }
       }
  }
  }
```

Figure 3.4: Simplified code pattern

```
outermost_loop
{
  code_1
  inner_loop_1
  code_2
  inner_loop_2
  ...
  code_n
}
```

Figure 3.5: General structure of a loop nest

3.3.2 Complex code patterns

We have adapted the polyhedral model for the general purpose code, and we focus for the moment on its dynamic usage. Namely, we target loops containing pointers, indirect references and control structures that cannot be statically disambiguated. Nevertheless, further extensions can be imagined. Our model is designed to handle imperfect loop nests, tiling or more aggressive transformations.

Imperfect loop nests

Any imperfect loop nest can be seen as having the general structure shown in Figure 3.5: inside the outermost loop, all inner loops are interleaved with some code executed at the outermost level.

Using the polyhedral model, such a loop nest can be represented as shown in Figure 3.6 for two nested loops, where inner loop indices are considered as having consecutive values, *i.e.*, the upper bound of the inner loop i is the lower bound of the inner loop $i + 1$. Each iteration is represented as an integer point whose coordinates are given by the values taken by the loop indices. The sequential order of computation consists in executing each iteration from bottom to top, and vertical line by vertical line, from left to right. Since inner loops are spaced by code snippets, such snippets can be associated either to the first iteration of the next inner loop, or the last iteration of the previous loop. We selected the associations represented in the figure. They provide us a way to handle imperfect loop nests even when applying the same parallel schedule to all their instructions, although the general theory of the polyhedral model supports different schedules per instructions when applying it at compile-time.

Parallelizing such a loop nest using the polyhedral model, from a geometric standpoint, is translated into a linear change of basis where new loop indices are used to scan the iterations following a new schedule. For example, the dashed lines represented in figure 3.6 symbolize such a transformation. The transformed outermost loop parses all lines, while the inner loops, plus the intermediate codes, scan the points from each line. Either the outermost loop or the inner loops can be parallelized. However, the latter solution might be less efficient since a synchronization barrier has to be set between the iterations of the outermost loop. But it could still be an interesting parallel version if it does not introduce any rollbacks and if the inner loop bodies embed a significant

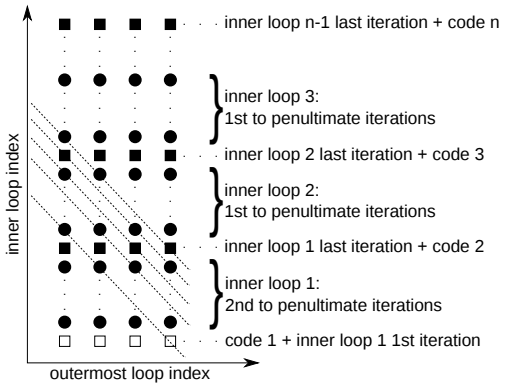


Figure 3.6: Graphic representation of the iterations of a loop nest (courtesy: Ph. Clauss)

```
forall (x=lb_x ; x <= ub_x ; x++)
{
  y = lb_1_y(x) ;
  if (j(x,y) == lb_1_j(x,y)) code_1 ;
  for (y=lb_1_y(x) ; y <= up_1_y(x) ; y++) { ... }
  if (j(x,y) == ub_1_j(x,y)) code_2 ;
  for (y=lb_2_y(x) ; y <= up_2_y(x) ; y++) { ... }
  ...
  if (j(x,y) == ub_n-1_j(x,y)) code_n ;
}
```

Figure 3.7: General structure of a parallel loop nest

amount of computations. When considering two successive loop levels, such a linear change of basis implies a linear transformation of the original loop indices, i and j, into new loop indices, x and y. Thus the original loop indices can be expressed depending on x and y by linear functions of the form: $i(x, y) = \alpha x + \beta y$ and $j(x, y) = \alpha' x + \beta' y$. This can be obviously generalized to any loop nest depth.

In Figure 3.6, it can be observed that different kinds of iterations have to be handled by the same loop body: iterations which are originally part of the inner loop, and iterations which are the merge of a code snippet and a boundary inner loop iteration. Hence the parallel code has to switch between different types of iterations during its execution. This can be accomplished using dedicated guards inside the loop bodies. Those guards consist in simple conditions on the original inner loop index of the form: $j(x, y) == lb_j^k(x, y)$ or $j(x, y) == ub_j^k(x, y)$, where lb_j^k, respectively ub_j^k, denotes the lower, respectively upper bound of the k^{th} original sequential inner loop, which can also be expressed as linear functions of x and y. Notice that without loss of generality, the upper and lower bounds must be consecutive, *i.e.*, $ub_j^k + 1 = lb_j^{k+1}$. The general scheme of a parallel loop nest is shown in Figure 3.7, where `forall` denotes the parallel loop, and `lb_x`, `ub_x`, `lb_1_y(x)`, `up_1_y(x)`, `lb_2_y(x)`, `up_2_y(x)` denote the new loop bounds related to x and y after the polyhedral transformation.

Listing 3.19: Affine loop nest of depth 3

```
for (i = l_i; i < u_i; i++)
    for (j = 0; j < ai + b; j++)
        for (k = 0; k < αi + βj + γ; k++)
            ...
```

Tiling

Since tiling a loop alters the structure of the loops inside the nest, a specific pattern must be designed to support such a transformation: an affine loop nest of depth 3 of the form shown in Listing 3.19 becomes the one in Listing 3.20, of depth 6:

Listing 3.20: Tiled loop nest of depth 3

```
for (i_T = 0; i_T < u_i; i_T = i_T + TS_i)

  for (j_T = 0; j_T ≤ upper_j(i_T); j_T = j_T + TS_j)

    for (k_T = 0; k_T ≤ upper_k(i_T, j_T); k_T = k_T + TS_k)

      for (i = i_T; i < min(i_T + TS_i, u_i); i++)

        for (j = j_T; j < min( ai + b, j_T + TS_j ); j++)

          for (k = k_T; k < min( αi + βj + γ, k_T + TS_k ); k++)
          ...
  where:
  if (a > 0) upper_j(i_T) = a * (i_T + TS_i - 1) + b
  else       upper_j(i_T) = a * i_T + b

  if (α > 0)
      if (β > 0) upper_k(i_T, j_T) = α * (i_T + TS_i - 1) + β * (j_T + TS_j - 1) + γ
      if (β < 0) upper_k(i_T, j_T) = α * (i_T + TS_i - 1) + β * j_T + γ
  if (α < 0)
      if (β > 0) upper_k(i_T, j_T) = α * i_T + β * (j_T + TS_j - 1) + γ
      if (β < 0) upper_k(i_T, j_T) = α * i_T + β * j_T + γ
```

To transform the tiled version into a parallel polyhedral transformed version, one must apply the new schedule on i_T, j_T, k_T, while the original iterators i, j and k remain unchanged. This ensures that a transformation such as skewing, for instance, skews the parsing of the tiles, and not the content of tiles itself, to preserve data locality.

The steps to build the parallel tiled version are:

1. Generate the dependence vectors: first, one computes the dependence vectors on the original iterators $i_1, i_2, .., i_n$ and obtains a set of vectors of the form

Listing 3.21: Tiled and transformed loop nest of depth 3

for $(x_T = FM_{low_x};\ x_T < FM_{upp_x};\ x_T = x_T + TS_x)$

for $(y_T = FM_{low_y};\ y_T < FM_{upp_y};\ y_T = y_T + TS_y)$

for $(z_T = FM_{low_z};\ z_T < FM_{upp_z};\ z_T = z_T + TS_z)$

$$\begin{pmatrix} i_T \\ j_T \\ k_T \end{pmatrix} = T^{-1} \begin{pmatrix} x_T \\ y_T \\ z_T \end{pmatrix}$$

for $(i = i_T;\ i < min(i_T + TS_i, u_i);\ i++)$

for $(j = j_T;\ j < min(\ ai + b,\ j_T + TS_j\ );\ j++)$

for $(k = k_T;\ k < min(\ \alpha i + \beta j + \gamma,\ k_T + TS_k\ );\ k++)$

...

where:

FM_{low_x}, FM_{upp_x}, FM_{low_y}, FM_{upp_y}, FM_{low_z}, FM_{upp_z} are the transformed bounds computed with FMlib.

$\vec{d_N} = \begin{pmatrix} d_1 \\ d_2 \\ ... \\ d_n \end{pmatrix}$, where n is the depth of the original loop nest. By decomposing the vector $\vec{d_N}$ in the canonical form, one obtains the set of dependence vectors for the transformed iterators $i_{1T}, i_{2T}, .., i_{nT}$:

$$\vec{d_N} = \begin{pmatrix} d_1 \\ d_2 \\ ... \\ d_n \end{pmatrix} \rightarrow \left\{ \begin{pmatrix} d_1/|d_1| \\ 0 \\ ... \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ d_2/|d_2| \\ ... \\ 0 \end{pmatrix}, ..., \begin{pmatrix} 0 \\ 0 \\ ... \\ d_n/|d_n| \end{pmatrix}, \forall d_i \neq 0 \right\}.$$

2. Compute the scalar product between each dependence vector $\vec{d_N}$ and the transformation matrix T and verify that the first non-null component of each resulting vector $\vec{r_N}$ is strictly positive.

3. Generate the transformed pattern, where the outermost parallel loop is parallelized.

For the loop nest from Listing 3.19, the pattern of the tiled transformed version is given in Listing 3.21.

The initialization and verification code and the guarding conditions are inserted in the innermost transformed loop, as in the case of un-tiled code patterns.

3.3.3 Dynamic code generation using code patterns

The genericness of the code patterns is given by the linear functions which can be adapted at runtime to accommodate different polyhedral transformations. Hence, the coefficients of the linear functions are the only parts of the code that vary during execution, enabling the adaptive parallelization. They are assigned values dynamically, based on the results of the instrumentation and of the dependence analysis. In consequence, generating the parallel code versions at runtime, from the parallel code patterns is very fast.

3.3.4 Genericness of code patterns

To argument our choice for generating the transformed code using code patterns, we compared them with two other approaches, varying from purely static towards entirely dynamic strategies:

1. Statically prepared versions;
2. Using Just-in-time (JIT) compilation.

Further, we discuss the advantages and drawbacks of each of these methods.

Statically prepared versions

The first approach considers performing a static analysis of the code and proposing several parallel schedules. The parallel code versions are statically generated and embedded in the binary file. In case the analysis cannot provide sufficient information, it can be aided by an offline profiling phase, to characterize the code's behavior. The main advantage is that it is very fast, since it incurs no runtime overhead for generating the parallel code. On the other hand, one trades flexibility, as the generated versions are fixed and cannot be adapted dynamically, should the code behavior change during execution or depend on the input data. Additionally, this approach leads to a significant code size increase, due to the numerous versions that must be prepared for providing parallelization opportunities.

JIT compilation

In contrast, a purely dynamic approach based on JIT brings the benefits of adaptiveness, as the code is generated at runtime, once all required information is fully available. The great advantage is that any transformation can be applied, thus opening the possibility for aggressive code optimizations. Moreover, this strategy is suitable for parallelizing codes whose behaviors are highly dependent on the input data, or vary during the execution. Nevertheless, despite these advantages, the JIT approach is expensive and not yet adapted for a frequent runtime usage, since the compilation overhead is too high.

3.3.5 Static versions *vs* Patterns

Statically propose several parallel schedules: The approach of generating at compile time multiple parallel versions presents some advantages that are not negligible in practice. For this reason, the next paragraph details this approach, yet it explains our reasons for considering the patterns as the most suitable strategy.

The main goal of speculative parallelization is to execute codes with multiple threads, with a minimal number of rollbacks. Yet, the driving force behind parallelization is to improve performance. Our strategy for achieving this target and going beyond the scope of existing speculative systems is to apply polyhedral optimizations before parallelizing the code. The main obstacle is represented by the overhead of generating the transformed code.

In this respect, preparing at compile time a number of parallel code versions, highly optimized thanks to advanced polyhedral transformations, represents a viable solution. More precisely, there are no constraints on the transformations to be applied, which opens the doors for generating highly efficient parallel code. Existing tools, such as Pluto [15] and Cloog [11], explore optimizations which allow instruction-wise scheduling and aggressive loop transformations, without preserving the original loop nest hierarchy and structure. Transformations such as loop unrolling, fission or fusion and software pipelining show impressive benefits, yet they are prohibitively time consuming to be applied dynamically.

On the other hand, static analysis reaches its limits soon, in case of codes that cannot be fully disambiguated statically, handling pointers and control structures that depend on values computed dynamically.

We considered these as strong limitations, and adopted the strategy of parallel code patterns, which allow more flexibility and support for adapting to the dynamic behavior of the code.

Searching for solutions to the drawbacks presented by the purely static approach, we chose the code patterns to address the following aspects:

1. Flexibility: when one cannot predict the behaviour of the code statically, it is difficult to generate good parallel schedules. The first strategy is to perform a *partial static dependence analysis* and to propose a set of polyhedral transformations, valid with respect to the information statically available. Several transformed code versions can be prepared and then evaluated at runtime. However, it is often the case that little information can be statically discovered and these versions would be soon invalidated by dependences occurring during execution. This is due to the fact that predicting the behavior of pointers and dynamic memory allocation at compile time is frequently close to impossible.

This method could benefit significantly from an offline profiler, aimed to exhibit the information statically unavailable. Whether profiling a sample or a full execution of the code leads to further argumentation. For our purposes, it suffices to consider an offline profiler powerful enough to provide the information required for a detailed dependence analysis. Similarly, several optimized and parallelized code versions can be statically generated, with higher chances for preserving their validity during execution. Still, these versions are subject to become invalid when the code behavior changes phases during the execution, or if it depends on the input data. More importantly,

Listing 3.22: Non-statically analyzable loop

```
while(!exit_cond){
    p = malloc();
    if (cond) q = malloc();
}
```

should none of the versions prepared at compile time be valid, one cannot continue the process of parallelization, as no other parallel schedules are provided.

In contrast, using parallel code patterns, one can decide at runtime the final shape of the parallel code, guided by the information that becomes available dynamically. Moreover, the same code pattern can generate different code versions, for a loop that exhibits different phases during the execution. Even the same parallel schedule can be applied, but the linear functions might differ. Hence, from the same pattern, one can build distinct versions to complete one execution of the same loop. Consider for example the following loop in Listing 3.22, in which *cond* cannot be predicted statically and it might depend on the input data. However, the pattern can handle the change of phases, as the parallel code is generated only after *cond* has been evaluated. If *cond* is false during a significant number of iterations it is worthwhile parallelizing the code, considering the memory allocation for p. As soon as *cond* becomes true, the pattern is adapted, such that it takes into consideration the interleaved allocations in memory of p and q, hence the coefficients of the linear functions are changed. This cannot be achieved by means of statically prepared versions, since one cannot foresee the iteration when the condition becomes true.

2. Adaptability: equally important, the patterns bring the advantage of adapting at runtime to some behaviors that could not have been statically predicted, i.e. for which no code version or transformation has been statically prepared. Under these circumstances, a new parallel schedule can be generated dynamically, potentially taking into consideration heuristics, such as, characterizing the program's behavior and selecting a suitable schedule accordingly; or using deterministic approaches, based on reuse distance, as done in Pluto [15], and considering a history of transformations that proved to be valid on previous runs. Currently, we have not explored this approach in depth, but the patterns are designed for further extensions.

3. Small code size: last but not least, the patterns ensure a reduced size of the binary code, since each pattern embeds a class of possible transformations, rather than including all transformed versions in the binary file.

Compared to statically generated versions, the drawback of the code patterns is that code is not specialized. The computation of the loop bounds and of the arithmetic expressions involved in the computation of memory addresses are not simplified. Also, patterns cannot yet support transformations that alter the structure of the loop, such as unrolling, or vectorization.

To conclude, we consider the code patterns more adapted for a dynamic use, as, by design, one can control the trade-off between performance and generality. Nevertheless, this imposes some limits such as a fixed structure of the loop nest and no reordering

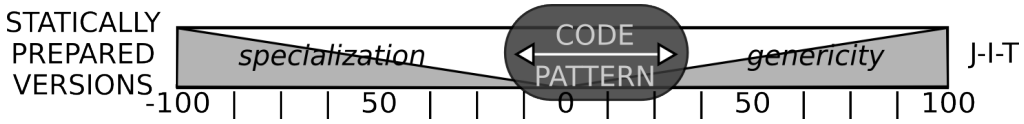


Figure 3.8: Code patterns to balance genericity and performance (courtesy: Ph. Clauss)

of instructions inside the loop body. One can argue in favor of more general code patterns, that would support more aggressive code transformations. This would bring the patterns closer to the JIT compilation strategy. However, the genericness of the code patterns comes with the price of a higher overhead: both for generating parallel code and for executing it. A highly generic approach requires more patching and testing. Thus, the generated code is not as efficient, due to the numerous conditions and checking code that needs to be evaluated. We consider that it is impractical to design a unique code pattern able to support all possible transformations. In consequence, we propose several patterns, each of them tailored for a class of transformations. In this manner, we can overcome the limits of the statically prepared versions that are too restrictive and do not allow any flexibility or adaptability at runtime. And we do not face the problem of the high runtime overhead for generating parallel code, as in the JIT compilation. Our measurements for runtime code generation are illustrated in Chapter 6, Section 6.2. According to the goals, one adjusts the balance between performance and genericness, as sketched in Figure 3.8.

3.3.6 Patterns *vs* JIT

The code patterns represent a trade-off between the statically prepared versions and the JIT compilation. The advantage of using code patterns is that the code generation incurs a considerably smaller overhead than in the JIT compilation approach, as it consist of simply assigning values to the coefficients of the linear functions, dynamically.

As seen in Section 3.3.4, JIT compilation may be expensive for performing all steps required for dynamic optimizations at runtime: schedule selection, code transformation and binary code generation. Yet, we plan to conduct experiments aimed to evaluate the performance of a mixed strategy of patterns optimized using JIT. The strategy is to build the patterns statically, as presented before, however, without generating the corresponding binary file, but keeping them in the intermediate representation of the compiler. Next, the patterns are patched dynamically, optimized and JIT-ed. Since the patterns abound in expressions and computations of linear functions that can be performed just once, given that the coefficients are known, the JIT can specialize the code through classic optimizing transformations such as constant propagation, strength reduction, common sub-expression elimination, etc. The aim of the experiment would be to determine what are the benefits of applying optimizations on patterns using JIT, compared to executing the un-optimized patterns without paying the cost of runtime code generation. The benefits of integrating the JIT in our framework is subject to further investigations and we consider it for future extensions.

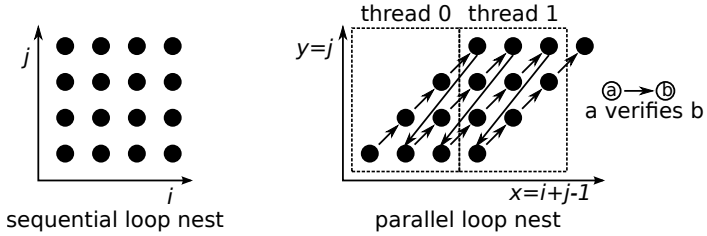


Figure 3.9: Illustration of basic scalar values verification in the parallel loop nest (courtesy: Ph. Clauss)

3.4 Verification and rollback systems in the polyhedral model

Since all code transformations, including parallelization, rely on speculations, one must periodically check the predictions in order to validate / invalidate the code. Upon a misspeculation, a rollback is initiated, restoring the correct state of the memory. Not only we maintain the correct memory state, but we also transform the control flow of the loop, using new loop iterators and bounds.

3.4.1 Verification

The model we propose is based in the linear description of the memory accessing behavior. Hence, our speculations consist of the linear functions that predict the memory addresses being accessed. Validating a transformation is equivalent to verifying the correctness of the interpolating linear functions. Under these circumstances, it suffices to compare the actual addresses being accessed, by the original instructions, to our predictions, given by the linear functions. Recall that the code inside the body of the loops in the code patterns is a copy of the original code. *Thus, the memory accesses are performed by a copy of the original memory instructions, whose target addresses are computed directly or indirectly from the basic scalars, which are initialized at each iteration.*

We divide the type of verification in three categories, depending on the instances being verified:

1. Basic scalars. Since we privatize scalars whose values depend on previous iterations (ϕ nodes) by initializing their values in the preamble of each iteration, a verification is required. When the execution of the iteration completes, we verify that the value computed by the code and the value we predict coincide. For this verification, *we compare the actual value with the one expected for the next iteration in the sequential order.* This strategy is illustrated in Figure 3.9, where the initial iteration domain has been skewed.

Notice that iterations verified by other iterations that are executed by a different thread can be executed before being validated. Nevertheless, any dependence violation is necessarily detected at a given time, either before or after it occurs.

Validation of basic scalars ensures that all values computed in the loop reach the

Listing 3.23: Original loop nest

```
while (cond1)
    while(cond2)
      ....
    endwhile
endwhile
```

predicted values. Hence, the result of the dependence analysis is preserved as long as the interpolating linear functions remain valid.

2. Memory accesses. As some iterations might execute before being validated by the preceding iteration according to the sequential order, one is required to verify all memory accesses, in the current iteration. Namely, one must ensure that each instruction that accesses memory, targets a location that has been predicted. This has twofold consequences. First, it ensures that no invalid access is performed. And second, it guarantees that the state of the memory can be safely restored, as no modification outside the predicted memory area has been performed.

3. Loop bounds. The number of iterations of the subloops are interpolated and have a direct role in the polyhedral transformation being applied. Therefore, one must verify the prediction. The verification code relies heavily on the *guarding* code, presented previously. The transformed loop bounds control the number of iterations to be executed by each loop of the nest and together with the *guarding code*, it must be verified that:

(a) each loop executes *all* its iterations,

(b) but no loop executes more iterations than it should.

Informally, this is equivalent to saying that the loop does not exit too early or too late. Due to the out-of-order execution of the iterations, the code must allow the execution of the last iteration of a loop (according to the sequential order) without exiting, as it might be followed by other iterations.

As an example, consider the border iteration of the subloop in the code snippet from Listing 3.23.

By inserting the i and j iterators and transforming the while loop into for loops, one obtains the code from Listing 3.24.

The interpolating linear function predicted M iterations for the subloop. Therefore, the condition on the transformed loop allows only M iterations, whereas for the outermost loop no prediction is possible, consequently a default value N is used.

By applying a transformation, the iterators i and j are replaced by x and y, whose bounds are computed with the FM library, as explained in Section 3.3.1. Note that by profiling on a sample, one cannot estimate the number of iterations of the outermost loop, unless it is known statically. Hence, its last iteration cannot be run in parallel with

Listing 3.24: Sequential code with virtual iterators

```
for i = 0 , N
   for j = 0, M

      if (cond1)
         if (cond2)
            ....
         endif
      endif
   endfor
endfor
```

other iterations, since one can not verify that the loop actually executed all iterations. *To ensure that the loop finished its execution at the correct point, the last iteration must be executed solely after the parallel iterations completed their execution, or, if several iterations remain to be executed, they must be run in sequential order.*

Nevertheless, in the case of subloops, their bounds are interpolated, and, thanks to this prediction, the execution order of their iterations can be changed, while precisely controlling the loop exiting iterations. In Listing 3.25, the last iteration (according to the sequential order) of the subloop might be executed before the parallel version has to exit, due to the re-scheduling of iterations. Therefore, *cond*2 becomes true, which would trigger the exit of the subloop in the sequential order. Yet, the code $(j \neq \alpha \cdot x + \beta \cdot y + \gamma)$ verifies if the exit condition became true at the predicted iteration. If there is a misprediction, the execution rollbacks, else, it continues until all predicted iterations of the loop have been indeed executed.

In short, the *guarding* code *cond*1 triggers the exit of the loop as soon as it becomes false, whereas *cond*2 guards the code and induces a rollback only in case of a misprediction on the bounds of the subloop.

Once a misprediction is encountered, it is signalled to all threads and they stop their execution. Next, a rollback is initiated, aimed to restore the correct state of the memory and to allow the re-execution of the faulty iterations.

3.4.2 Rollback

Any speculative system relies on a mechanism to restore the execution upon a misspeculation. Several strategies have been proposed, such as executing the speculative threads in a separate memory area which is copied in case of validation or simply discarded if a misprediction is detected [119]. Other proposals employ the Copy-on-Write method, which limits the amount of memory that needs to be copied between the threads [104, 117, 118], or rollback mechanism relying on transactional memory [1, 85, 104].

Since faulty iterations can alter memory with incorrect values, an anticipated memory backup is performed before launching a chunk of code embedding a speculatively

Listing 3.25: Transformed loop with guarding code

```
for x = lb_x , ub_x
    for y = lb_y , ub_y

      if (!cond1)
            rollback();
       if (!cond2)
          if (j ≠ α·x + β·y + γ)
            rollback();
          else continue;
                  endif
               ....
       endif
      endif
    endfor
endfor
```

optimized code version. The part of memory that has to be saved is the one that is predicted to be touched during the chunk's execution. With the codes and transformations considered by our framework in the polytope model, the memory addresses that are referenced can be represented by linear functions. Thus, for each write instruction, one is able to compute the memory range susceptible to be accessed, by using the interpolating linear functions and the loop bounds. This procedure is known under the name *extreme value range analysis*. These address ranges are backed up, such that they can be restored by a reversed copy, if the iterations are cancelled.

3.4.3 Conclusions

Our proposal tries to fill in the gap between the scientific codes and the general purpose codes, by applying powerful optimizations at runtime, on codes which cannot be statically analyzable. Our techniques for monitoring samples of executions of the loops, bring support for representing in the polyhedral abstraction loops that contain pointers or dynamic control structures. Next, the loops can be further modeled and transformed in this representation, in the view of generating high performing, parallel code. Since the entire procedure is applied at runtime, one must consider the costs of applying such optimizations. For this purpose, we have designed a set of code patterns, from which different parallel code versions can be dynamically generated, even during one execution of the loop nest. Since several versions can be generated from the same pattern, one can choose dynamically the optimal version, depending on the current execution context and on the observed behavior of the loop.

Finally, since the transformations are applied speculatively, we presented the mechanisms dedicated to ensure the correctness of the execution, relying on the verification code and a rollback system, should a misprediction occur.

Chapter 5 gives in-depth details on our implementation of these concepts.

Chapter 4

Strategies for full exploitation of the available parallelism

The goal of this chapter is to outline another main contribution of our proposal, namely, dynamically adapting to the current behavior of the loop. Our technique exploits parallelism in *partially parallel* loops and *adapts* to the current behavior of the code by successively applying a suitable polyhedral transformation. The rest of the chapter discloses our strategies on discovering phases in loop execution and automatically adapting by generating a different, optimized parallel code version, customized for each new phase of the loop. The interplay of two key aspects are essential in achieving this:

- *Multiversioning*: a method to generate several semantically equivalent versions of a piece of code. Nevertheless, each version is different: they could be either designed for instrumenting the code, or obtained as a result of a different optimization. Having a set of versions of a loop nest, allows one to select dynamically the version that suits best the current execution behavior. More details are offered in Section 4.2.

- *Chunking*: in order to discover and to adapt to a changing behavior, one solution is to slice the execution of the outermost loop in chunks. In short, each chunk consists of a number of consecutive iterations. Ideally, each chunk of the loop should map on a new phase of the loop's behavior. In-depth information is provided in Section 4.3.

We start with a short motivation that arguments the need of applying multiversioning and chunking in order to fully exploit the parallelization opportunities present in a loop nest.

4.1 Program phases

Understanding the program's behaviour is a fundamental aspect in performing code optimizations, whether they target performance, energy efficiency, thread scheduling optimizations, feed-back directed optimizations, simulation of architectures, predictions, and many other objectives. However, during one execution, the behavior may

change many times, therefore optimizing for an average behavior would not yield the optimal results. To overcome this problem, many studies [67, 110, 113, 135] were conducted on detecting *phases* that occur in the behavior of the code, during one execution, such that optimizing for each particular phase would result in a more precise optimization and incontestably better results. A *phase* is characterized by an interval of time during which the program exhibits a similar behavior.

To evaluate the similarity of behaviors, one requires to define a metric, which is a function of various characteristics of a program. For instance, in computer architecture, several works focus on characterizing parts of execution as being memory bound, or stalling due to branch mispredictions, having a high or a low number of cache misses, etc. [110]. Similarly, works focusing on estimating or reducing the power consumption [113] identify phases of programs that have a stable energy consumption.

To perform optimizations or predictions on the behavior of the code, one monitors the code to detect phases and whether they repeat after some intervals of time. Also, the change of phase is an important point during the execution, as it should trigger a new optimization strategy.

In the context of our work, we characterize phases with respect to the memory accessing behavior of the loop. We define the duration of a phase in terms of number of iterations of the outermost loop in the nest. Using the interpolating linear functions presented in Chapter 3, Section 3.2, we detect whether the behavior of the loop is stable during the sample of execution we instrument. Namely, if for all values that we monitor, we can build interpolating linear functions that remain unchanged during the instrumented iterations of the loop. Under these circumstances, we consider that a phase is detected, characterized by the set of interpolating linear functions we computed. This enables us to model the loop in the polyhedral abstraction, with a different representation for each phase.

Consider the following code extract from the *Floyd-Warshall problem*:

Listing 4.1: Loop with linear phases

```
for (k = 0; k < n; k++)
   {
     for(i = 0; i < n; i++)
        for (j = 0; j <n; j++)
          path[i][j] = path[i][j] < path[i][k] + path[k][j] ?
          path[i][j] : path[i][k] + path[k][j];
   }
```

The execution of this loop (on a given input) shows that the $if-branch$ is taken in an unpredictable behavior during the first iterations, and it is always executed afterwards. Therefore, during a very large number of consecutive iterations, the loop has a stable behavior, as it executes the same set of instructions and accesses memory using the pattern described by the same linear functions. Given the large number of consecutive iterations during which the behavior of the loop is stable and predictable, we can optimize and parallelize these chunks, without being hindered by phases interruptions that might occur. One can already identify the two phases exhibited by the loop: one, when the $if-branch$ is executed, and another when it is not. Consequently, to adapt

to the current behavior, one requires one code version for each phase. The next section presents *multiversioning* and the steps required for generating several versions of a piece of code.

4.2 Multiversioning

With the increasing complexity of the available hardware, including multicores and co-processors, multi-versioning has become a classical technique for using efficiently all available resources [26, 50, 51, 59, 80, 130]. It consists in compiling and embedding in the binary different versions of a hot region of code. It is particularly adequate for periodic instrumentation, adaptive version selection, and dynamic optimization in general. Our work presents a mechanism for switching dynamically from a version of a code to another, without interrupting the execution, and it provides support for advances in the following fields:

- instrumentation and sampling: instrumenting code usually induces a huge overhead (typically 10x at least, and up to 1000x for complex instrumentations [52]). Sampling is a classical technique to reduce the overhead, that consists in running the instrumented instructions only during some parts of the execution time. An efficient technique to implement this strategy [4, 38] is to generate two (or more) versions of a code at compile time, one instrumented and the other one non-instrumented (or some partially instrumented). The runtime system can then periodically switch from a version to another to enable/disable instrumentation.

- adaptive version selection: in some cases, the performance ratio between several versions of a code heavily depends on dynamic factors, such as the input data size, the processor load and number of available cores, or even the architecture itself when distributing a multi-platform executable. Thus, the best performing version to be executed at a given time on a given computer cannot be determined statically by the compiler, but should rely on a dynamic selection process. The runtime system may switch from a version of code to another, depending on the dynamic factors [26, 102].

- dynamic optimization: one can either launch a precompiled optimized version of code or JIT some parts of the code dynamically, to optimize them according to the current context.

Multi-versioning is an efficient technique for implementing such dynamic processes. However, in order to be transparent to the user, it requires the compiler to generate different versions of a piece of code automatically, and to customize them to be handled by a runtime system. It relies on two main features: tracking the interesting regions of code from the source to assembly; and cloning those regions in the Intermediate Representation (IR) of the compiler or in assembly.

Code tracking is a classical problem in debuggers, and it is well known that compiler optimizations make source-level tracking and debugging a difficult problem [17, 124]. As we are interested in performance, this is an important issue: we want to run full

optimizations, typically -O3, and still be able to track some regions of the source code till the assembly generation. Some work has been done on dynamic optimization [61, 69, 71] with a limited scope of optimizations, and mostly in virtual machines. We address a similar problem, but in the case of a classical compilation chain, where an executable file is generated from source code by the compiler.

Whether the multiple versions are prepared statically or generated dynamically, the main steps in generating multiple versions are the following:

- Identify the code regions marked for processing
- Clone the regions
- Customize each clone to create different versions
- Build a mechanism to enable switching between versions

In what follows, we review the main research works on each of the above steps.

4.2.1 Related work

Code tracking. Tracking code has always been a necessary technique, evolving from the simple strategies employed in the early debuggers, to complex approaches meant to correlate the original source code with dynamically optimized code.

Solutions have been proposed [17, 124] addressing the well-known *code location problem*, locating an original statement in the optimized code, and the *data-value problem*, retrieving the value of a variable which is not available due to code modifications. Tracking the suite of code transformations performed in the optimization phase has been identified early as an impractical solution, since compilers reorder, replicate, delete, merge, transform the code, eliminate variables, or synthesize new ones. A viable alternative is presented by Brooks *et al.* [17] as a method for acquiring extended debugging information, communicated from one optimization phase to another. It is thus demonstrated that by annotating the symbol table and creating maps between the source code and the optimized code, it is possible to highlight optimizations as loop interchange, loop invariant optimizations or to track the value of a variable, regardless if it resides in memory or in a register. Van Deursen *et al.* [124] give an efficient implementation method for origin tracking, which is a method for incrementally computing the relation between pieces of the program such as identifiers, expressions, or statements. However, this is presented only as a prototype for the construction of bi-directional mappings between source programs and optimized code.

More recent and daring work tackling debugging of dynamically optimized code has been reported [61, 71]. Kumar *et al.* [71] describe a set of techniques to monitor code transformations performed by a dynamic optimizer and to communicate this information to a native debugger. The steps are first to create a transformation descriptor of an instruction or data variable, then generate debug information, and to transmit it to the native debugger. The challenge consists in discerning between the optimized code and the optimizers dynamically, and to map it back with the source code, which is no longer available at runtime. A strategy to avoid this problem is offered by the

developers of Java HotSpot compiler [69] who interpret the unoptimized code during debug sessions. In [61] the focus is on reporting the expected values of source variables computed in the optimized code.

In the gcc compiler [53], generating debug information is possible via the option -g. Also, one can control the amount of information transmitted to the debugger by specifying the level, from -g0 to -g3. This option has been implemented in LLVM [77] and in the Clang front-end [29] and the result consists in populating the code represented in LLVM IR with a significant amount of metadata information, which is then transformed into debug information.

We have adopted a similar approach in tracking code from the source level to the intermediate representation, by marking interesting code regions with *metadata information*. This will be presented in more details in the next Chapter 5, Section 5.1.2.

The next step in performing multi-versioning is cloning, associated with the construction of a selection mechanism. Much work has been oriented towards this research direction [4, 27, 38, 49, 50, 56, 58, 59, 80, 83].

Cloning, multi-versioning, instrumentation by sampling. Multi-versioning is a widely adopted strategy to reduce the cost of code instrumentation by sampling [4, 27, 38, 56, 58, 83]. A selection mechanism periodically switches execution between a number of versions embedding instrumentation code and the original version. Efforts to reduce the runtime overhead and an efficient framework for instrumentation by sampling are presented in [4, 38]. Chilimbi and Hirzel [27, 58] add finer control on the sampling rate and eliminate redundant checks to decrement the overhead. They operate directly on the x86 assembly code using Vulcan [39] for capturing sequences of data references (dynamic executions of loads or stores).

An interesting use of sampling is presented by Chilimbi and Hauswirth [56] for checking program correctness. They develop an adaptive profiling where the sampling rate is the inverse of the frequency of execution of each code region. They adapt the framework introduced by Arnold and Ryder [4] to detect memory leaks. Marino *et al.* [83] extend this solution to multi-threaded programs to find data races.

Our goal is to create a static-dynamic framework that supports multi-versioning and sampling, by means of a generic runtime system that patches the code to enable various types of profiling, instrumentations and code optimizations.

Multi-versioning in optimizations. Fursin *et al.* [50] present a framework for continuous compilation within gcc for cloning code sections, applying various optimizations on the clones, and randomly selecting one version for execution. Evaluation is done using the *gprof* profiler. Luo *et al.* [80] use the Open64 4.0 compiler and the Interactive Compilation Interface [60] to select a limited number of optimized versions across all datasets, avoiding performance loss or code-explosion. Heuristic methods are employed to find a representative set of optimizations, and machine learning techniques correlate characteristics of the datasets with the optimized versions. Interactive Compilation Interface (ICI) [60] has been developed with the aim of providing access to the internal functionalities of compilers. Extensions to ICI [59] provide generic function

cloning, program instrumentation, pass reordering and control of individual optimizations. Patching is used to insert an event call before and after the execution of each version, either only for transferring information for further processing, or to change the selection decision of the compiler. In these regards, we have a very similar approach, as we insert callbacks to a runtime system to guard the execution of each code version. However, this framework implements multi-versioning and cloning at function level only, while we want to perform fine grain adaptiveness. For example, it does not allow to efficiently activate/deactivate instrumentation at loop-level, some iterations of a loop nest being instrumented while the others are not: the proposed approach would require a call to the runtime system in the body of the loop to be evaluated at each iteration, thus inducing a high overhead.

ADAPT [130] is a high-level adaptive optimization system. It proposes a domain specific language allowing the user to specify the heuristics for applying optimizations dynamically. ADAPT reads the descriptions and generates the executable code for a target application to apply the user-defined techniques. The optimization targets of ADAPT are the loop nests containing no I/O operations and no function calls. Considering the runtime information, a decision is taken whether optimization would be profitable. Applying the user defined heuristics, the set of optimizations is chosen and new code versions are generated. Before executing a code section, the framework verifies if experimental versions are available, otherwise the best known version is executed. Compared to our proposal, ADAPT requires a lot of source code modifications by the user, and to our knowledge no automatic multi-versioning using ADAPT has been reported.

The next part is dedicated to address the open questions and the limits of modern compilers, concerning multi-versioning.

4.2.2 Open questions and limits of modern compilers

Extending the intermediate representation vs using annotations. Compilers use an internal intermediate representation for manipulating code at compile time. The question that arises is how to delimit interesting code regions translated into this form.

The naive approach is to use *dummy* instructions as barriers. Namely, instructions that already exist, do not modify the semantics of the code, and are recognized by the compiler as marking the beginning and the end of the region. The great disadvantage is the implication that one has to find a particular instruction that takes this role in all cases, hence, it is not used anywhere else in the code, which is rather a strong assumption.

Another option is to extend the internal representation with additional instructions, having the role of barriers. The drawback is that the compiler has to be rewritten to accept new instructions. Moreover, these instructions might influence the code generator or even prevent some optimizations. Also, in both approaches, using barriers is not a viable strategy with higher optimization levels due to instruction reordering. On the other hand, since they are included in the set of instructions, they are not eliminated during optimization phases, which is a very important aspect, as explained in the following chapter.

At last, some compilers enable annotation inclusion in the intermediate representation, such as metadata carried by instructions. This seems to be the most inoffensive solution, as the code generator and the optimization phases are not disturbed. Thus, one can attach metadata to all instructions residing in the interesting regions, without disturbing the transformation phases or being influenced by instruction reordering. However, not all optimizations preserve the attached information and it might be difficult, after optimization, to recover the code originally marked for multi-versioning.

Our solution to this problem will be discussed in Chapter 5, Section 5.1.2.

Higher level *vs* lower level IR. Choosing a high- or a low-level intermediate representation is an open discussion, both forms presenting advantages and disadvantages. For code manipulation purposes, such as loop transformations or various high-level optimizations, preserving high-level information available in the source code significantly facilitates the process. On the contrary, for retrieving low-level information, for instance register or memory location accesses, one requires an intermediate representation that makes this information available or easy to track. Nevertheless, a low-level internal representation is not general enough to cover all architectures and is not suitable for most of the compilation stages.

In multi-versioning, choosing one representation or another might be a challenge, especially if each version is designed for a different purpose. For instance, one version may be tailored to perform low-level instrumentation, and another to apply the results of the instrumentation phase and perform high-level code optimizations. Therefore, each version should be manipulated into the most convenient representation.

We present our reasoning in these matters in the next chapter.

Communication between high- and low-level representations. Not only each version may be represented in a different IR, but, in addition, multi-versioning implies the presence of a runtime system, able to decide dynamically the version to be executed. Consequently, there are several situations where control flow between high- and low-level representations must be supported:

- communication between versions, in distinct representations
- communication between versions and embedding code, in distinct representations
- communication with the runtime system

Current compilers do not allow these types of communication, as they do not handle control flow entering or exiting lower level representations, such as inline assembly. Inline assembly is expected to 'fall through' to the following code. The gcc 4.5 compiler offers some support in these matters. Namely, it handles jumps from inline assembly to labels defined in C, but not the other way around. Also, jumps from an assembly code to another are not supported. To overcome this problem, one has to overwrite by hand part of the control flow graph – expressed in the higher intermediate representation – with branches generated from inline assembly code containing labels and jumps. Special attention must be given, as compilers do not parse the inline assemblies for their semantics.

In depth details concerning this approach will be given in Section 5.1.3 of Chapter 5.

Inserted code should not disturb the behavior of the original code. Inserting additional code, such as instructions for tracking or the mechanism to switch between versions, might have a negative impact on performance. Compilers must be tailored to generate multiple versions in a manner which minimally influences the behavior of the original code and does not degrade the result of the optimization phases. On the other hand, aggressive optimizations could lead to alterations of the inserted code.

Compilers must efficiently manage the interplay between the inserted code and the optimization passes.

We address this aspect throughout all stages of code manipulation. Details can be found in Chapter 5, Section 5.1.2.

4.2.3 Adapting to the current behavior using multiple versions

In our approach, multi-versioning is employed in generating three types of versions for each loop nest:

1. original version,
2. instrumented version,
3. optimized versions (parallel code patterns).

As presented in Chapter 3, Section 3.2, we build the polyhedral representation of a loop, by instrumenting samples of iterations. In order to monitor the code, we generate at compile time an instrumented version that acquires the required information. However, note that this version executes only a subset of the iterations of the loop nest. Provided that the loop can be optimized with a polyhedral transformation, a parallel transformed version continues the execution. The parallel version is generated dynamically from the parallel code patterns introduced in Chapter 3, Section 3.3.1. As shown previously, during its execution, a code might display several phases, characterized by different memory accessing behaviors. Therefore, the selected optimization strategy must accord well with the new behavior and new versions must be generated as soon as a change is detected. For example, memory accesses can suddenly follow new access patterns, hence requiring new data locality optimizations. When performing speculative parallelization, the current parallel schedule can suddenly incur too many rollbacks due to numerous memory access conflicts. In this case, a new parallel schedule has to be applied. Otherwise, if no loop transformation is possible, the original version executes a subset of iterations and the process repeats.

To complete the execution of the code, while adapting to the current phase, one needs to link different versions of the same code. Each of the versions will be launched to execute a subpart of the code and another one will continue, as in relay races.

This is illustrated in Figure 4.1. The instrumented, original, and two parallel code patterns are built at compile time, and at runtime one or another version is selected

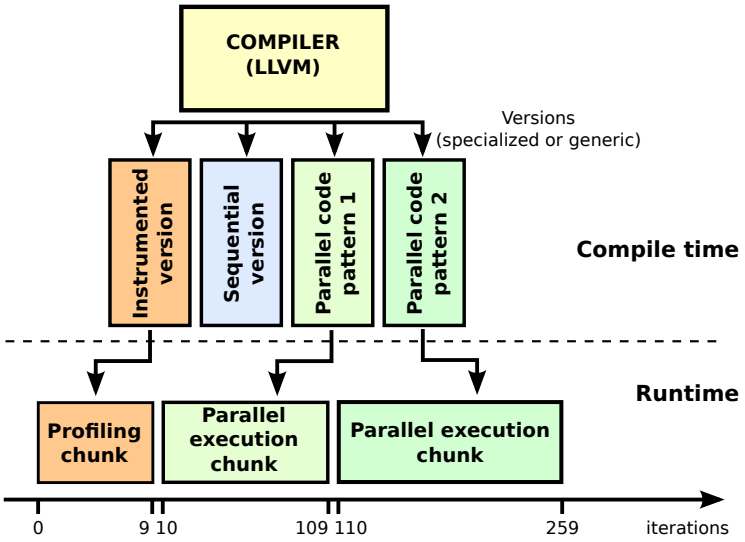


Figure 4.1: Alternating the execution of different versions, during one run of the loop nest (courtesy: Ph. Clauss)

to be executed for a number of iterations. Linking the execution of different versions is possible thanks to our chunking mechanism, presented in the next section.

4.3 The chunking system

To adapt to the current behavior of a code during execution, we introduce the *chunking system*. We define a loop chunk by a set of *consecutive iterations of the outermost loop* of the nest. For instance, the first chunk contains the first 10 iterations, the next chunk iterations from the 11^{th} to 70^{th} and so on. The execution is orchestrated such that each chunk continues the execution from the iteration where the previous chunk ended. Hence, a target code is run as a *sequence of chunks*, each of them embedding a *different version* of the code (instrumented, parallel or original). The frontiers between chunks give room for decision-making about the nature of the next chunk, by using information collected during the execution of the last chunk. When applying a speculative optimization, the decision can be to re-execute the last chunk using another code version if the previous version incurred incorrect computations or to continue with the same schedule, if the parallelization was successful. In this manner, one can execute parallel chunks followed by sequential chunks, without missing any parallelization opportunities in *partially parallel loops*.

The technique of slicing the loop in several chunks to exploit partial parallelism has been previously employed in several works [28, 33, 48, 85, 105]. Spice [105] and SpiceC [48] chunk the iteration space of a loop in intervals of successive iterations, depending on the number of processor cores. The chunk size is determined dynamically, with respect to the load balancing, using value speculation. By predicting the values of a certain variable every n iterations, rather than each iteration, the chances of a good prediction increase. Cintra and Llanos [28] discuss a speculative system based on chunks with different strategies. One can divide the iteration space in as many chunks

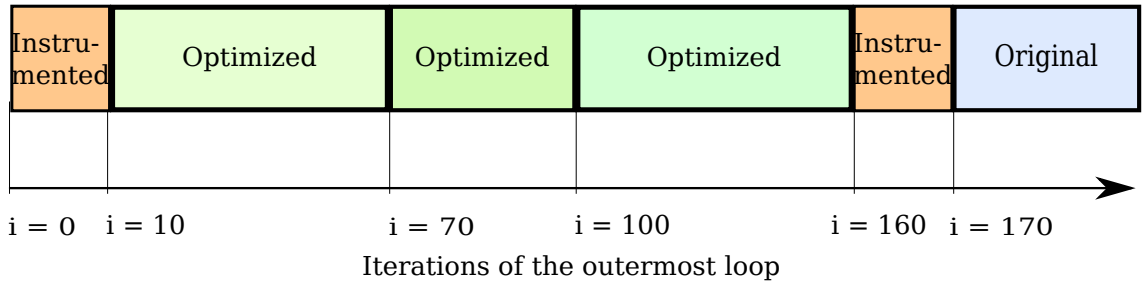


Figure 4.2: Loop chunking

as the number of cores and assign the chunks statically, one chunk per processor. Another approach is to create more chunks than there are processors and to use a scheduler to assign them dynamically.

In contrast to previous works, where the chunks are executed in parallel, but intra-chunk iterations are run sequentially, we provide the means to execute the chunks one after the other, while intra-chunk iterations can be executed either in parallel or sequentially.

A similar approach is the R-LRPD test [33], which applies a method that divides a loop exhibiting partial parallelism into several smaller loops which are speculatively parallelized, using the LRPD test [106]. This resembles the one we propose, since the chunks are executed successively and they execute the corresponding iterations in parallel, if possible, or in sequential order, otherwise. The advantage is that, although the iterations of some chunks are executed sequentially, this does not prohibit parallelism on the iterations of other chunks. More details on the LRPD and R-LRPD tests were presented in Chapter 1, Section 1.2.

Exploiting partial parallelism in loops has already raised the research interest of the TLS community. Nevertheless, to our knowledge, no prior works target optimizing each chunk of the loop using a different optimizing strategy, adapted to the current behaviour of the code. For achieving this, our strategy is to embed different loop versions in successive chunks, such that their chaining completes the whole computation. Under these circumstances, the loop bounds for each chunk have to be parametrized and carefully set. Using a common iterator among multiple versions of the loop, each chunk can embed a different version, as the loop execution advances. This mechanism is shown in Figure 4.2. The next subsections describe how the versions are tailored to be embedded in successive chunks.

4.3.1 Chunking one sequential loop

Considering just one unique loop version, splitting it into several consecutive chunks is straightforward and similar to variable-size strip-mining: a loop index has to be introduced ranging from the start to the end of a chunk. Its bounds are parameters that are set when the position and the size of the chunk have been decided. This is done at runtime in our framework. Obviously, these new chunk exit conditions have to be carefully merged with the initial exit conditions of the loop:

```
for (chk=chk_start ; chk<chk_start+chk_size ; chk++)
```

```
{
  if (original_exit_conditions) break;
  /* original loop body */
  /* original incrementation (if necessary) */
}
```

The initialization of the original variables must be performed before launching the first chunk, and the context has to be preserved between successive chunks. In our framework, this is achieved by transmitting the updated information from one chunk to another. Notice that any kind of loop, either *for* or *while* loops, can be chunked in this way, by transforming them systematically into *for* loops and inserting the original loop exit conditions and control inside the loop body.

4.3.2 Chunking with transformed loops

When considering different kinds of code versions, some of them can be the result of significant code transformations. The chunk index and its bounds must be properly inserted inside the code of each version, such that a new chunk embedding any version continues the execution of the previous one.

Classic loop parallelization Optimizing and parallelizing loop transformations have been studied intensively in the last decades. Classic parallelizing transformations, as those considered in the OpenMP library, split the parallelized *for*-loop into blocks that are distributed to threads following one or another schedule, as static, cyclic or dynamic. In our framework, such transformations are applied on the chunked version of the original loop and the loop is split along the chunk index range. In this way, the chunk index is naturally included in the transformation, resulting in a chunk embedding a parallel code. The chunk exit condition is then properly considered and other versions, either sequential or differently parallelized, can precede or follow. However, special attention has to be paid when the exit of a chunk has been determined by the original exit conditions. Since the parallelization has been done by considering the chunk index, the original exit conditions may have been reached by one unique thread, while the other threads performed supernumerary computations. Hence the whole chunk has to be rollback-ed and re-executed using another code version, typically the original sequential one.

Advanced loop transformations More advanced transformations consider a whole loop nest and may result in a completely new schedule of the original iterations. Such transformations are for example loop interchange, tiling and skewing. More generally, these transformations are linear transformations of the original loop nest where new indices with new bounds are computed [15]. In the same way as before, the chunk index is naturally included by applying the linear transformation on the chunked version of the loop nest.

As an example of a linear transformation of a loop nest, let us consider the transformation of the 2-depth nest from Listing 4.2, whose indices i and j range from 0 to $N-1$. Since the outermost loop has been chunked, it is also bounded by *chk_start*

Listing 4.2: Original loop

```
for (i = 0; i < N; ++i)
    for (j = 0; j < N; ++j)
        ...
```

Listing 4.3: Chunked loop

```
for (i = chk_start; i < chk_start + chk_size && i < N; ++i)
    for (j = 0; j < N; ++j)
        ...
```

and $chk\_start + chk\_size$, as shown in Listing 4.3. Without loss of generality, let us consider these latter bounds as additional bounds on index i. Consider now the transformation defined by matrix $T = \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix}$: the initial iterators $\begin{pmatrix} i \\ j \end{pmatrix}$ are transformed into new iterators $\begin{pmatrix} x = i - j \\ y = j \end{pmatrix}$. Since x is the index of the outermost loop, its constant bounds are defined by the minimum and maximum values of $i - j$. Those values are respectively $chk\_start - N + 1$ and the minimum between two values, $min(chk\_start + chk\_size - 1, N - 1)$.

Index y has bounds depending linearly on x. Moreover, these bounds are expressed as maximum or minimum of several values, since:

$$\begin{cases} 0 \leq j \leq N-1 \\ 0 \leq i \leq N-1 \\ chk\_start \leq i \leq chk\_start + chk\_size - 1 \\ j = y \\ i = x + y \end{cases}$$

$$\Rightarrow \begin{cases} 0 \leq y \leq N-1 \\ -x \leq y \leq N-x-1 \\ chk\_start - x \leq y \\ y \leq chk\_start + chk\_size - x - 1 \end{cases}$$

$$\Rightarrow \begin{cases} max(chk\_start - x, 0) \leq y \\ y < min(N, N-x, chk\_start + chk\_size - x) \end{cases}$$

The computation of new loop bounds uses the Fourier-Motzkin elimination algorithm, as illustrated in Listing 4.4. Notice that the transformed and chunked loop nest can then be parallelized using one of the classic strategy by splitting one parallel loop level into blocks.

When the original loop nest is characterized by loop bounds that are linear functions of the enclosing loop indices and the body statements are exclusively composed of array references that are also linear functions of the indices, such a linear transformation can directly be applied to result in a completely transformed code. However, our framework considers any kind of loops containing any kind of instructions and exit conditions,

Listing 4.4: Transformed chunked loop

```
for (x = FM_low_x ; x < FM_upp_x ; ++x)
    for (y = FM_low_y ; y < FM_upp_y ; ++y)
        ...
```

```
/* chunk 1 */
i=0;
for (chk=0 ; chk<10 ; chk++)
{
  if (i>=100) return(1); // early exit
  for (j=0 ; j<100 ; j++)
    a[i][j]+=1;
  i++;
}
return(0); // normal exit

/* chunk 2 */
for (x=-89 ; x<16 ; x++)
{
  if (x+y>=100) return(1); // early exit
  for (y=max(10-x,0) ; y<min(100,100-x,16-x) ; y++)
    a[x+y][y]+=1;
}
return(0); // normal exit
```

Figure 4.3: Chunked versions example

as for instance indirect memory references. In this case, a linear transformation is necessarily speculative and based on predictions. As a final pedagogical example, consider the previous loop nest, with N=100:

```
for (i=0 ; i<100 ; i++)
  for (j=0 ; j<100 ; j++)
    a[i][j]+=1;
```

Let there be two chunked versions: the first one which is the sequential original one where the chunk index has been added as shown in the previous subsection, and the second one which is the version resulting from the previous example of linear transformation. If we consider two successive chunks where each embeds a different version, and where the first chunk ranges from 0 to 9 and the second chunk from 10 to 15, the chunked versions shown in Figure 4.3 are launched successively.

One can verify that the first chunk updates the 10 first rows of the array. The second chunk first updates element `a[x+y][y]` where $x = -89$ and $y = max(10 + 89, 0) = 99$, *i.e.*, `a[10][99]`. This is the unique iteration with $x = -89$. Then, with $x = -88$, two iterations update elements `a[10][98]` and `a[11][99]`, and so on until element

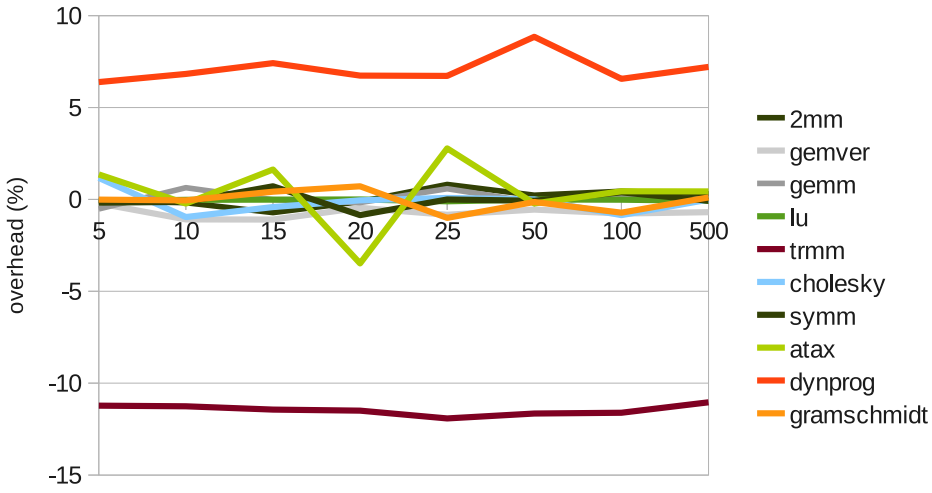


Figure 4.4: Chunking time overhead

`a[15][0]` is finally updated when $x = 15$ and $y = 0$. Rows 10 to 15 are updated by the second chunk. Globally, both chunks update rows 0 to 15.

Time overhead (execution by chunks) We measured the time overhead induced by executing by chunks ten codes containing loop nests from the PolyBench suite of programs [95]. This suite is made of kernels commonly used in scientific and multimedia codes. We fixed the problem sizes to values inducing significant execution times and compared the original execution time with the times of executing the loop nests with different chunk sizes, varying from 5 to 500. Results are represented in Figure 4.4. One can observe that the time overhead is very low and stays stable around zero for most of the codes. A significant speedup is even obtained with `trmm`, about 11%, since chunking can be assimilated to strip-mining the outermost loop. However, `dynprog` has more significant time overheads, around 6-8%. This is due to the added chunk bounds that prevented the compiler to apply the same optimizations that were applied with the original constant loop bounds. We conclude that the chunking mechanism we propose does not incur an overhead too high for the types of codes that we target.

4.3.3 Rollbacking with chunks

Slicing the loop in consecutive chunks, allows our system to periodically validate the execution, after each chunk completes its iterations. The sequential chunks (either running the original or the instrumented version) are automatically validated, since their execution is not speculative, whereas the parallel versions include the verification code which sets a flag upon encountering a misspeculation (Chapter 3, Section 3.4.2). We recall the reader that prior to launching a speculative chunk, a memory backup is performed, saving the memory area predicted to be altered by the chunk. Then, should a faulty chunk be detected, the memory state is restored from the safe copy and the iterations of this chunk are re-executed sequentially. Consequently, the size

of the chunk embedding a speculatively parallelized version has a great impact on performance, from several standpoints.

By launching speculative chunks of a small size, one ensures that the rollback costs are not too high, since the number of iterations that are canceled and re-executed is low. On the other hand, this strategy shows its disadvantages on other aspects, such as load balancing, granularity of parallelism, overhead of the threads and of the communication with the runtime system between the chunks. Executing the loop nest in a multitude of small chunks might have a negative impact if the work load assigned to each thread is too low to compensate for the overhead incurred by the creation of threads and by the exchanging of information. Similarly, a large number of chunks requires frequent interruption of the execution and communication with the runtime system, which influences the performance. Consequently, we have evaluated several strategies concerning the dynamic adjustment of the chunk size, as presented below. Additionally, depending on the observed behavior of the code, several scenarios can be considered regarding the nature and the size of the next chunk. Given that the loop's behavior is stable one could increase the chunk size with a given step. On the other hand, since the chunk executing the last iteration of the loop must be run sequentially to ensure that all iterations have been correctly computed (see Chapter 3, Section 3.4), another strategy is to keep the chunk size fixed, or even decreasing as it approaches the end of the loop (which is unknown in most of the cases).

We compared the efficiency of four different chunking and rollback strategies, with an increasing chunk size. In each scenario, the profiling, sequential and parallel chunks have the default initial sizes of 10, 100 and 100, respectively. Their characteristics are:

1. When launching a chunk of the same type as the previous one, the chunk size is doubled. In case of a rollback, a sequential chunk of the same size as the rollbacked chunk is launched, to re-execute the canceled iterations. Then a profiling chunk is run.

2. Similar to the previous strategy, the chunk size is doubled when the predicted behavior is unchanged. However, when a rollback occurs, we launch directly a profiling chunk. As a consequence, a parallel chunk of the same type as before might be executed, having the default size. If the first parallel chunk is invalidated, then a sequential chunk is launched, which will necessarily overcome the rollback point. This strategy aims to overcome the problem of very large rollbacked chunks, which can still be executed in parallel, until an earlier point.

3. The third strategy replaces the algorithm of doubling the chunk size, by a fixed size increment. This technique is aimed to test whether slower increases of the chunk size lead to lower rollback costs. As in the first strategy, in case of a rollback, the following chunk executes the iterations serially.

4. Finally, the fourth strategy combines the method of an incremental chunk size and the technique of initiating a profiling as soon as a rollback occurs.

These strategies have been tested on a benchmark that exhibits several phases of a loop nest and requires a significant number of rollbacks. The pseudo-code is illustrated

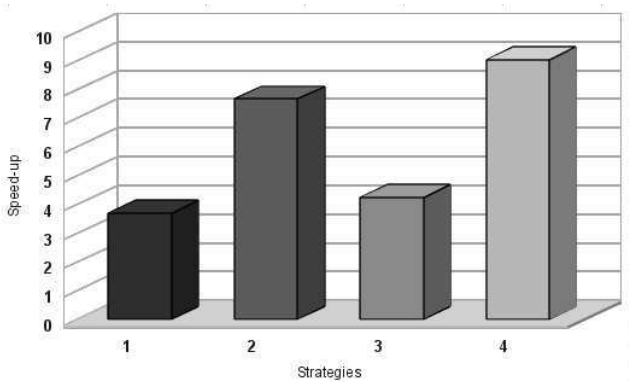


Figure 4.5: Speed-up obtained with various chunking strategies

in Chapter 3, Listing 3.5. And our findings are depicted in Figure 4.5, which shows that the incremental approach followed by an instrumenting phase is the most promising one, nevertheless, the optimal strategy is highly dependant the code behavior.

Many other combinations of such strategies can be imagined, such as trying to detect as precisely as possible the first misprediction point encountered by the threads. In consequence, one can launch first a parallel chunk with a static schedule or with dynamic or guided schedule, in which the slice of iterations allocated to each thread is fixed. Such that, one can compute the window where the misprediction point occurs, depending on the faulty iteration it, on the number of threads th_{num} and on the size of the slice allocated to each thread $slice$. Thus, one can estimate that after the rollback occurs, it is safe to launch a parallel chunk with the upper bound given by $it - th_{num} * slice$. However, for a precise computation, one requires to assign small slices to each thread, which may damage performance.

In short, to keep a general approach suitable to any code, the most interesting strategies prove to be the ones that either keep the chunk size fixed or increment it with a fixed, small step. Moreover, after a rollback is initiated, the strategy we propose is to launch a sequential chunk of the same size as the faulty one, in order to overcome the rollback point. This method has a positive impact both on codes that exhibit a stable behavior, and also on the ones that require rollbacks, since the number of canceled iterations do not increase over a given threshold. The default size of a parallel chunk is set depending on the total number of iterations of the outermost loop, if known statically, otherwise it is set to 100 iterations.

4.3.4 Profiling with chunks

The chunking technique puts in place a mechanism to perform instrumentation by sampling, by alternating the execution of the instrumenting versions with the original or the optimized ones. Since instrumentation incurs an overhead, one would want to limit it, by controlling the size and the frequency of the instrumented chunks. As in the previous section, this is a matter of choosing the optimal chunk size and a suitable scenario for launching an instrumented chunk.

For the purpose of generating interpolating linear functions, one requires a minimum

of three iterations per loop, in which the monitored instruction is executed. Thus, one can use the first two acquired values in each loop, to compute the linear function depending on the iterator of the loop, and the third value to verify the correctness of the function. Nevertheless, since some codes include conditional branches, a higher number of iterations is required. Empirical tests show that setting the size of the instrumenting chunk on 10 iterations suffices to capture the behavior of most codes. Provided that the target loop executes a high number of iterations, the cost of the instrumentation is negligible in practice. The frequency of restarting the instrumentation is dictated by the behavior of the loop. Thus, a new instrumentation phase is initiated after each sequential chunk, in the hope of discovering parallelization opportunities. Either the behavior of the loop changed and a rollback was initiated, followed by a sequential chunk, or no parallelism was possible and an original version was launched.

To improve the runtime footprint of the instrumentation, one could disable the instrumentation on the instructions for which the interpolating linear functions have been computed, and continue the instrumentation only for those which were not yet executed sufficiently many times. Nevertheless, in practice this strategy could incur a higher overhead, due to many conditionals that verify which instructions still require monitoring.

All in all, the simple approach of monitoring all interesting instructions for a fixed, but small number of iterations, already yielded good results. Several benchmarks evaluating this strategy are presented in Chapter 6, Section 6.1.

4.4 Conclusions

Combining the techniques presented in this chapter results in a powerful and efficient strategy for performing dynamic loop optimizations that adapt easily to any execution context and changing behavior of the code. This emphasizes two of the important advancements that we provide:

- Support for *partial parallelism*, since one chunk of the loop can be run in sequential order, followed by another chunk executing in parallel
- Parallelizing a loop with *several different parallel schedules*, during one run.

Although we applied them on loop nests, the methods described above can be generalized to find their applications on any types of code, from linear codes to the ones exhibiting recursions. Similarly, each code version could be tailored for various other purposes. Beyond boosting performance, one could target for instance security, by generating several versions of code which are alternated randomly to complete the execution, such that their behavior is not predictable.

Chapter 5

The VMAD framework

The proposal is implemented and evaluated in a collaborative static-dynamic framework (VMAD), depicted in Figure 5.1. VMAD consists of a static component, which tailors the code at compile time, and a runtime system, which orchestrates the execution. The static component is built on top of the LLVM compiler suite [77], while the runtime system was developed to handle x86_64 binary code.

At compile time: Speculative code parallelization starts, in our approach, at the level of the original source code, where the programmer guards interesting loops with a specific pragma.

```
#pragma speculative_parallelization {

  loop 1
    loop 2
     ...
}
```

Our compiler extensions identify the loop nests marked for parallelization and generates several versions of these code regions. Namely, one version is customized for the profiling phase. It includes instrumentation code capturing dynamic data such as the accessed memory locations, values of the basic scalars and the number of iterations performed by the subloops. Our goal is to build linear functions to interpolate each sequence of memory addresses being accessed and all successive values taken by each basic scalar variable, whenever possible, as explained in Chapter 3, Section 3.2.

Several other versions are built, each of them representing a code pattern, designed for a class of parallelizing transformations, as presented in Chapter 3, Section 3.3.1. Also, the original version of the loop is preserved, in case no valid parallel code can be generated. Additionally, a switching mechanism is embedded, called the *decision block*, which controls the alternation of these versions at runtime. The *decision block* contains a callback to the runtime system, which decides which version to be executed next. Using the chunking system presented in Chapter 4, Section 4.3, each version will be executed in one chunk of the loop, and they will alternate until the loop execution

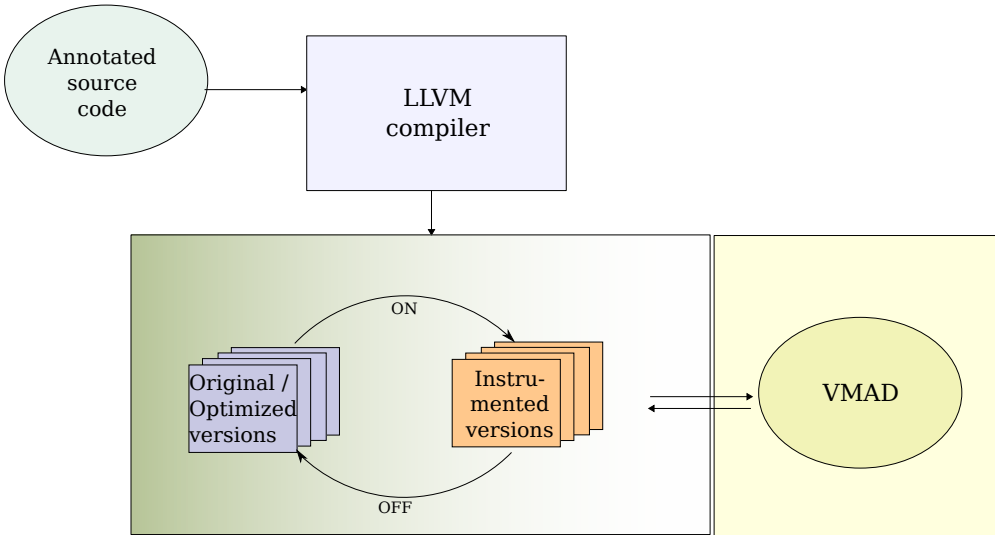


Figure 5.1: Framework overview: static-dynamic collaboration.

completes. The versions are preceded and followed by callbacks to the runtime system, enabling it to process the information acquired during the profiling phase, to validate or invalidate a speculative execution, or to decide upon the next version to be launched. Moreover, the compiler appends to the binary file information statically available, to be transmitted to the runtime system. This includes data about the loop nests, such as the loops' hierarchy and depth, addresses of the callbacks in the code or values of some parameters.

At runtime: At start-up, the runtime system is first invoked when the execution reaches the decision block to select among the code versions. Then, it initiates a profiling phase, by launching the instrumented version to execute a chunk of the loop, of reduced size. During this step, the runtime system collects data and builds interpolating linear functions to describe the behavior of the loop, and computes distance vectors, as a lightweight dynamic dependence analysis (Chapter 3, Section 3.2.1). When the chunk finishes its execution, the control reaches the decision block again. The runtime system evaluates the results of the dependence analysis, and decides if a parallel code version can be launched and which is the corresponding code pattern. Next, it patches the pattern to generate the appropriate code and launches a parallel chunk. During its execution, the code is monitored thanks to the verification system, to ensure its correctness (Chapter 3, Section 3.4). If verification succeeds, the chunk is validated and a new chunk is launched, with the same parallel code. Nevertheless, as soon as a misprediction occurs, the runtime system is invoked to rollback the faulty chunk and to restore the memory to a correct state. Next, a sequential chunk is launched, to overcome the rollback point, followed by an instrumented chunk which restarts the process for speculative parallelization.

A more detailed view of the VMAD framework is given in Figure 5.2.

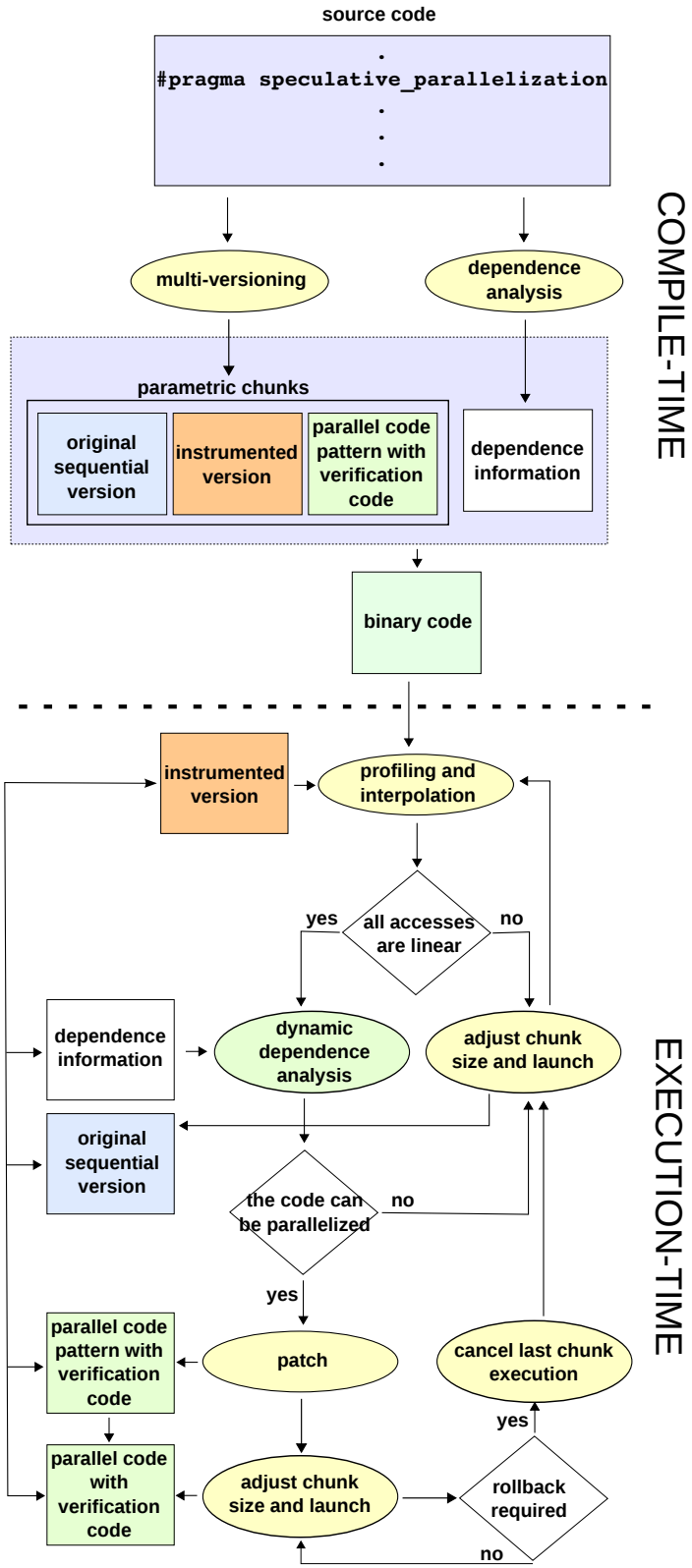


Figure 5.2: Framework overview (courtesy: Ph. Clauss)

5.1 Static component

Our framework relies on a dedicated `pragma` to mark the interesting loops in the source code. We have extended the Clang front-end [29] of the LLVM compiler suite [77] to process the `pragma` and to preserve its semantics in the compiler's intermediate representation (IR).

5.1.1 Short introduction of the LLVM IR

The LLVM Intermediate Representation (LLVM IR) is built upon the Static Single Assignment (SSA) form and it confers type safety, low-level operations and flexibility. An increasing number of high-level languages may be translated into the LLVM representation, which is the internal language used throughout all phases of the LLVM compilation strategy.

The LLVM code representation is designed to be light-weight and low-level, for efficient compiler transformations and analyses. On the other hand, it provides type information and supports mapping higher level information from the source code, which facilitates the development of various optimization passes [73].

Although LLVM IR already offers support for embedding high-level information, new methods have been developed to include annotations or debugging information by attaching metadata to the IR.

Until LLVM 2.6, debug information represented a channel from the front-end to the DWARF emitter, without being included in the executable code. Moreover, it was encoded using global variables with tags, which prevented a number of optimization passes and was very expensive from the time and memory footprint viewpoint.

An important enhancement in these regards was brought in LLVM 2.6, with the development of the metadata. The main goals were to provide the means to attach information in the IR, without influencing the optimizers (unless metadata was explicitly specified for this). Also, the cost, in terms of time and memory use, has significantly been reduced. Metadata is attached to instructions and improves the implementation of the debug information.

The optimizers do not have to be aware of the metadata, but the other side of the coin is that they do not preserve it. Hence, metadata information might be lost in code transformations. Special care might be taken to update metadata information during optimization phases, nevertheless, this is not suitable to all passes. Especially when aggressive code transformations are performed, such as with -O3 optimization level, tracking code and metadata is particularly difficult.

LLVM provides a wide range of functionalities to facilitate code transformations. Regarding multi-versioning, LLVM offers support for cloning, however limited. There exists a suite of *clone* utilities, able to create copies of instructions, basic blocks or functions, but no correlation is made between values in the source and in the clones. Therefore, LLVM cloning can only be applied in some very specific situations.

Although the LLVM IR provides interesting information, low level instrumentation remains impossible for several performance related mechanisms, such as tracing memory behaviors. This is due to the fact that LLVM IR uses an infinite number of virtual registers, which will be later mapped either to physical registers or to memory

```
#pragma speculative_parallelization
{
   while( p != NULL ){
        p->val = ..
        p = p->next();
    }
}
```

Figure 5.3: Annotated source code

locations. As registers are not yet allocated in the LLVM IR, one cannot distinguish whether the LLVM "load" and "store" instructions represent memory or register accesses. Additionally, LLVM IR is in static single assignment (SSA) form [32], which simplifies the analysis of the control flow graph, but introduces a number of unnecessary "load" and "copy" instructions. These are eliminated when generating the code for a specific target architecture. Also, SSA ϕ instructions are not supported by traditional instruction sets, hence the compiler replaces them with instructions preserving their semantics, but which are not present in the LLVM bytecode [78].

In what follows, we present the steps taken by the compiler for preparing the code for the interaction with the runtime system, in order to speculatively parallelize the loops. The first step consists of identifying the region marked in the source code. Once aggressive compile time optimizations have been applied, the optimized code might not resemble the original source code, as discussed in Chapter 4, Section 4.2. The implementation details on overcoming this problem are presented below. Next, the compiler generates several versions of the code region and prepares them either for instrumentation or for parallelization. Since in our framework we parallelize loops using code patterns prepared at compile time (Chapter 3, Section 3.3.1), we dedicate a consistent part of this section explaining the construction of the code patterns, giving all the implementation details. Finally, we provide information on the means of communication between the code and the runtime system.

5.1.2 Identify the loop nests marked for speculative parallelization

Extending Clang Recall that our goal is to enable the compiler to create a series of code versions, among which the runtime system can dynamically select one or another for execution. Hence, for specific code regions, different versions (original, instrumented, code patterns) are prepared statically. In this respect, a new `pragma` is introduced, namely "*#pragma speculative_parallelization*", which is inserted in the source code for delimiting the code regions to be instrumented, as shown in Figure 5.3.

Our system extends the LLVM [77] compiler as well as Clang and Clang++ [29] front-ends to handle our `pragma`. The C/C++ code is translated into the LLVM bytecode and the `pragma` delimited region is marked by using metadata information.

The steps required for extending Clang to be aware of a new `pragma` imply first

to augment the parser, by defining a new class for the `pragma`. The class contains a specification for the `pragma` handler, which describes the action to be taken, once the compiler encounters the `pragma` in the source code. The handler verifies the syntax of the `pragma` and calls the corresponding action, which has to be available in the list of actions taken by the compiler and to define the semantics of the `pragma`.

For instance, a `pragma` may be attached to a specific structure such as a compound statement, a function or a loop. In this case, the action consists of marking this particular structure with a symbol indicating the existence of the `pragma`. In this respect, we have defined a new data structure *PragmaCollector* which keeps track of all associated `pragma`s. The next step is code generation. As we are extending the LLVM compiler, the source code is converted into LLVM IR. The decision to be taken at this point is whether to create a new instruction for the `pragma`, to use LLVM intrinsics, annotated attributes or the LLVM metadata. Nevertheless, there are both advantages and disadvantages presented by each of these solutions, as discussed below:

First, adding a new intrinsic or a new instruction in LLVM, designed with no other purpose, but to mark the beginning and the end of the region. However, a number of drawbacks result from this approach:

1. Adding new instructions is discouraged in LLVM, as all passes have to be updated and maintained to work with the new functionalities. Since LLVM already includes a considerable number of analysis and transformation passes, this would surely lead to a significant amount of work [20].

2. In case a new functionality can be expressed as a function call, adding an intrinsic is a more elegant and simple solution. Intrinsics do not require updating the optimizers, but the LLVM IR and the code generator must be extended to support it. However, if the intrinsic does not have any side-effect, the optimizers will remove it.

3. Barriers are a reliable solution when no optimizations are applied, but become unsafe with a higher optimization level. The strongest argument against using barriers is that instructions belonging to the region might be hoisted above or sank below the barriers (or, vice-versa, instructions that did not originally belong to the region, can be included), as depicted in Figure 5.4.

Second, attaching metadata to all instructions in the code region. This strategy gives an answer to all the problems displayed above, since LLVM already offers support for metadata, it does not influence the optimizers and it is not disturbed by instruction reordering.

In our work we use the metadata based method. The difficulties of tracking code throughout optimization phases is that metadata information is not preserved, and that code suffers significant transformations. For instance, if one marks the instructions building up a loop, after running the loop optimizations, additional code is included (*e.g.* due to loop fusion) or excluded (*e.g.* loop invariants, loop split) from its original body. Therefore, identifying the original instructions is not always possible. Focusing on loops, the conservative solution we propose is to consider that the original loop is transformed into the code region containing:

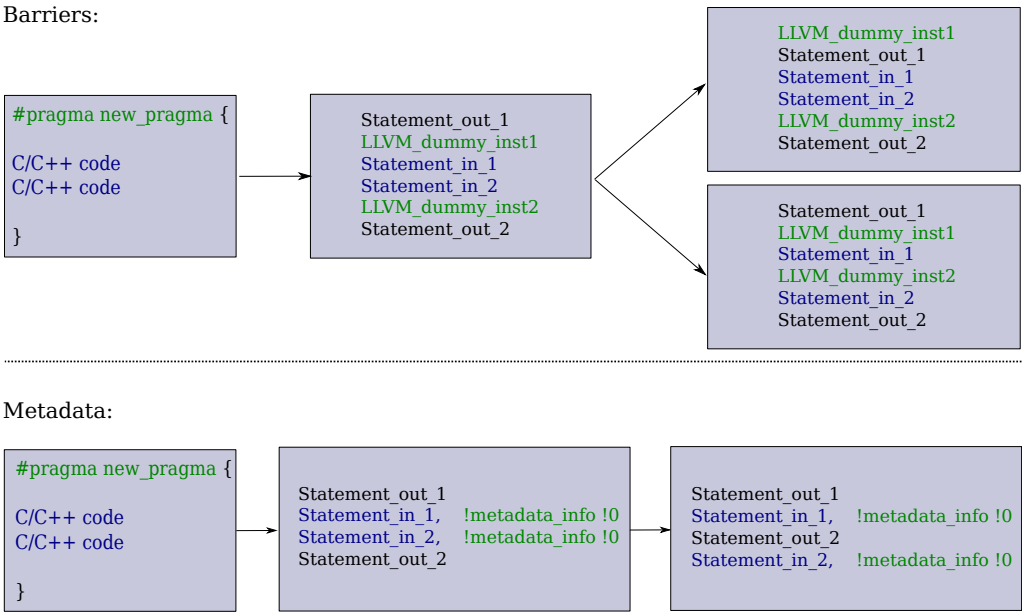


Figure 5.4: Delimiting code regions using barriers compared to metadata

- all loops that include ...
  - at least one basic block containing ...
    - at least one instruction that carries metadata

The consequence is that more code than the one originally marked for multi-versioning is considered. However, in this manner, we ensure that all instructions of the targeted code region are safely enclosed. An example of a loop represented in the LLVM IR, delimited by barriers or using metadata information is displayed in Figure 5.5.

We design an LLVM pass taking specific actions, with respect to the semantics of the metadata, as described in the next section.

5.1.3 Generating multiple versions

An LLVM pass operates on the generated bytecode, with the aim of selecting and processing the regions.

Cloning Once the region is identified, several clones are created. In LLVM IR, a set of restrictions is strictly imposed as SSA form must be preserved:

1. Instructions and their return values are equivalent. Hence, in the example:

   ```
   %tmp = load i32* %i, align 4
   %inc = add i32 %tmp, 1
   ```

   instruction `%inc` uses the value stored in `%tmp`. When cloning using the LLVM *clone* functions, the result is:

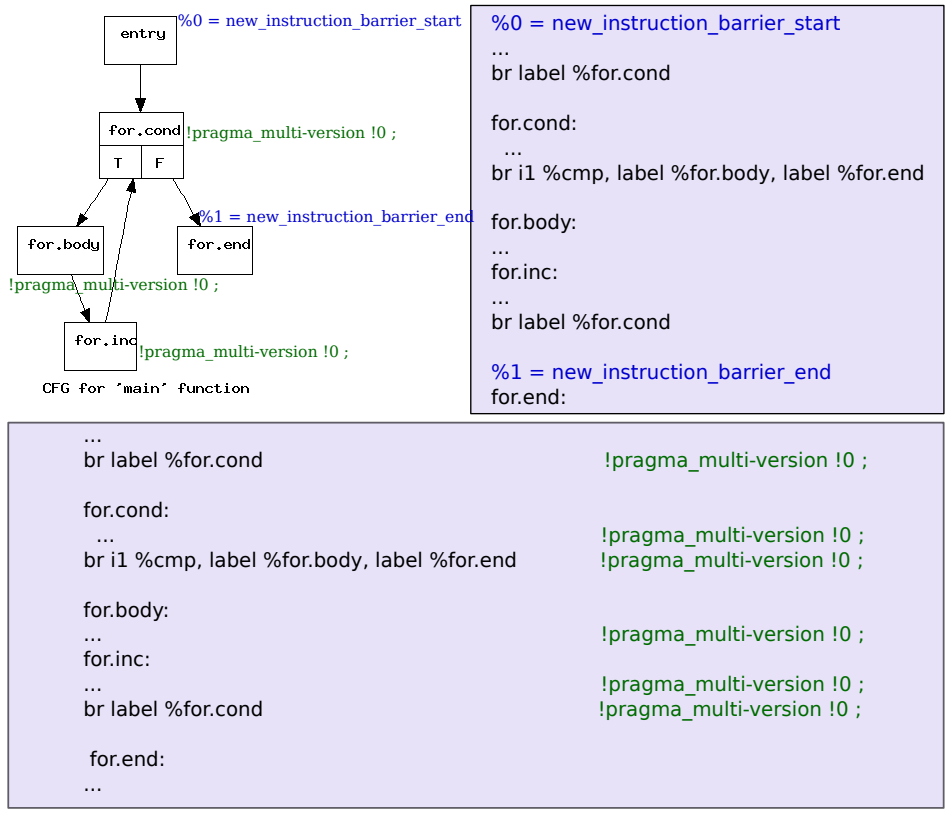


Figure 5.5: Loop in LLVM IR with metadata

```
%tmp_clone = load i32* %i, align 4
%inc_clone = add i32 %tmp, 1
```

whereas the target is:

```
%tmp_clone = load i32* %i, align 4
%inc_clone = add i32 %tmp_clone, 1
```

Note that instructions outside the region are not cloned (`%i`). Nevertheless, they may be used both by instructions belonging to the region and by their clones. For instance instructions `%tmp` and `%tmp_clone` use `%i`, but `%i` is not cloned.

2. Each value (instruction, basic block) must have a unique parent. It cannot be duplicated in the same function, nor copied in a new one, unless it is removed from its parent function. For this reason, we cannot simply insert each value twice, but we need to create and maintain individual clones.

3. Each value must dominate all its uses.

Our proposal for cloning is to create a map between all instructions and their clones; similarly, for all cloned basic blocks. In figure 5.6, blocks BB1 to BB4 belong to the region marked for multi-versioning and BB1_clone to BB4_clone are their clones.

As clones are created, a map containing the pairs of original and cloned values is maintained, *e.g.* BB1 → BB1_clone. Using the *clone* function available in LLVM, each instruction or basic blocks will use the same values as its original version. Hence blocks BB1_clone to BB4_clone will point to blocks BB1 to BB5, instead of using the cloned versions of the blocks from the region. Namely, they should point to BB1_clone to BB4_clone, since these are the corresponding clones, and to BB5. As BB5 does not belong to the region, it is used both by BB3 and BB3_clone.

The objective is to rebuild the control flow graph between the clones, as illustrated in Figure 5.7. Similarly for instructions, each clone that uses a value (either instruction or basic block) from the region is updated to use its corresponding cloned version. For instance, block BB1_clone branches to block BB2. As BB1_clone is identified as a clone version, and BB2 is an original version belonging to the region, the edge BB1_clone to BB2 is suppressed and replaced by the edge BB1_clone to BB2_clone. In other words, the clone BB1_clone is updated to use the value BB2_clone, instead of the original version BB2. On the other hand, a clone version is allowed to use an original value which does not belong to the region: BB3_clone branches to block BB5.

When clones are created, they are not automatically assigned a parent. We do this manually, by inserting each cloned basic block in the same parent function as the original version. In the case of instructions, they are inserted in the corresponding clone of the basic block, and not in the original block.

With this, we achieve to create a copy of the code marked for multi-versioning, while fulfilling the above mentioned constraints.

As presented in Section 5.1.1, one needs to convert the LLVM bytecode marked for instrumentation into x86_64 assembly code. The code which is not instrumented is represented in the LLVM IR, conserving the higher level information for other code manipulations. On the other hand, the regions marked for instrumentation are cloned and extracted into new functions, since a function is the minimal unit that can be compiled independently by LLVM. Thus, in order to customize the copies, we extract each version of code in a separate function, as illustrated in Figure 5.8. Original blocks, BB1 to BB4 are extracted in function *Version_1* and replaced in the original code with a call to this function. In the same manner, the clones BB1_clone - BB4_clone are extracted in function *Version_2* and a call is inserted. We use the function *LLVM::ExtractCodeRegion*, which automatically identifies the values that must be sent as parameters and updates the values used outside of the function. For preserving the SSA form, *LLVM::ExtractCodeRegion* will create multiple copies of the values used outside the function, updated with the results computed in the body of the function. Consequently, it is highly important to replace the uses of the original values in the clones, before extracting each code version in a separate function. Not doing so leads to violation of constraint number three, because copies of the original values would be created.

Mechanism to switch between multiple versions Having created the clones and extracted them in separate functions, we need a mechanism to allow the runtime system to switch between them dynamically. In this respect, for each set of clones and original version, we build a *decision block* consisting in a condition, and branching to function

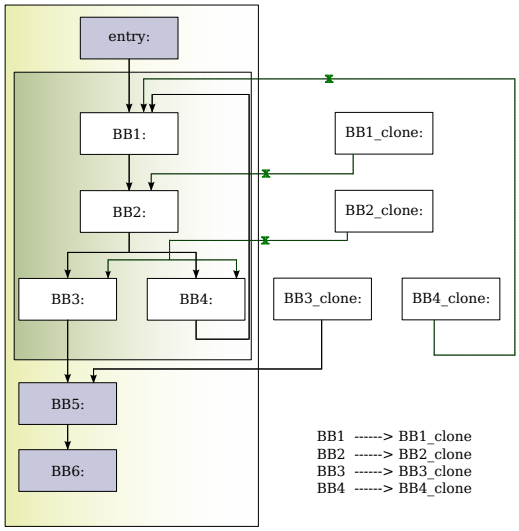


Figure 5.6: Cloning

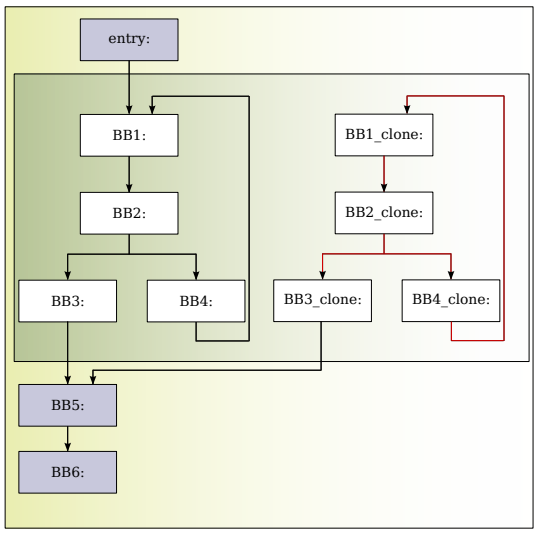


Figure 5.7: Rebuild control flow graph in clones

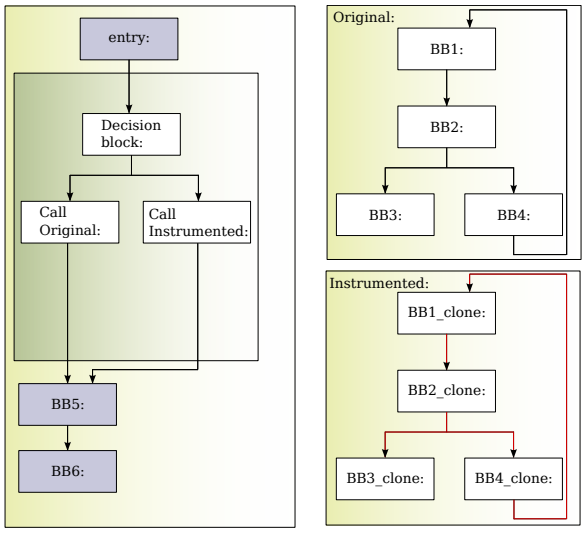


Figure 5.8: Multi-versioning

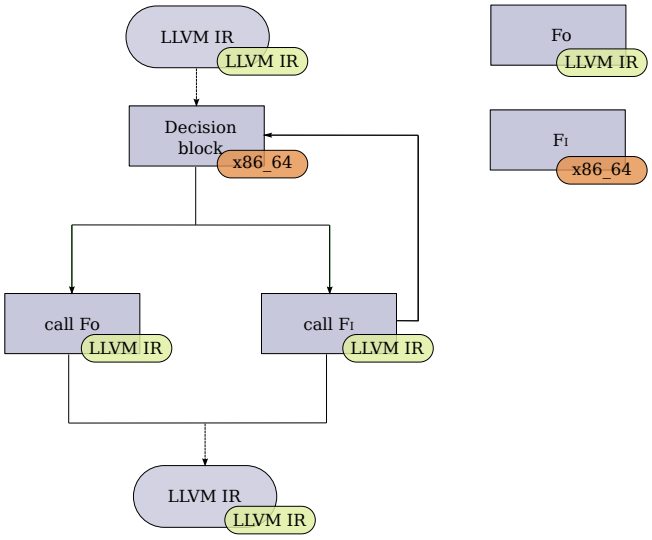


Figure 5.9: Each version is transformed into a suitable representation

calls depending on the result of its evaluation. The condition is evaluated through a callback to the runtime system, which will decide the function to be invoked. Recall that each function represents a different version (figure 5.9).

Extracting versions in separate functions allows us to decide about the most convenient representation. Clones designed for high-level code transformations are represented in LLVM IR, whereas clones targeting low-level information are translated into x86_64. Each clone is compiled, customized and further processed independently. Also, in this manner, we have a clean separation between the regions marked for multi-versioning and the embedding code, from the source level to the LLVM IR and to x86_64 form. The clones represented in the LLVM IR are inlined back in the original code after they have been customized, to reduce the runtime overhead. But this is not possible for the versions in x86_64 assembly representation, due to register allocation. Nevertheless, the overhead incurred by extracting the versions in separate functions is minimal, even when the most time consuming regions of code are marked for multi-versioning.

Handling jumps between LLVM IR and inline assembly

In our proposal, the multiple versions are generated statically and separated in new functions. Nevertheless, we designed a generic framework to manage dynamically generated code and switch between all available versions. In this respect, the switching mechanism requires to insert a call to the function containing the version selected for execution. Furthermore, the runtime system in charge with version selection is enabled to manage the various operations for which different code versions are generated.

As a challenging goal, we focus on loop instrumentation and, more precisely, on interpolating memory addresses accessed inside loop nests. For performance reasons, we tackle loop instrumentation by sampling. Consequently, the runtime system switches between the original and instrumented versions. Furthermore, the runtime system manages various operations required for each type of instrumentation. Chapter 6 presents

```
// backup the stack red zone
// backup the scratch registers
// stack adjustement (x86_64 convention):
mov %rsp,%rbp
mov $0xfffffffffffffff0,%rsi
add %rsi,%rsp
mov $0x0,%rax  // move 0 to %rax (amd x86_64 convention)
// registers for the 'call'; $0x0 will be patched:
mov $0x0,%rdi  // address of the module
mov $0x0,%rsi  // address of the function
// function 'call':
// 1st parameter = rdi (convention)
// 2nd parameter = rsi
callq *%rsi
mov %rbp,%rsp // stack readjustement
// restore scratch registers
// restore stack red zone
```

Figure 5.10: callback in x86_64 assembly code

an application targeting advanced loop instrumentation and illustrating this switching mechanism between versions.

Callbacks. The original code, enclosing the multiple versions and the switching mechanism, is sprinkled with callbacks to the runtime system, positioned at some key-points of the program. To preserve genericness, we developed a modular runtime system, each module consisting in the set of functions required for each operation. The callbacks are inserted as in Figure 5.10.

Saving and restoring the stack red zone and the scratch registers is common to any callback, however, to stay on generic realms, we insert a call to the function located at address 0x0, belonging to the module located at address 0x0. The runtime system will patch the code with the correct addresses of the function and its corresponding module at start-up. For this reason, we insert the callback code in assembly x86_64 representation, inlined in the enclosing code in LLVM IR form. Moreover, the two instructions

```
mov    $0x0,%rdi //address of the module
mov    $0x0,%rsi //address of the function
```

which have to be patched, are inserted in their hexadecimal equivalent form, such that enough space is available to accommodate 64 bits addresses.

Labels and jumps. Constant communication must be ensured between the enclosing code, the multiple versions and the runtime system. In these regards, the runtime system must be able to identify the beginning and the end of each version of code, as well as the address of the code to resume execution when returning from a code version.

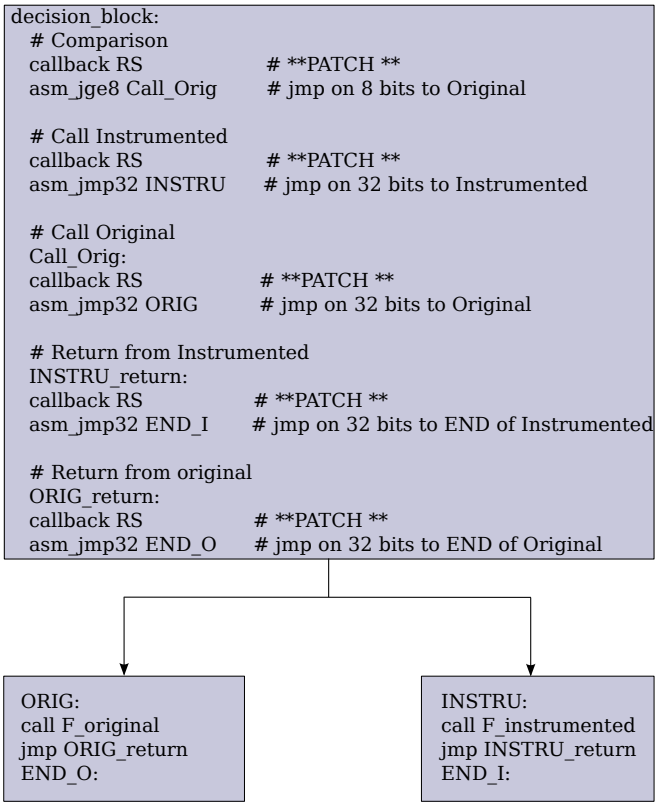


Figure 5.11: Code structure

We mark the key-points of interest by inserting labels as inline assembly code in the above mentioned positions. In Figure 5.11 labels *ORIG* and *END_O* are inserted as x86_64 code, inlined in the LLVM IR code to mark beginning and end of the original version. Similarly, *INSTRU* and *END_I* mark the borders of the second version.

Additionally, the mechanism that allows switching between versions is written entirely in inline x86_64 assembly code. By default, the framework is designed to execute the original version of code. In contrast, when the runtime system is available, instrumentation is enabled. For this, the runtime system patches the branch that points to the original version to point to the switching mechanism. Toggling between versions is achieved by means of a decision block that contains callbacks to the runtime system and jumps to each version of code. Since all callbacks are patched, the code is in x86_64 representation and hexadecimal form, to ensure preciseness. The code structure is depicted in Figure 5.11.

Each callback to the runtime system performed from the decision block requires patches. As a consequence, the size of the inserted code must be fixed. Therefore, each `jmp` is replaced with a jump on a fixed number of bits, either 8 or 32 bits, which will prevent the compiler to generate variable sized `jmp` instructions, as illustrated in table 5.1.

However, in the LLVM IR the names of the labels created by the code generator are not yet available, also the code suffers significant transformations when converting from LLVM IR to x86_64 assembly code. In this respect, each jump inserted as inline

Macro	Hexadecimal form
asm_jge8 TARGET	.byte 0X7D .byte \TARGET \()-.-1
asm_jge32 TARGET	.byte 0X0F, 0X8D .long \TARGET \()-.-4

Table 5.1: Inline assembly code in hexadecimal representation.

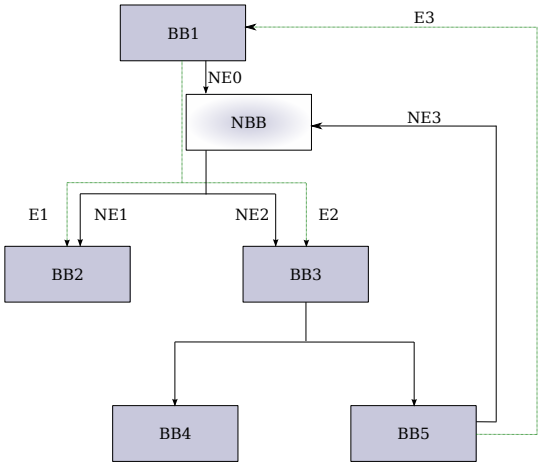


Figure 5.12: Shadowed control flow graph

assembly must be accompanied by a label, inserted in the convenient position in the code. What we obtain is a partial control flow graph managed as x86_64 assembly code, inlined in the LLVM IR.

Handling CFG as inline assembly code. Current compilers do not allow control flow entering or exiting inline assembly code. Furthermore, inline asm is regarded as a constrained string, emptied of semantics, simply printed to the *.s file when machine code is generated. Having no support from traditional compilers, the partial control flow graph in inline assembly code must be handled while preserving the original control flow graph, maintained by the compiler. The labels and jumps inside inline assembly code, partially rewrite the control flow graph, as shown in Figure 5.12.

Blocks BB1 to BB5 and edges E1 to E3 represent the original control flow graph, maintained by the LLVM compiler. By inserting inline assembly code, we shadow part of this graph, by adding a new block, NBB, and replacing a series of edges. A new edge NE0, branching to the new block NBB, is added. Also, edges E1 and E3, originally from BB1 to BB2 and BB3, are replaced with NE1 and NE2, from NBB to BB2 and BB3. Similarly, E3 is replaced by NE3, connecting now BB5 and NBB.

Nevertheless, the inline code is not accessible to the LLVM compiler in this compilation phase, hence, partial rewriting of the control flow graph must be totally transparent. Our approach follows the guidelines below:

1. **Keep original CFG represented in LLVM IR:**
   LLVM does not allow blocks terminating with a non-terminating instruction (terminating instructions are *branch*, *switch*, *return* instructions for example). In

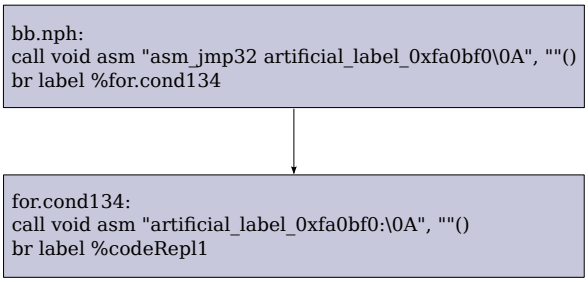


Figure 5.13: Control flow graph rewritten in inline code

consequence, the edges E1, E2 and E3 must be preserved, they cannot be replaced with inline assembly code.

2. **Overwrite branches with jumps in inline code:**
The solution at hand would be to replace edges E1-E3 with new ones, represented in the LLVM IR: NE0-NE3 represented as LLVM IR terminating instruction, rather than inline assembly code. However, as presented in subsection *Labels and jumps*, the runtime system requires a fixed sized code, in order to patch the callbacks. Using LLVM IR terminating instructions, the size of the generated `jmp` instructions may vary.

   Our strategy is to precede the branches with fixed size `jmp` instructions in inline x86_64 assembly code, such as `asm_jge32 TARGET`, previously presented. The targets of the jumps are uniquely generated labels, inserted as first instructions in the target blocks. An example is given in Figure 5.13.

   Hacking the control flow graph is challenging in the LLVM IR, due to the $\phi - nodes$, which must be updated accordingly. For performing multiversioning, one must ensure that all values defined inside the cloned region and used outside, are correctly maintained. As illustrated in Figure 5.14, the values of v are cloned in block BB2_clone, and used outside the region in BB_7. However, the code would not execute correctly if the cloned version is run, since the value v would not be initialized before reaching BB_7. The solution is to introduce a value $v1$ merging the two values v and v_clone, once the region exits. Nevertheless, this could be rather difficult in real-life code examples, with multiple exit points and complex control flow graphs.

   Another aspect concerning $\phi - nodes$ when the CFG is modified by the inline assembly code, is that it might produce invalid code. The LLVM IR imposes that $\phi - nodes$ are the first instructions of a basic block. On the other hand, our strategy requires the inline assembly defining labels to be the first instruction, as shown in Figure 5.15.

   To overcome these problems, we promote the registers to memory, using a dedicated LLVM pass, which eliminates all $\phi - nodes$ and transforms them into pairs of "load" and "store" instructions. The drawback is that increasing the number of memory accesses, leads to a higher runtime overhead.

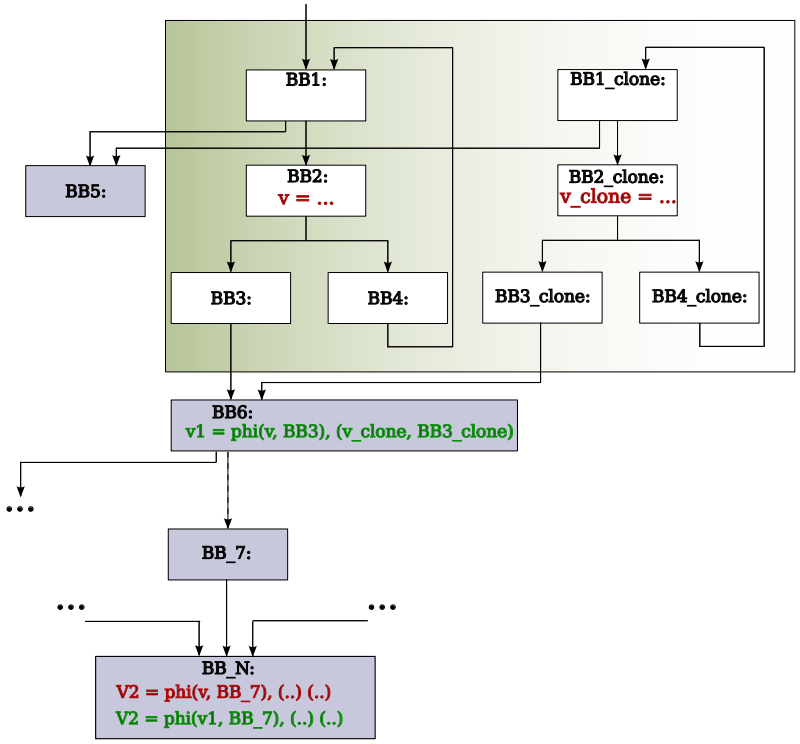


Figure 5.14: Control flow graph with $\phi - nodes$

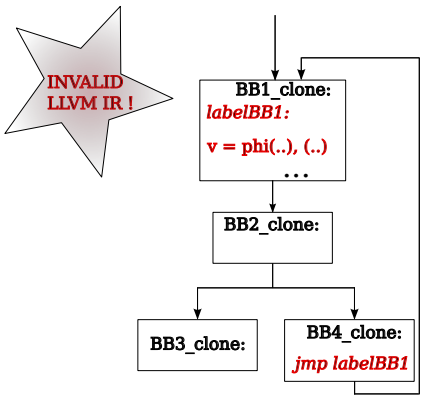


Figure 5.15: Control flow graph with $\phi - nodes$ and labels

3. **Ensure that the overwritten branches are not reachable in the generated code:**
   Not only we duplicate parts of the original control flow graph with instructions represented in x86_64 assembly code, but we also change a series of branches, as illustrated in Figure 5.12. Therefore, special care must be taken to ensure that the newly inserted edges and the ones intended to be rewritten do not interfere. Shortly, we must handle the inline assembly code, such that, in the generated machine code, the edges chosen for elimination are not reachable.

   This may prove to be a challenge, due to optimization passes that perform block fusion, instruction reordering or similar low-level code transformations. A number of optimization passes are executed by default in LLVM; they cannot and should not be disabled. Thus, it is impossible to prevent various optimizations from making multiple copies of the inline assembly code. On one hand, we aim to minimally disturb the optimization process, but, on the other hand, we require the inline code to stay unchanged.

4. **Ensure that inline assembly code is not:**

   - **duplicated**: copying inline code leads to errors due to name conflicts generated by multiple declarations of labels. To avoid it, one must update the original control flow graph such that the optimizers will not attempt to copy the x86_64 code in multiple blocks. The solution we propose is to create a new block for each snippet of inline code. Although in the LLVM IR form, a new branch is added, when converting to machine code, the block is inlined, thus, no additional `jmp` instructions are required.
   - **eliminated**: new basic blocks have to be evaluated as reachable by the LLVM compiler, otherwise they are eliminated as dead code. Namely, there must be at least one branch represented in the LLVM IR pointing to the new blocks. Jumps inserted in inline assembly code targeting labels from new blocks are not accessible, yet not recognized by the compiler. In consequence, one must alter the original control flow graph to include the new block, and, simultaneously, manage the control flow graph expressed in inline assembly code to bypass this branch.
   - **relocated:** instructions reordering has an undesirable effect on our framework, unless influenced from the LLVM IR. It is of high importance to place correctly the artificial labels we insert for marking the beginning and end of each code version. Nevertheless, the code generator reserves its rights regarding the order of the instructions. For instance the LLVM IR code:

     ```
     artificial_label:
     function_call
     ```

     is converted into x86_64 assembly code as:

     ```
     save_machine_state_for_function_call
     artificial_label:
     function_call
     ```

`restore_machine_state_for_function_call`

which causes problems when jumping from the inline code to the `artificial label`. As in the case of code duplication, the solution is to create a new block containing only the inline code. Consequently, the code generator and the optimizers place the `artificial_label` in the correct position.

5. **Minimally influence code behaviour and performance:**
Since our goal is code instrumentation and profiling, we aim to grasp accurate information concerning the behaviour of the code, without degrading the performance. Nevertheless, introducing instrumentation code, as well as the mechanism that allows switching between versions, has an impact on the code generator and optimizers. In this respect, we use metadata information, the least invasive form of tracking code, and we perform multi-versioning in the LLVM IR only after the optimization phases complete. Still, there is a number of optimization passes which are run by default by LLVM before the code generation step. The interaction between these passes and the newly introduced code has an influence on both: the behaviour of the optimizers is affected by the presence of the inserted code, whereas the new code suffers transformations in the optimization process. The compromise we accept is to minimally influence the optimizers such that the code remains unchanged, which is a strict constraint for ensuring the functioning of our framework.

   From the performance standpoint, we tackle instrumentation by sampling, which motivates the need of multi-versioning, and we enable our framework to support manipulation of highly optimized code (-O2, -O3 - the highest optimization levels available in LLVM).

In what follows, we focus on the design and implementation of the code patterns, from the compiler's perspective.

Building the code patterns

Generating parallel code patterns can be a difficult task, as they are very sensitive to the structure of the loop nest. Multiple-exit and multiple-condition loops, rotated or aggressively optimized loops, must be handled by a unique, general algorithm. Additionally, the code patterns require further transformations of the loop nest and of the control flow graph, such as replacing the original loop header with new conditions on the virtual iterators, modifying the backedges of the loops and the exit edges. The loop body is then augmented with initialization and verification code.

Inserting virtual iterators in the loop nest: By inserting new iterators in the loops, one replaces the original control structure with the new conditions on the virtual iterators. Nevertheless, the original conditions are maintained in the code to ensure that no unpredicted iterations are executed. Additionally, since the conditions on the number of iterators are speculative, they require verification. The code structure is displayed in Figure 5.16 and 5.17. Figure 5.16A shows the original CFG of a loop nest of

depth 2; while Figure 5.16B displays the chunked original version of the outermost loop. The condition on the virtual iterator (vi) is evaluated in the block *bb.nph_viCond_orig*. Note also that the backedge of the loop has changed to reach the new header.

The code pattern is displayed in the sequence of images from Figure 5.17C, in which the second loop is prepared for parallelization. Figure 5.17C.1 shows the body of the outermost loop. The innerloop has been extracted into a new function which is called by the block *codeRepl*. Figure 5.17C.2 presents the parallelized code pattern and will be explained in details later. Finally, Figure 5.17C.3 contains the body of *loop*2, which includes, as one can remark, the entire original loop nest. The original conditions of the loops (blocks *bb_nph_par_0* and *for_body6_par_0*) are now part of the transformed innermost loop, and they represent the *guarding code*. Their role is to check the speculation on the loop bounds, i.e. if the original conditions are evaluated to *true* during the execution of the transformed loop, the predicted number of iterations was incorrect and the chunk must be rollbacked. The conditions of both original loops must be evaluated at each innermost iteration, to support transformations such as skewing or loop interchange. This requires a significant restructuring of the nest, since the entire code must become reachable irrelevant of the evaluation of the original conditions. The blocks *for_body6_par_0_uppBoundCond* and *for_body6_par_0_vi_viCond_outermostRollback* initiate the rollback in case of mispredictions on the loop bounds. An example of a parallel code pattern in the LLVM IR is appended in the Annexe.

For the computation of the loop bounds we consider the linear functions obtained from instrumentation and the polyhedral transformation. For the original outermost loop, the original condition is replaced by the chunking condition. With this information we call the FMlib [97] to obtain the correct values of the transformed loop bounds.

Initialization code As presented in the previous section, it suffices to initialize the *basic* scalars originating from ϕ nodes, as the values of all other variables are computed from them. We first collect the list of all ϕ nodes, and next we demote the registers to memory. This will transform the ϕ nodes into pairs of *load* and *store* operations. The following is an example from a 2-depth loop nest:

ϕ = phi(v1,BB1)(v2,BB2)	ϕ = load M1 use ϕ update v store v, M1	store $\alpha \cdot x + \beta \cdot y + \gamma$, M1 ϕ = load M1 use ϕ update v store v, M1

The original ϕ node is instrumented, and we obtain a linear function $a \cdot i + b \cdot j + c$. After the elimination of ϕ nodes, we insert initialization code, before the value of ϕ is loaded from memory. Note that if the loop was parallelized without being transformed, it would suffice to initialize the scalars just once, in the first iteration of each slice

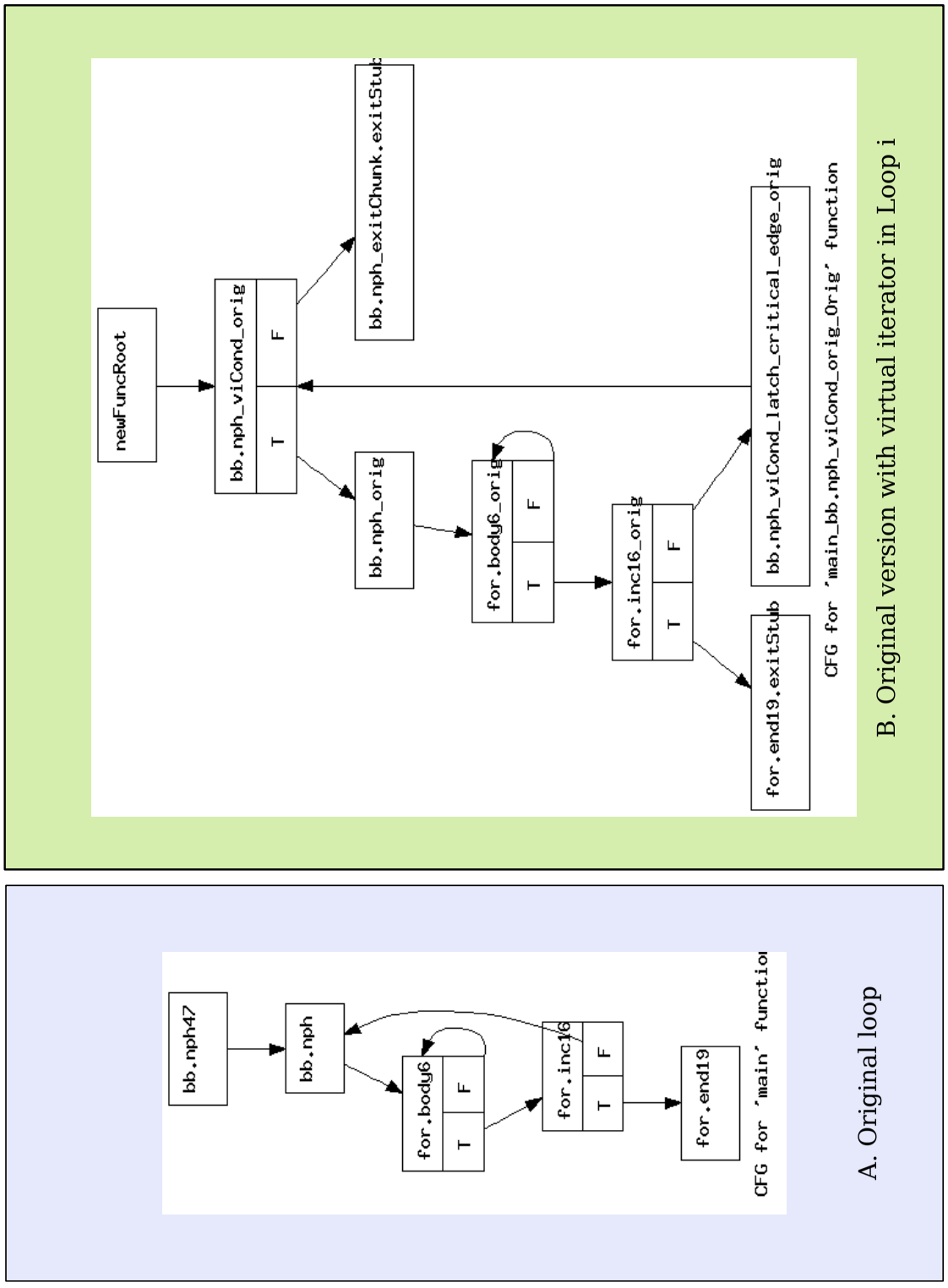
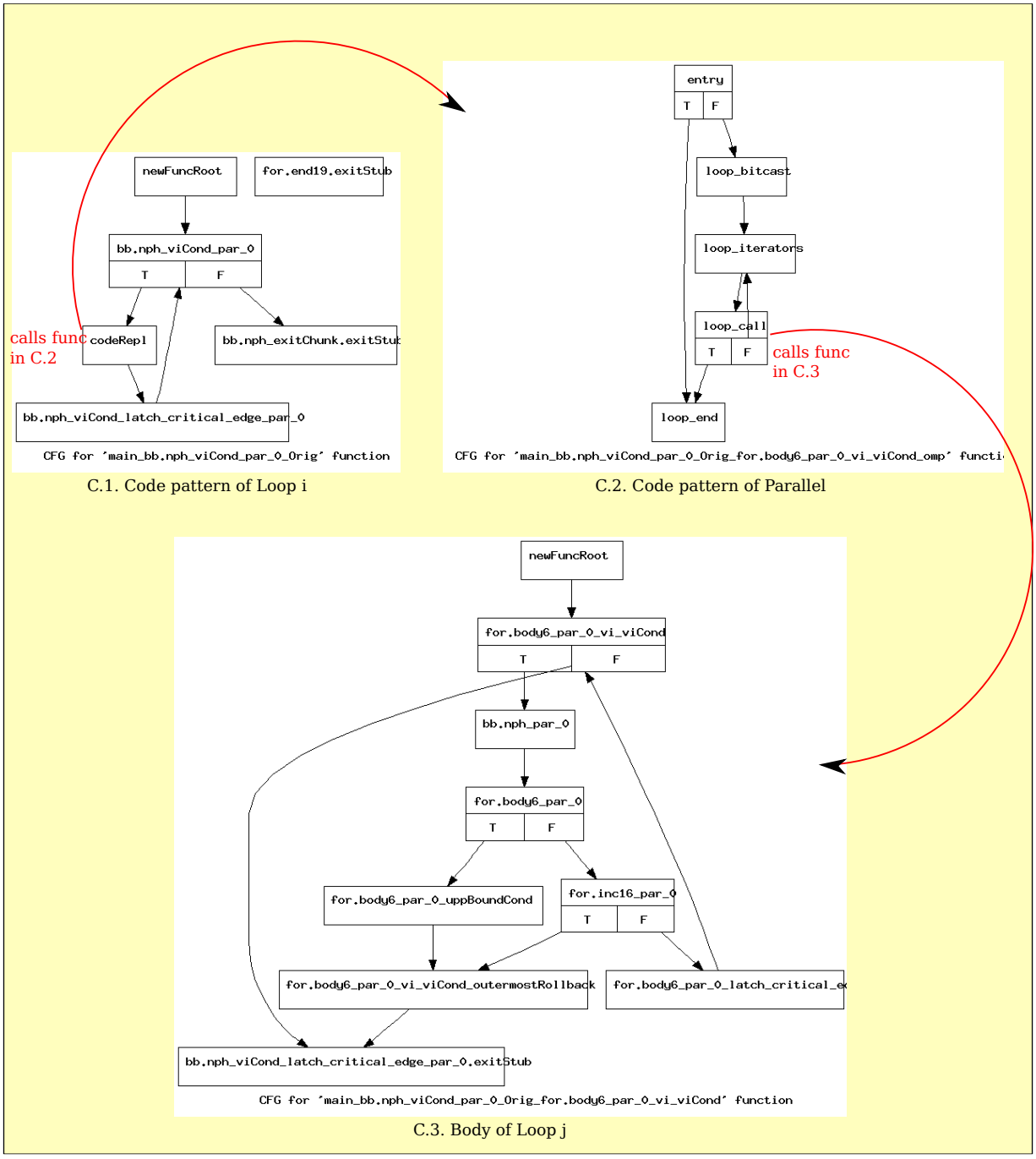


Figure 5.16: Loop structure with virtual iterators



C.1. Code pattern of Loop i

C.2. Code pattern of Parallel

C.3. Body of Loop j

Figure 5.17: Loop structure in the parallel code pattern

allocated to be computed by a thread, because the intra-thread iterations will follow the sequential order. Nevertheless, due to the polyhedral transformations, the initialization code must be executed in the beginning of each iteration, as the execution order of the iterations may change.

The linear function used for initialization is computed by applying the polyhedral transformation T on the linear function obtained from instrumentation:

Given:

$$T \cdot \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} \Leftrightarrow \begin{pmatrix} i \\ j \end{pmatrix} = T^{-1} \cdot \begin{pmatrix} x \\ y \end{pmatrix}$$

For $\mathrm{M} = a \cdot i + b \cdot j + c$ one obtains:

$$\begin{pmatrix} a & b \end{pmatrix} \cdot \begin{pmatrix} i \\ j \end{pmatrix} + c = \begin{pmatrix} a & b \end{pmatrix} \cdot T^{-1} \cdot \begin{pmatrix} x \\ y \end{pmatrix} + c = \begin{pmatrix} \alpha & \beta \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} + \gamma.$$

We insert the initializing instruction `store` α`*x+`β`*y+`γ`, M1` , using the transformed virtual iterators x and y. The coefficients of the linear functions α, β and γ are declared as external global variables, and their values are computed and assigned during the execution, by the runtime system.

Privatization All values that are used in the body of the loop and are defined outside its body, are transmitted as parameters to the function in which the loop is extracted. In this manner, we can easily communicate the updated information from one chunk to another, but also, this is a safe method to identify the values that are transmitted to all threads. To guarantee a data race free code, we privatize the values which should not be shared. In our approach, we privatize all scalars, by creating a local copy in the body of the loop. For the iterators of the loops in the nest, we keep the original values (sent as parameters of the function), the basic scalars are initialized with the interpolating linear functions, whereas the remaining of the scalars are computed from the basic scalars.

Finally, to transmit the values of the private copies from the code patterns to the other versions (original or instrumented), one would require to know which thread executed last, and simulate the `lastprivate` option of the `pragma omp parallel for` directive. However, inserting checks in the patterns to identify the last executing thread would have a considerable impact on performance. Therefore, our strategy is to reuse the interpolating linear functions, after the execution of the parallel chunk finishes. The strategy is to initialize these scalars before launching a sequential version (original or instrumented), with the linear functions, by considering the starting values of the iterators in the current chunk. In this manner, we do not copy the values of the private scalars in the last executing thread, but we directly initialize the original scalars with their values, computed by the linear functions. This strategy is guaranteed to be correct by the verification code, which signals a misprediction, in case the interpolating linear functions cannot predict the correct values, as explained in what follows. Note that these initializations performed *before* launching a chunk are required only for chunks embedding sequential code, because the parallel versions already include the initialization code.

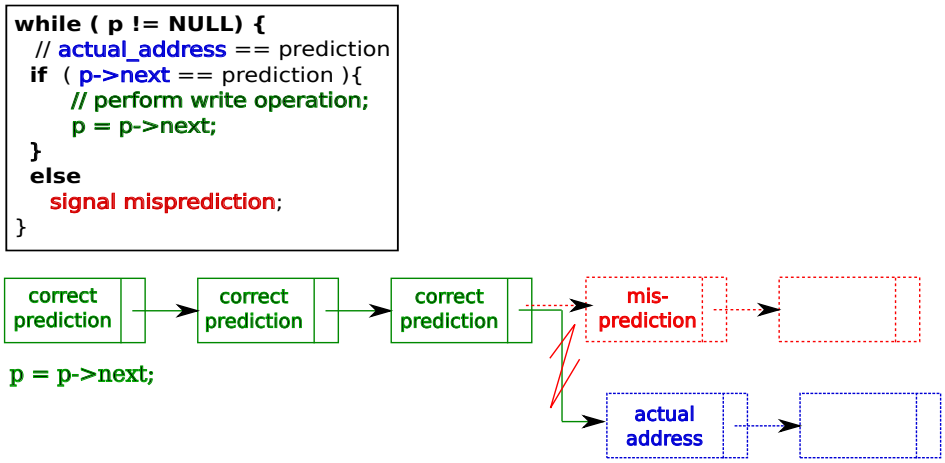


Figure 5.18: Verify *write* accesses

Verification code Not only the memory state must be initiated to ensure that each thread receives the correct initial values, but also the speculations must be verified during the execution of the code. Following the guidelines presented in the previous section regarding the verification code, our system verifies that: i) only predicted (and saved) memory locations can be modified; ii) each iteration computes the expected values for the next iteration considering the sequential order; iii) the speculated loop trip counts are correct.

Hence, the rollback system is based on memory backups performed before launching a parallel chunk. The memory space which should be updated is predicted using the interpolating linear functions. Upon a misprediction, the memory is restored from the backup and a new chunk processing again the restored data is launched. Notice that computing the range of memory addresses touched by a parallel chunk has a negligible cost, since it is equivalent to computing the values of the linear functions at the loop and chunk bounds.

To ensure that no memory location outside the predicted and saved area is modified, verification code is inserted guarding all *store* instructions. Thus, before being allowed to write into memory, the target address of the *write* instruction is compared to the predicted one, and writing is permitted only if they match as in Figure 5.18. The predicted value is computed with the linear function on the transformed iterators (x, y).

The verification code for the next sequential iteration considers the same instructions as the ones requiring initialization, i.e. all scalars originating from ϕ nodes. Once the new value is computed a *store* instruction writes it into memory. Hence, it suffices to verify that this is the value expected for the next sequential iteration, in order to ensure the correctness of the prediction. For the verification code, we compute the values of the original iterators

$$\begin{pmatrix} i \\ j \end{pmatrix} = T^{-1} \cdot \begin{pmatrix} x \\ y \end{pmatrix}$$

corresponding to the current iterators (x, y). Next, based on the prediction on the

```
//parse the old iterators
for (iter = 0 ; iter < depth ; iter++){
   // express them as functions of the new iterators
   for (newit = 0 ; newit < depth+1; newit++){
       funct_iter = compute the linear function for the iterator iter
       funct_uppB = compute the linear function for its upper bound
   }
   lfit[iter] = funct_iter;
   lfup[iter] = funct_uppB;
}
```

Figure 5.19: Original iterators & upper bounds wrt new iterators

loop bounds, the values of the iterators for the corresponding next sequential iteration are computed with:

$$\begin{pmatrix} i_{next} \\ j_{next} \end{pmatrix} = \begin{pmatrix} i + (j == upp_j) \\ (j+1) * (j \neq upp_j) \end{pmatrix}$$

which makes use of the fact that the virtual iterators in the not-transformed code start from 0 and are incremented with a step of 1 in each iteration. i_{next}, j_{next} denote the original iterators in the next iteration considering the sequential order, and upp_j represents the upper bound of j, obtained from instrumentation. The prediction is computed with the linear function $a \cdot i_{next} + b \cdot j_{next} +$ c obtained from profiling.

As a general algorithm for loops of any depth, the steps are:

1. *Express the original virtual iterators with respect to the transformed ones:* computes the values of the original iterators and of their interpolated loop bounds, with respect to the transformed iterators, as in Figure 5.19.

2. *Compute the value of the original virtual iterators in the next iteration, according to the sequential order:* computes the value of the original iterators in the next sequential iteration, by comparing it to the predicted upper bound. The pseudo-code is displayed in Figure 5.20.

3. *Verify the predictions:* For each *store* instruction originating from a ϕ node, we insert the verification code which checks if the computed value and the predicted one coincide. Should a mismatch be detected, the code will signal a misprediction, Figure 5.21.

Finally, verification of the loop bounds relies on the *guarding* code to ensure that no unpredicted iterations are executed. Thus, the original loop bounds cannot be exceeded, since the original conditions are preserved in the code, guaranteeing that no additional iterations are performed. On the other hand, as soon as the exiting conditions become true, the verification codes checks if this is the last sequential iteration, according to the prediction. If the prediction is correct, the execution continues, such that all iterations are executed, following the parallel schedule (see Section 3.4). Otherwise, the misspeculation is detected.

In order to signal a misprediction we make use of a set of flags. Each thread owns a flag which can be set by any thread that detects a misspeculation through one of its verification instructions. Next, each thread polls this flag at every iteration of the

```
//parse the old iterators
for (i = 0; i < depth ; i++){
   // if i is the innermost loop
   if (i==depth-1 && depth>1){
       compute value of the next sequential iterator
       nfit[i] = (lfup[i] != lfit[i]) * (lfit[i] + 1);
   }
   // if i is a middle loop
   if (i != 0 && i<depth-1){
       compute value of the next sequential iterator
       nfit[i] = ((lfit[i+1] == lfup[i+1]) + lfit[i])
               * (lfit[i] != lfup[i]);
   }
   // i is the outermost loop
   if (i == 0){
       compute value of the next sequential iterator
       nfit[i] = (lfit[i+1] == lfup[i+1]) + lfit[i];
   }
}
where:
depth = depth of the loop nest (starting from 1)
lfit[i] = value of the i^th original iterator
lfup[i] = value of the upper bound of i^th original iterator
nfit[i] = value of the i^th original iterator in the next sequential
iteration
```

Figure 5.20: Computation of the next sequential iterations

```
// init code
store α*x+β*y+γ, M1
φ = load M1
use φ
update v
store v, M1
compute the linear function predicting the value that should be stored
pred_val = Σ_{i=0}^{depth-1} coef[i] * nfit[i] + cst
// verif code
if (pred_val ≠ v) set flag

where:
depth = depth of the loop nest (starting from 1)
coef[i] = coefficients of the interpolating linear function
nfit[i] = value of the i^th original iterator in the next sequential
iteration
```

Figure 5.21: Verification code

parallel loop and stops whenever it has been set. In short, once a thread detects a misprediction, it sets the flags of all other threads and stops its execution. The other threads poll their own flags and stop in case it is set. Using one flag per thread limits the synchronization required for accessing the flag. Since its value is checked at the beginning of each iteration, it would be inefficient to synchronize across all threads. By assigning one flag per thread, synchronization is required only when a misprediction is encountered and all flags must be set. However, this event appears infrequently.

One could argue regarding the frequency of polling the threads to limit the overhead. It can be performed every n iterations, in each iteration of the parallel loop, or in the outermost loop only. This can be adjusted according to code's characteristics.

OpenMP calls Once the patterns are generated to support various polyhedral transformations, the code for managing the creation and execution of threads must be prepared. Since the current version of the LLVM front-end, *Clang*, does not support multi-threading, we implemented this task by inserting calls to the GOMP library [54]. To generate multi-threaded code, we took the following steps:

1. Extract the parallel loop into a new function using *llvm::ExtractLoop(...)* (see Figure 5.17C.3).

2. Generate the *void* omp_data* argument for the thread function

3. Generate the thread function (see Figure 5.17C.2).

4. Call *GOMP_parallel_loop_runtime_start* and *GOMP_parallel_end* functions

Note that we use the schedule *runtime*, to be able to adjust it easily from the runtime system, during the execution.

The code corresponding to the thread function depicted in Figure 5.17C.2 (*main_bb.nph_viCond_par_0_Orig_for.body6_par_0_vi_viCond_omp*) is shown in Figure 5.22. Before executing its slice of iterations, each thread polls its flag and computes the values of the starting and ending iterations it has to execute. Provided that the flag allows the execution and there are still iterations to be computed, the thread updates the values of the current iterators and of the loop bounds and transmits them to the function which contains the loop. Hence, each call to the loop function (*main_bb.nph_viCond_par_0_Orig_for.body6_par_0_vi_viCond*) computes the slice of iterations allocated for the calling thread.

By extracting the parallel loop in a new function, one can decouple the support for multi-threading and the code of the loop. Communication is achieved by updating the structure of arguments sent to the loop function. Nevertheless, the code of the loop must be transformed to be compliant with the GOMP standards. Thus, multiple exit loops must be converted to a unique exit loop, since the thread function cannot return values different than void. Using the flags, the runtime system can distinguish whether the chunk completed its execution correctly or if a misprediction was detected.

```
define void @main_bb.nph_viCond_par_0_Orig_for.body6_par_0_vi_viCond_omp(omp_data)
{
entry:
currTh = call i32 @omp_get_num_threads()
poll thread's flag
checkFlags(flags[currTh])
compute the slice executed by the thread
GOMP_loop_runtime_next(omp_start, omp_end)
...

loop_bitcast:  ; preds = entry
bitcast the void* omp_data to a structure
...

loop_end:  ; preds = loop_call, entry
call void @GOMP_loop_end_nowait()
ret void

loop_iterators:  ; preds = loop_call, loop_bitcast
initialize the bounds computed by each thread
thread_iter = thread_iter + omp_start
thread_uppBound = omp_end
...

loop_call:  ; preds = loop_iterators
call the thread function to execute the slice
call void @main_bb.nph_viCond_par_0_Orig_for.body6_par_0_vi_viCond(omp_data_struct)
GOMP_loop_runtime_next(omp_start, omp_end)
if the chunk did not finish
go to loop_iterators
else go to loop_end
}
```

Figure 5.22: Generating OMP code in LLVM IR

Static dependence analysis to propose polyhedral transformations

In addition to the multiple versions generated statically (original, instrumented and code patterns), the compiler prepares a set of polyhedral transformations to be tested and applied dynamically. A possible approach is to extract the information statically available and to perform a partial dependence analysis, using for example the ISL library [127]. Next, the compiler can generate a large set of transformations, which are verified against the results of the dependence analysis, as a preliminary test. Transformations declared as invalid according to the statically detected dependences are discarded, while the others are encoded as matrices in the binary file. They represent good candidates to be verified dynamically by the runtime system, when the complete information regarding inter-iteration dependences is available.

The underlying reason for generating such transformations statically is to limit the overhead. First, this strategy eliminates the overhead of building polyhedral transformations dynamically, and second, it is a guided approach which removes from consideration a subset of the invalid transformations.

The current implementation focuses on building the framework for a speculative parallelization approach based on patterns, which represent masks for various polyhedral transformations. Thus, it only proposes a naive method, by randomly generating a large set of the most common transformations, from which the invalid ones are discarded. However, the algorithm for building the initial set of transformations can be replaced by more complex ones. One could suggest an offline profiling, aimed to capture more information and to further guide the process for generating more suitable transformations. Moreover, an interesting strategy enters the realm of machine learning [122], where transformations could be proposed based on the codes' characteristics. Such heuristic methods would improve the quality of the transformations being proposed and of the system in general, since the generated parallel code would perform better and the number of rollbacks could be significantly reduced. All these methods are subject to further analyses and introspections.

Moreover, our system is designed with great care for flexibility and support for further extensions. Thus, in case none of the statically prepared transformations is eligible, it allows generating new transformation at runtime, based on the history of previous invocations. We consider this a powerful feature, which trains the system in proposing increasingly better transformations and generating highly performing parallel code. We have envisaged the importance of maintaining a history of previous transformations, rollback points and dynamic dependences and we designed the system to support these extensions with minimal implementation efforts.

5.1.4 Inserting static information: headers and parameters

Finally, as means of transferring information from the compiler to the runtime system, we append to the binary file a list of *headers* and *parameters*. The headers inform the runtime system regarding the actions to be taken, such as: perform instrumentation on loops, gather memory locations and perform interpolation; run a dependence analysis, and decide which code version to be executed. These headers only specify for what type of analyses and transformations the code has been statically manipulated.

```
# number of headers.
.global vmad_headers_nb
 vmad_headers_nb:
        .long   1
```

```
#list of headers
. global vmad_headers
 vmad_headers:
. global vmad_0_entry_lb
 vmad_0_entry_lb:
        .quad   vmad_0_entry
        .quad   vmad_loop_entry
        .quad   vmad_0_param
```

```
#list of parameters
. global vmad_0_param
 vmad_0_param:
        .long   3           # loop depth
        .quad   vmad_0_end_original
        .quad   vmad_0_loop_reinstru
        .quad   vmad_0_instru_call
```

Figure 5.23: List of headers and parameters

Nevertheless, the runtime system has full responsibility in deciding the order of actions and in orchestrating the execution. More details are given in the next section 5.2.2.

Each header has associated a list of parameters, which give specific information for the particular action, such as the structure of the loop nest, depth, parent loops, addresses in the code that require patching etc. The coefficients of the linear functions that are inserted in the code patterns for initialization and verification code are unknown at compile time. They are declared as external global variables and are embedded in the parameters sent to the runtime system, such that they can be correctly identified and assigned values. Likewise, the proposed transformation matrices are included in the list of parameters.

Hence the list of headers and parameters represents the bridge between the static and the dynamic components of the framework, transmitting information statically available which would be impossible or prohibitively expensive to be computed at runtime. Additionally, they allow the set-up of the runtime system in accordance with the statically tailored code.

As illustrated in Figure 5.23, the list of headers is introduced by its length, given by *vmad_headers_nb*. The list of headers is specific to the type of instrumentation or optimization, as they determine the modules to be loaded in the runtime system (*vmad_0_entry*, *vmad_loop_entry*). Headers are linked to the corresponding parameters (*vmad_0_param*), containing higher level information statically available, but which would be time-expensive to identify in the binary representation, such as the loop depth. Furthermore, the compiler transmits as parameters instrumentation specific information, for instance the addresses of the code snippets inserted in the original code (*vmad_0_loop_reinstru*, *vmad_0_instru_call*).

The list of parameters may have a varying length and includes information such as labels indicating sections of code which are patched by the runtime system, or specific details, for instance characteristics of a loop, its depth or parent loop. The parameters may differ depending on the instrumentation type.

5.2 Dynamic component

The development of the runtime system was started by Matthieu Herrmann for his master thesis in 2010 [57, 62].

The runtime system has a modular implementation, which eases the process of extending VMAD to support various types of code analyses and transformations. For a

particular type of instrumentation or optimization, several modules might be required. Data is transmitted from the program to the runtime system using a dedicated header per module, stored in the binary file as an array of three pointers (see Figure 5.23). Each module is built independently to carry out a specific action, thus the modules are decoupled but they can exchange information by means of a set of parameters. In short, the runtime system can be seen as a collection of modules that are dynamically loaded when necessary. Each module must implement five main entry points: *init* to instantiate the module, *quit* to kill such an instance, *on*, *off* and *reset* for enabling, disabling and resetting the module and its data. Additional operations are provided by a module for its specific purposes. They are invoked using callbacks in the application code, patched initially by *init* in order to point to the corresponding instance of the module and to the corresponding operation.

The static-dynamic collaborative framework is depicted in Figure 5.24. Once the code has been statically prepared, the runtime system uses LD_PRELOAD to load VMAD's dynamic shared library at startup. It provides its own version of the C-library entry point *libc_start_main*. This allows VMAD to initially read the relevant information from the headers, to load the required modules and to patch the binary file.

In case the VMAD's dynamic shared library is not loaded, the program executes normally, without any instrumentation or dynamic optimization. In its *libc_start_main*, VMAD performs the following operations:

1. It reads the headers from the program's binary file. There is a one-to-one mapping between the list of headers in the binary file and the modules loaded at startup. Additionally, the parameters of the headers are read, such that each module is instantiated with the correct information, before the execution begins. At startup, our system relies on the dynamic linker in order to find the predefined symbols generated at compile time, giving access to the static information, which is used to load the appropriate dynamic modules and to instantiate them. For each header:
   - VMAD gets the type of instrumentation it has to manage;
   - it obtains the necessary parameters;
   - it loads the module corresponding to the type of instrumentation;
   - it updates the address of the module stored by the runtime system.
2. since by default, the program is designed to run without the runtime system, VMAD patches the code in order to branch to the instrumented version;
3. the *init* function of each module is invoked to patch all callbacks in the code to point to the corresponding functions of the module;
4. VMAD calls the original *libc_start_main*.

The main roles of the runtime system are to orchestrate the execution flow and to process the data dynamically acquired after instrumentation to be used in the optimization phase. For this purpose, there are two types of information transmitted to the runtime system:

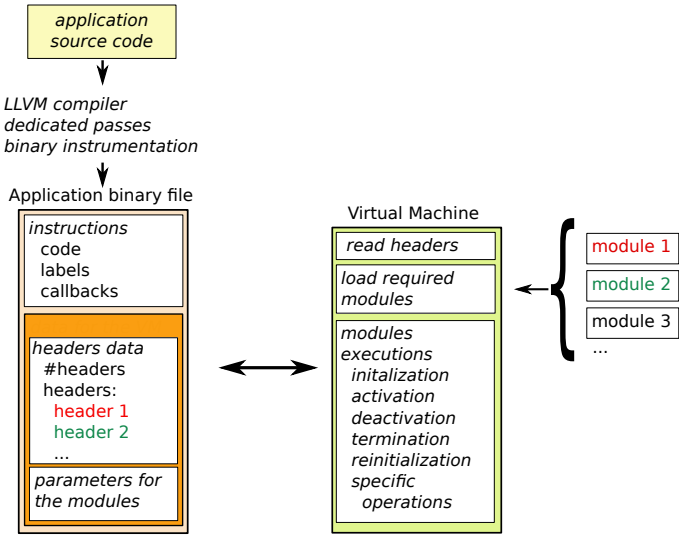


Figure 5.24: Static-dynamic collaboration

1. *Static data* is compiled in the binary code which can be used:
   - by the runtime system itself: for example, to know which modules are required (headers);
   - by the loaded modules: to indicate which addresses should be patched (parameters);
   - or directly by the inlined instrumentation code: to allocate memory for saving the registers in use (parameters).
2. *Dynamic data* represent the information captured during the execution and is transmitted to the modules to be analyzed.

5.2.1 Code manipulation

Recall that the selection mechanism consists of preceding each set of versions by a *decision block*, which invokes the runtime system to decide upon the version to be executed. Additional callbacks to the runtime system (Figure 5.10) are performed during execution to transmit the data collected via instrumentation, as presented in Section 5.1.2. They are placed statically at the beginning and end of each instrumented version. Following a generic approach, all callbacks have a standard form, using indirect calls. This approach enables the compiler to generate multiple generic versions, and relies on the runtime system to patch the address of the corresponding function, at runtime.

Actions The modules' functions that must be called from the instrumentation inserted in the code are defined as a list of *actions*. They offer support to perform multiple function calls with only one generic call inserted in the instrumented code. Moreover, they provide an easy way to change which functions are called at runtime. An action contains:

- a pointer to the name of a function;
- the address of the function, initialized by the runtime system using the dynamic linker;
- the address of the header on which the action applies;
- a generic pointer that can be used for passing extra parameters to the function.

The runtime system uses the information sent in the list of headers and parameters to find and initialize all actions at startup.

Control flow orchestration In addition to the callbacks, the runtime system must patch the inline assembly code to point to the instrumented version, before starting the execution. The code intended to be patched must follow a fixed design. For example, there exist 8- or 32-bits displacement jumps. Jumps that must be patched always have a displacement of 32 bits. When patching, the runtime system writes binary instructions in the code area. Instructions are made of operator codes and operands; operands can be changed dynamically when needed, by other callbacks if some conditions are met.

Dynamic data processing

Since the type of data captured during the instrumentation phase is specific to the particular analysis, so is the set of operations initiated by the runtime system in order to process this information.

5.2.2 Runtime code orchestration for speculative parallelization

For performing speculative parallelization of a loop nest using the strategy we described, we wrote the following modules:

1. vmad_loop
2. vmad_memory_address
3. vmad_interpolation
4. vmad_dependence_analysis
5. vmad_decision
6. vmad_code_generator
7. vmad_FM

The first module, **vmad_loop** is dedicated to handling loops. It expects a list of parameters with the addresses of the callbacks that require patching, in the beginning and ending of the loop, and of each of its iterations. Based on the information encoded in the binary file, the module will patch the callbacks and prepare the code for the specified type of analysis / transformation. Additionally, **vmad_loop** is responsible for maintaining information specific to each loop and for building a map of the structure of the loop nest, available at runtime to the other modules.

For profiling purposes, the next two modules, namely **vmad_memory_address** and **vmad_interpolation**, are invoked. During the execution of one iteration of a profiled version of a loop, all values that require interpolation are stored in a buffer. From there, they are identified by means of a dirty flag and consumed by the **vmad_memory_address** module. Its role is to interface with the memory area of the program, to acquire the data and to transmit it to the **vmad_interpolation** module for further processing. As its name suggests, this module will build the interpolating linear functions on the virtual indices of the enclosing loops (when possible). For a loop nest of the form

```
#pragma speculative_parallelization {

  loop 1
    loop 2
     ...
}
```

the module builds a linear function during the execution of the subloop, depending on the virtual iterator of loop 2 (vi_2): $\beta \cdot vi_2 + \gamma$. As soon as loop 2 finishes its execution, these coefficients are propagated to the parent loop, and the corresponding **vmad_interpolation** module computes the coefficient for the iterator vi_1 in the interpolating linear function $\alpha \cdot vi_1 + \beta \cdot vi_2 + \gamma$. Once the function is computed, the module uses any further information it retrieves to verify the correctness of the interpolating function.

During the profiling phase, the acquired memory addresses are used to perform a light-weight dependence analysis. The algorithm is described in Section 3.2.1. For this purpose, the **vmad_dependence_analysis** module builds a table storing for each instruction accessing memory: the ID of the instruction, read/write type, the values of the indices of the enclosing loops and the accessed memory address. This table is dynamically updated and it serves the purpose of computing distance vectors between instructions. Thus, for each two instructions (dynamic instances) that access the same memory location and at least one is a write, a distance vector is built. Next, the table is updated, such that it only stores information for the instructions performing the last relevant accesses for each memory address, see Section 3.2.1. When instrumentation ends, the module considers pairs of instructions (i_1, i_2), at least one being a write:

- if there exist a distance vector, that remained constant during the profiling phase, the distance vector is included in a set S and the pair is discarded.

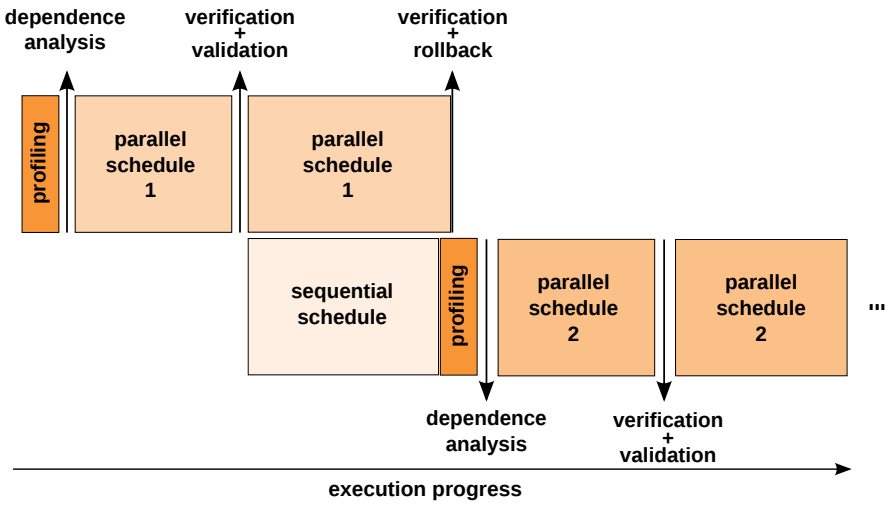


Figure 5.25: Code orchestration

- if no distance vector has been computed for (i_1, i_2), the *value range analysis* and the *gcd* tests are run to conclude whether the instructions could access the same memory location. To complete the tests, the module uses the interpolating linear functions of each instruction, computed by the **vmad_interpolation** module. For the *value range analysis*, the lower and the upper bounds of the chunk embedding the parallel version are considered. Since the chunk size can be adjusted dynamically, we impose a maximal size and we use this value for the test.

- if no conclusive result can be given by any of the previous tests, a conservative decision is taken and a dependence is introduced between the two instructions, at the level of the outermost loop.

Finally, the role of the **vmad_dependence_analysis** module is to select a suitable transformation, according to the dependence information it computed. We remind that given a transformation T and the set S of distance vectors, the test is to compute the product $r_s = T \cdot d_s$, for all distance vectors d_s in S. If the first non-null component is positive for all r_s, the transformation is valid according to the dependences discovered during the profiling phase. The parallel loop level is the first level that does not carry any dependence r_s, as noted in Section 3.2.1. The module receives a list of transformations T, statically proposed for the loop nest and verifies them for validity until a suitable one is found. Our current implementation simply returns the first found valid schedule, nevertheless we have envisaged future improvements for the selection algorithm, discussed in Section 3.2.1.

For the orchestration of the whole execution, the *vmad_decision* module is invoked at loop entry by the decision block. Its role is to select the version to be executed by the next chunk and to adjust the chunk size. It implements the following algorithm, illustrated by an execution example in Figure 5.25:

- the first chunk of the loop runs an instrumented version;

- after an instrumented version, based on the result of the dependence analysis, either an original sequential or a parallel chunk is launched;

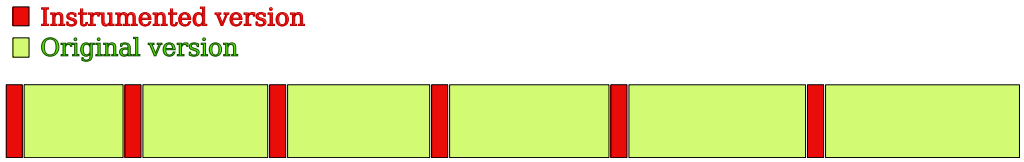


Figure 5.26: Chunk size increase for the original sequential version

- after an original sequential chunk, another instrumented one is launched, in the hope of discovering parallelization opportunities;
- after a parallel chunk, if its execution was validated, another parallel chunk is launched, with the same schedule; otherwise, if a rollback was detected, the same iterations are re-executed by an original sequential one;
- the process repeats until the end of the loop.

Also, for each chunk, the size is adjusted correspondingly. Namely, instrumented chunks have a fixed, small size. In case no parallelization opportunities can be discovered and the instrumented and original chunks alternate, the size of the original chunks increases, with a fixed step, as in Figure 5.26.

Similarly, if a parallel version is validated, the size of the next chunk is increased with a fixed size, such that, very large parallel chunks can be launched, if the selected schedule proved to be suitable. In short, for a stable behavior of the loop, the chunk size is increased gradually, until large sizes, while for unstable behaviors, smaller chunks are launched, to limit the overhead of a potential rollback. We carried out tests with chunks of fixed sizes, increasing sizes and decreasing sizes, but there was not a significant impact on performance. To ensure that the correct state of the memory can be restored after a faulty chunk, before launching a parallel chunk, the **vmad_decision** module builds a safe-copy of the memory area predicted to be modified. For this purpose, it computes the extreme values of the predicting linear functions on the given chunks and uses *memcpy*, which is highly optimized for such operations. More details are given in Section 3.4.2. Similarly, after a faulty chunk, the memory state is restored from the back-up in the same manner.

With the ten loop nests from the PolyBench suite, we measured the time overhead of executing a `memcpy` before launching a chunk, in order to backup the data that will be updated (Figure 5.27). It is quite low for most of the codes, even if it can already be observed that the impact of the copy decreases while the chunk size increases. The unfavorable cases are represented by the codes `dynprog` and `gemver`. For `dynprog`, the amount of saved data is not proportional to the chunk sizes. Hence, even with small chunks, an important amount of data has to be saved. And since more chunks have to be launched to complete the whole execution, the impact of executing a `memcpy` at each chunk is obviously quite important. For such codes, a linear transformation can be helpful in getting proportional amounts of saved data per chunks. For `gemver`, the amount of saved data is important and increases significantly with the chunk size.

For these measurements, we performed one memory copy for each instruction writing into memory. However this technique can be further improved, such that only one

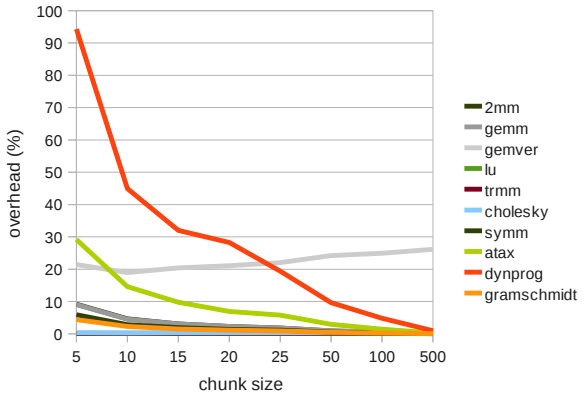


Figure 5.27: `memcpy` time overhead

`memcpy` call is required for intersecting memory ranges, as shown in Figure 5.28A, saving M1 and M2. Moreover, it might be optimal to perform a memcpy of a large memory area even for disjoint memory ranges, in case the memory zone between these areas is small enough. Empirical tests show that it is worthwhile performing a large `memcpy`, as in Figure 5.28B, as soon as the unnecessarily saved area is equal to or less than the total memory range to be copied. Nevertheless, note that `memcpy` is highly optimized and it could exhibit different behaviours, depending on the data to be copied. As a consequence, the threshold for performing either just one call to copy a large array, or several calls with smaller array, should be adjusted depending on the data for optimal results. We carried out tests on arrays of 100MB of the form:

|——|xx|——|xx|——|xx|——|xx|...|

where a memory range is represented by : |——|, and the spacing between the ranges by |xx|. The results are displayed in Figure 5.29, where each curve corresponds to a different spacing, expressed in bytes, between the ranges. We vary the size of chunks and measure the time taken to transfer them using several `memcpy` calls. The black continuous line is the reference and it represents the time required to copy the entire array (chunks and spacing) in one `memcpy`. To guarantee that no fault occur due to accesses to unallocated memory areas, we have implemented a handler for the `SIGSEGV` signal, and if the memory backup fails, it calls our custom handler. It will then perform a `longjump` to get out of the `memcpy` call. In this manner, we can return in the original context and continue the execution with a sequential chunk, if the predicted memory area cannot be saved. This situation appears upon a misprediction, but also when we approach the end of the loop and attempt to save a memory range that overflows the allocated memory. Recall that the memory backup is required only for write memory accesses performed inside the loop. For protecting the patterns against incorrect read accesses, we use a similar strategy, of protecting the execution of the parallel code by our own `SIGSEGV` handler, such that the execution can safely resume if an incorrect read memory access is performed.

The **vmad_decision** module is the glue binding together the results of the analysis modules and calling the suitable transformation modules if necessary. Additionally, it

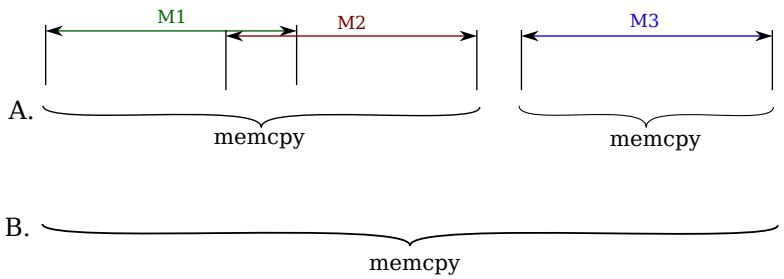


Figure 5.28: `memcpy` calls

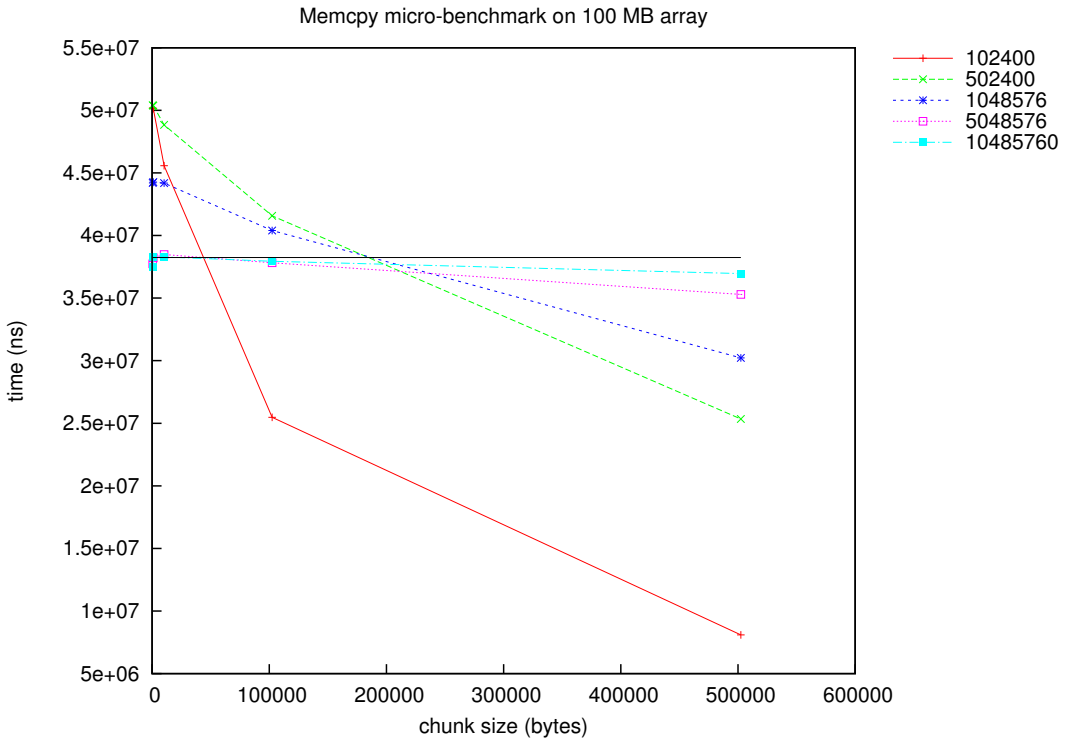


Figure 5.29: `memcpy` calls with varying chunk sizes (courtesy: J-F. Dollinger)

represents the core mechanism for switching between different versions of the loop and launching chunks of corresponding sizes and types. In what follows, we present the modules responsible for applying transformations on the loops dynamically, using the code patterns statically prepared.

If the **vmad_dependence_analysis** module finds a valid schedule, it is the role of the **vmad_code_generator** module to detect the suitable code pattern and to prepare it for execution. The selection of patterns refers to retrieving the parallel loop level from the **vmad_dependence_analysis** module, and finding the pattern in which the corresponding loop is prepared for parallelization. Next, the coefficients of the linear functions used for initialization and verification are assigned values. Since they are declared as external global variables, it suffices to assign them the correct values before launching a parallel version. Thus, the code generation is simple and fast, as shown by the experiments we present in Chapter 6, Section 6.2. The computation of their values depends on the utility of the linear function. Namely, linear functions used for code initialization will be computed as a scalar product between the linear function obtained from instrumentation and the transformation matrix. The iterators used for the initialization code are the current transformed iterators. Similarly for the verification code guarding the write memory accesses, to ensure that no write operations are performed beyond the saved memory area. On the other hand, for verifying the speculations on the dependences between iterations, the values of the original iterators in the next iteration are computed, according to the sequential order. Hence, for these linear functions the runtime system applies simply the values of the coefficients resulting from profiling. The global variables corresponding to the coefficients and the type of linear function that must be computed are transmitted by the compiler in the list of parameters associated to the **vmad_code_generator** module. In short, in our proposal, parallel code generator consists of computing the coefficients of the linear functions using dynamically available data and assigning the values to the corresponding global variables.

Finally, once the code is prepared for a parallel execution, it is the **vmad_FM** module that computes the correct loop bounds, considering the new schedule. To achieve this, the FMlib library is used. First, **vmad_FM** builds the system of inequalities and invokes FMlib to obtain the solution. The module provides two functions, computing the lower and the upper bound for each loop. They take as parameter the depth of the loop - corresponding to the index in the solution - and the vector with values of the current iterators of the parent loops. Hence, to avoid multiple min-max bounds, the computation of the loop bounds is replaced with a function call invoking the functions of the **vmad_FM** module. For the lower bounds, **vmad_FM** computes a max on all positives of the FM solution, for the corresponding index. Reversely, for the upper bounds the minimum value computed from the negatives of the FM solution is returned.

Time overhead of the Fourier-Motzkin elimination

We measured the time required to compute the loop bounds resulting from a linear transformation of several loop nests, using the FM library. The computed equation systems were composed of 5 to 9 variables and of 5 to 12 equations. The measured time was always quite stable and stayed between 0.02 and 0.03 second.

5.3 Conclusions

To conclude, implementing such advanced code transformations in the intermediate representation of the LLVM compiler shows both advantages, as well as several challenges. LLVM is a modular framework, which eases the interaction with the internals of the compiler. On the other hand, it is still a research dedicated and rather young compiler which does not yet provide full functionalities, such as various advanced loop analyses and optimizations or support for OpenMP. Additionally, frequent releases and changes of API makes it a tedious task to always migrate to the latest release. Nevertheless, LLVM is a developer friendly platform with an active community and which stays as the foundation of many open research projects.

In general, to facilitate the implementation of a TLS system that relies on multiple versions prepared statically, the compilers should allow a more flexible control flow graph, containing control instructions defined by the developer. This would simplify the runtime scenario of launching different versions successively, as one could prepare the switching mechanism statically, based on some control instructions defined at runtime, avoiding unnecessary conditionals.

Compilers should be able to parse and understand the semantics of the inline assembly code, such that they do not alter them during the optimization phases. Currently, it is a real challenge to combine control flow instructions defined in the intermediate representation of the compiler and in inline assembly code, since one cannot perform jumps from one to another. GCC allows one to jump from an inline assembly code to a label defined in C, but not vice-versa. In contrast, LLVM does not embed such functionalities.

Additionally, it would be helpful to have compilers preserving a mapping of the code transformations and optimizations that have been applied, such that the original unoptimized code can be recovered for various further manipulations.

Finally, to perform dynamic analysis and optimizations, the code must be prepared to communicate with a runtime system. In this respect, the compiler could provide an interface to ease such a communication, by allowing incompletely defined function calls and control flow graphs, annotated with some information that transfers the responsibility on the runtime system. Since this does not exist, we were forced to use indirect function calls and inline assembly, which either introduce an overhead, or are extremely error prone and difficult to handle and debug.

Since we have entirely developed the runtime system, we had more flexibility in the design choices as the only constraints were imposed by the x86_64 architecture. The challenges we faced in building the runtime system were related to maintaining a minimal runtime overhead.

Chapter 6

Evaluation of the VMAD framework

Runtime code optimization becomes the main strategy for facing the ever extending and changing variety of existing processor architectures and execution environments that an application can meet. Unlike static compilers, which take conservative decisions from limited information available in the source code, runtime optimizers can rely on more information extracted at execution time. However, extracting this information induces inevitably some time overhead that has to be minimized and broadly hidden by the optimization gain.

Extracting runtime information is typically performed via code instrumentation, namely additional instructions aimed to collect relevant values such as accessed memory locations, variable values or taken branches. In order to minimize the time overhead, a classical strategy is to perform instrumentation by sampling. Consequently, a dynamic mechanism must provide the means to alternate quickly between instrumented and non-instrumented code, without altering the code semantics. The first section of this chapter illustrates our approach for performing code instrumentation by sampling with the purpose of building interpolating linear functions for each monitored value. We carried out experiments aimed to validate our approach of instrumenting by sampling and to identify loop nest candidates for speculative parallelization that fit the requirements of our system.

Section 6.2 presents the evaluation of the entire VMAD framework, when speculatively parallelizing loops. We dedicate this section to assess the overhead of different components of the framework: instrumentation, dependence analysis, runtime code generation from the parallel code patterns; and to evaluate the speed-up of the generated code, by comparing it to the speed-up obtained by manually parallelizing a code using OpenMP directives, when possible. We conducted experiments on two different architectures, varying the number of threads and various parameters that tune the framework.

6.1 Code instrumentation

This application presents a static-dynamic collaborative approach dedicated to instrumentation of nested loops. The goal of our instrumentation is to collect the memory addresses that are accessed during samples of the executed iterations and, if possible, to compute a linear function interpolating their values. Additionally, the loop trip counts of each loop, except the outermost, are also collected for a few runs to capture their linearity.

As explained in Chapter 5, Section 5.1.2, we extended the LLVM compiler to process the marked regions and to statically create a set of instrumented and non-instrumented versions of the loops. For a minimal time overhead, we use a sampling strategy where we instrument the first iterations of each loop of the considered nest. At runtime, the executions of instrumented and non-instrumented versions alternate, thanks to the runtime system which guides the execution flow. Based on the acquired information, the runtime system computes interpolating linear functions on the enclosing loop indices, for all instructions accessing memory and for the loop trip counts.

6.1.1 Related work

Many tools designed for program instrumentation have been proposed, among which some of the most well-known are PIN [79], Valgrind [87], DynamoRIO [18] or Dyninst [21]. But they are not adapted for a runtime optimization system, due to their high time overhead.

Pin [79] is a software system that performs runtime binary instrumentation, as presented in Chapter 1, Section 1.2. It enables the user to build a wide variety of program analysis tools, known as pintools. However, dynamic instrumentation, such as the interpolation of memory accesses in loops presented in this thesis, would be impossible to be implemented efficiently with Pin, as it is tailored to instrument the code for the whole execution time, leading to large overheads.

More recently, in [74], the authors propose the toolkit PEBIL for efficient x86 binary instrumentation. However, the proposed instrumentation approach is not well-suited to advanced instrumentation of loop nests. PEBIL performs function relocation to acquire enough space to insert branch instructions dynamically, to the instrumentation code. Hence, write operations are required at runtime to instrument the code or to set it back to its original state. A classical technique to avoid an increased number of write operations, is duplicating the code and branching to different versions, such as profiled, original or optimized versions [4,27]. Thus, efficient loop nest instrumentation can be achieved by profiling only a subset of the iterations for each loop level of a nest, and switching during the execution between the instrumented and non-instrumented versions. However, with PEBIL, a profiling strategy based on sampling would induce a lot of memory writes to toggle between instrumentation code and NOP instructions. PEBIL is similar to VMAD since it does not use software dynamic translation (SDT), but static binary instrumentation, yet the tool uses a different strategy for transferring the control from the application to the instrumentation code.

Multi-versioning and sampling have been widely adopted for some particular dynamic analyses [4,27,56,83]. Arnold and Ryder [4] were the first to propose an efficient

framework for instrumentation by sampling, their work being extended by Chilimbi and Hirzel in [27], Hauswirth and Chilimbi [56], and Marino *et al.* [83]. Gao et al. [52] propose to acquire dynamic memory traces with low overhead using their tool MetaSim. Similar to our strategy, they include sampling and process the memory addresses as soon as they are collected. Also, they instrument code efficiently, by saving only the required registers. Nevertheless, MetaSim induces a 5 fold slowdown, which makes it impractical for dynamic instrumentation.

VMAD has been built by taking great care of its performance and its runtime overhead. Hence we avoided to use software dynamic translation that would have delayed the run of the input program, contrary to what is done in Pin [79] and in other similar tools. Further, instrumentation instructions are not inserted on-the-fly by replacing some NOP instructions that have been previously inserted at compile time, as it is done with PEBIL [74]. Rather, several copies of the same code extracts are built at compile time, each copy corresponding to a phase in the whole program run in which the profiling processes will either operate fully, partially or will be completely disabled. The price to pay with this approach is the larger size of the program binary file. However, great care can also be taken to minimize the size of the copies by inserting branches to the original code whenever possible. Besides performance, another noticeable benefit with this approach is the opportunity to create any advanced instrumentation for which the related instrumented copy can be significantly different than the original code.

VMAD is, to our knowledge, the first proposal providing low-level instrumentation initiated from the source code.

6.1.2 Instrumentation by sampling in VMAD

The static-dynamic collaborative framework dedicated to code profiling is depicted in Figures 6.1 and 6.2. At compile time, we manipulate the higher level C/C++ source code and translate it into the LLVM intermediate representation with additional specific metadata. Further processing leads to a multi-versioned code, containing original and instrumented versions. Each loop is extracted into a function prior to inserting the instrumentation code. Both original and instrumented versions are compiled into a binary code, thus ending the statical phase of the process (Figure 6.1). Dynamically, the runtime system selects which version to run, as shown in Figure 6.2.

Since we focus on loops, the first iterations are employed to compute whether the memory accesses follow the pattern of an affine function of the loop iterators. This information can then be used to predict the memory accesses in the subsequent iterations. In this respect, each instruction accessing a memory location is tracked in order to study the linearity of its associated function.

Loop nest instrumentation

In order to handle the exact trip counts of the considered loops, that can be either *while*-loops or *for*-loops with more than one exit point, we reuse the notion of "virtual iterators", introduced in Chapter 3, Section 3.2. Recall that these iterators are initialized with 0 and incremented with the unit stride, and are used to keep track of the actual number of executed iterations.

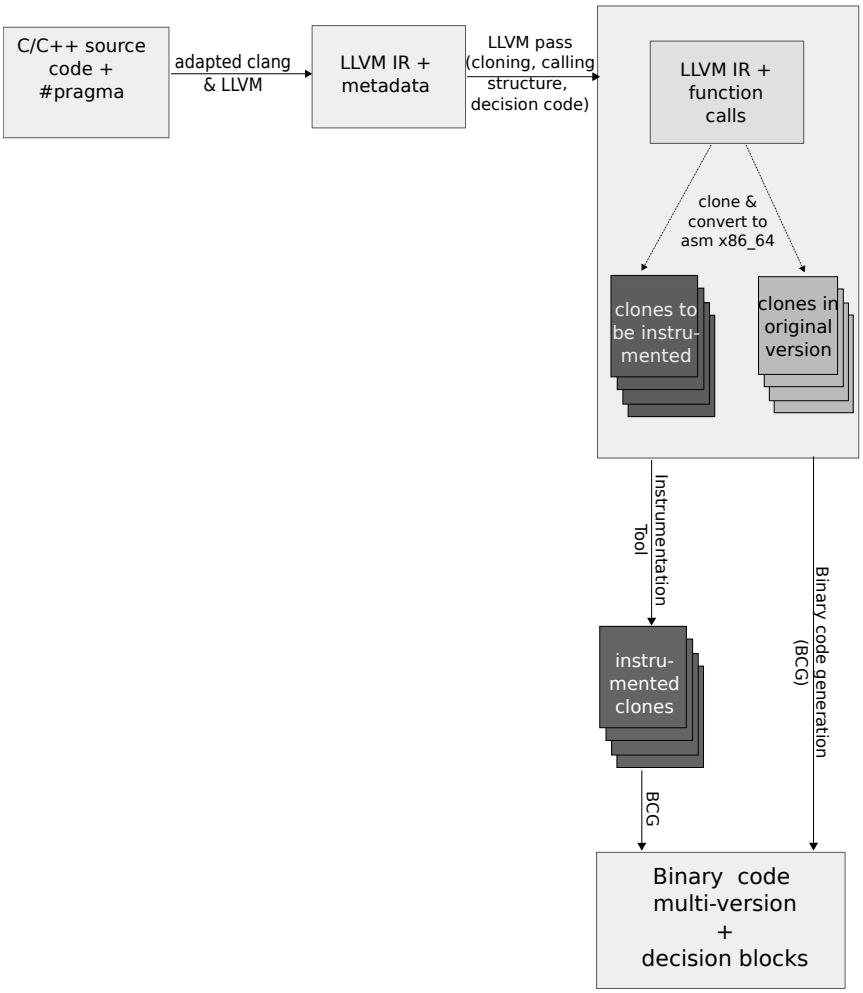


Figure 6.1: Static component of the framework

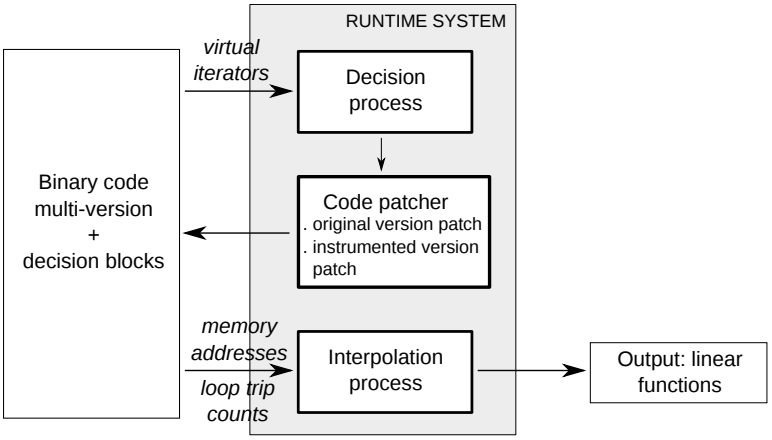


Figure 6.2: Runtime component of the profiling framework

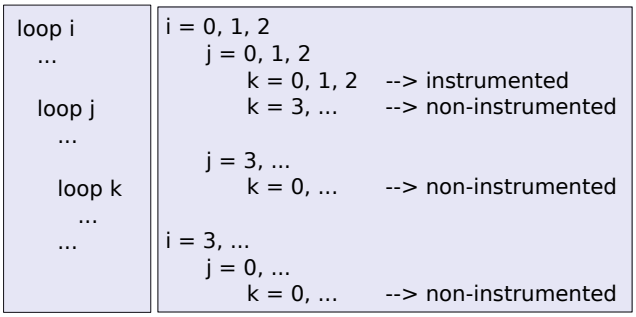


Figure 6.3: Loop nest instrumentation

The complexity of the method is outlined in the case of nested loops, as instrumentation depends not only on the iteration of the current loop, but also of the parent loops. For a throughout understanding, consider the loop nest in Figure 6.3. The first three iterations of each loop are instrumented. One may easily notice that instrumented and non-instrumented iterations alternate, hence the execution has to switch from one code version to another at runtime.

Figure 6.4 illustrates the structure of the code and the links between different versions. Blocks O_i, O_j and O_k represent the original version of the code, while I_i, I_j and I_k represent the instrumented bodies of each loop. The instrumented and original versions are connected together at their entry point, where a choice is made at runtime deciding which version to run, based on the values of the virtual iterators. One decision block is associated to each loop, represented by D_i, D_j and D_k, correspondingly. More precisely, if $i = 1$, the value of iterator i allows instrumentation, therefore its body will be instrumented, block I_i. But if $j = 3$, the body of the second loop will be non-instrumented, O_j as well as the body of its subloop, O_k. The same strategy is applied to the other iterations as well. As a general rule, if a parent loop is non-instrumented, all its subloops will be non-instrumented. On the other hand, if the parent loop is instrumented, its subloops may or may not be instrumented, depending on the value of their own iterators.

For linear interpolation of memory accesses, each memory instruction should be profiled during the execution of at least three iterations, in order to get sufficient address values. However, since some memory instructions can be guarded by conditional branches, it is required to profile such instructions for more iterations, to increase the chances of collecting enough, *i.e.*, at least three, address values. This contributes to the accuracy of the computed interpolating functions. In our experiments, we fixed the number of instrumented iterations to 10, which was a good trade-off between overhead and accuracy. The sampling rate can be set by a parameter. The first two collected values are dedicated to computing the affine function coefficients, while the remaining values are used to verify the correctness of the interpolation.

6.1.3 Instrumenting memory accesses

Our instrumentation tracks the memory accesses inside loops and to each instruction accessing memory, it associates an affine function on the loop iterators, interpolating

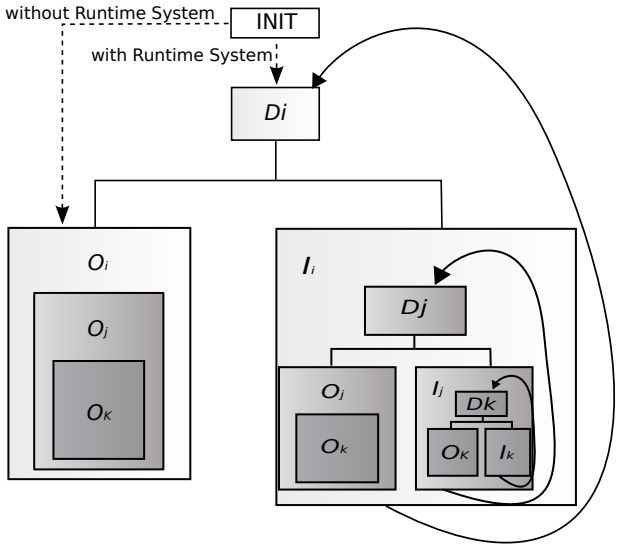


Figure 6.4: Code structure

the accessed addresses, when possible. Similarly for the loop trip counts.

Since the number of accessed locations can be very high, considering a memory intensive loop nest, it is recommended that each accessed memory address is processed immediately by the interpolation process, rather than stored for a later utilization. In the frame of our purpose, each memory access is considered either to compute or to validate the function as soon as it has been registered.

For each instrumented loop, a buffer is created at compile time, to save the state of the machine before the interpolation process and to store the accessed memory addresses. Communication with the runtime system is achieved by means of a dirty flag, which indicates that a new memory location is available in the buffer.

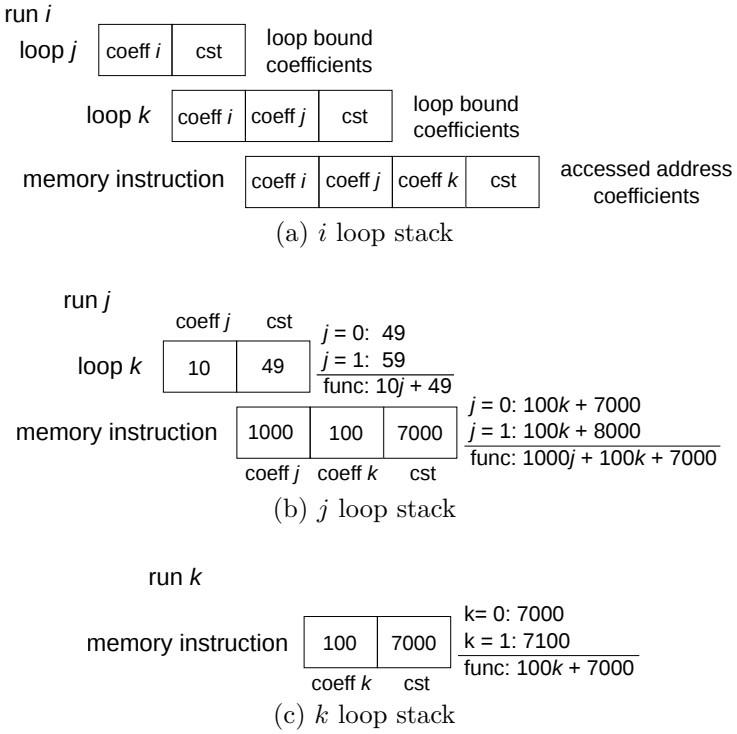
As the instrumented iterations of a loop are executed, the runtime system reads the values of the memory locations from the designated buffer and computes the function coefficients. Subsequent instrumented iterations are used to verify the linearity of these functions.

To achieve this aim, for each instrumented loop nest, the runtime system creates a stack and pushes a structure to accommodate the loop nest [64]. Each instruction accessing memory requires space to store the coefficients of the interpolating function and the depth of the embedding loop. Analogously, the same is required for the loop bounds.

A simplified version of the process is described in Figure 6.5, where only one instruction is considered to access memory, as part of the innermost loop.

When execution reaches the i loop, the structure pushed on the stack contains space for (Figure 6.5a):

- coefficients of the function interpolating on the j loop upper bound,
- coefficients of the function interpolating on the k loop upper bound,



(a) i loop stack

(b) j loop stack

(c) k loop stack

Figure 6.5: Stack structure of each loop level

- coefficients of the function interpolating on the memory location accessed by the instruction contained in the innermost k loop; the four coefficients correspond to the i, j and k indices of the enclosing loops, plus a constant.

Execution continues with the j loop, and similarly a structure for the body of j is pushed on the stack, containing space for two coefficients to interpolate on the upper bound of the k loop, and three coefficients for the function tracking the accessed memory address, depending on indices j and k, plus a constant (Figure 6.5b).

Execution of the k loop triggers a push of a structure containing only two coefficients for the function of the accessed memory location, as is depends only on k, plus a constant (Figure 6.5c). As the instrumented iterations of the k loop are executed, the corresponding coefficients are computed and stored in the associated positions. As soon as the execution of the k loop ends, its structure (Figure 6.5c) is popped and the values of the coefficients as well as the total number of iterations of the inner loop are propagated in the structure of the parent loop.

As a new iteration of the parent j loop begins, we push a new structure for the k loop on the stack and compute the new memory access function.

In the same manner, the process is repeated until all loops finish their execution. At this time, all coefficients have been computed and the functions verified for linearity. This structure can be easily extended to handle multiple instructions which access memory inside the body of a loop nest.

Table 6.1: Measurements made on some of the C programs of the SPEC CPU 2006 (first part) and Pointer-Intensive (second part) benchmark suites.

Program	Runtime overhead (-O0)	Runtime overhead (-O3)	code size increase	# linear m.a.	# instrumented m.a.	Percentage of linear m.a.
bzip2	0.24%	12.31%	218%	608	1,053	57.74%
mcf	20.76%	17.23%	213%	2,848,598	4,054,863	70.25%
milc	0.081%	3.61%	44%	1,988,256,000	1,988,256,195	99.99%
hmmer	0.062%	0.76%	63%	845	0	0%
sjeng	182%	11.13%	80%	1,032,148,267	1,155,459,440	89.32%
libquantum	3.88%	2.76%	21%	203,078	203,581	99.75%
h264ref	0.49%	4.59%	0.44%	30,707,102	32,452,013	94.62%
lbm	0%	0.93%	170%	358	0	0%
sphinx3	172%	27.62%	20%	51,566,707	78,473,958	65.71%
anagram	-5.37%	34.88%	73%	134	159	84.27%
bc	183%	36.79%	11%	243,785	302,034	80.71%
ft	-8.46%	176%	86%	22	36	61.11%
ks	29.7%	2.98%	268%	29,524	42,298	69.79%

6.1.4 Results

For our experiments, we targeted the C codes from the SPEC CPU 2006 benchmark suite [112] and four codes from the Pointer-Intensive benchmarks [94]. We instrumented the loop nests in the most time consuming functions [131]. The benchmarks were run using the `ref` input files to compute VMAD's runtime overhead, and using the `test` input files to get output files with the interpolation results, since runs using the `ref` files produce an amount of data too large to be stored on the disk, but suitable for online consuming. We have carried out the experiments using the -O0 and the -O3 optimization levels. The execution platform is a 3.4 Ghz AMD Phenom II X4 965 micro-processor with 4GB of RAM running Linux 2.6.32. We ran each program in its original form and in its instrumented form to compute the runtime overhead introduced by VMAD. For each instrumented loop nest, the dynamic profiling is activated each time its enclosing function is invoked, for the experiments using -O0 optimization level. In the experiments with a higher optimization level (-O3) we instrument the first eight calls of each function.

Our measurements are shown in Table 6.1. The columns show for each program: the program name (first part of the table: SPEC CPU 2006, second part: Pointer-Intensive); VMAD's runtime overhead, both with -O0 and with -O3; the code size increase; the number of instructions performing *linear* memory accesses ; the number of instrumented memory instructions; the percentage of memory accesses that were identified to be linear.

For most programs, VMAD induces a very low runtime overhead, which is even negligible for `bzip2`, `milc`, `hmmer`, `h264ref` and `lbm`. For the programs `sjeng` and `sphinx3`, the significant overheads are mainly due to the fact that the instrumented loops execute only a few iterations, but they are enclosed by functions that are called many times (with O0). Thus, all iterations are run while being fully instrumented.

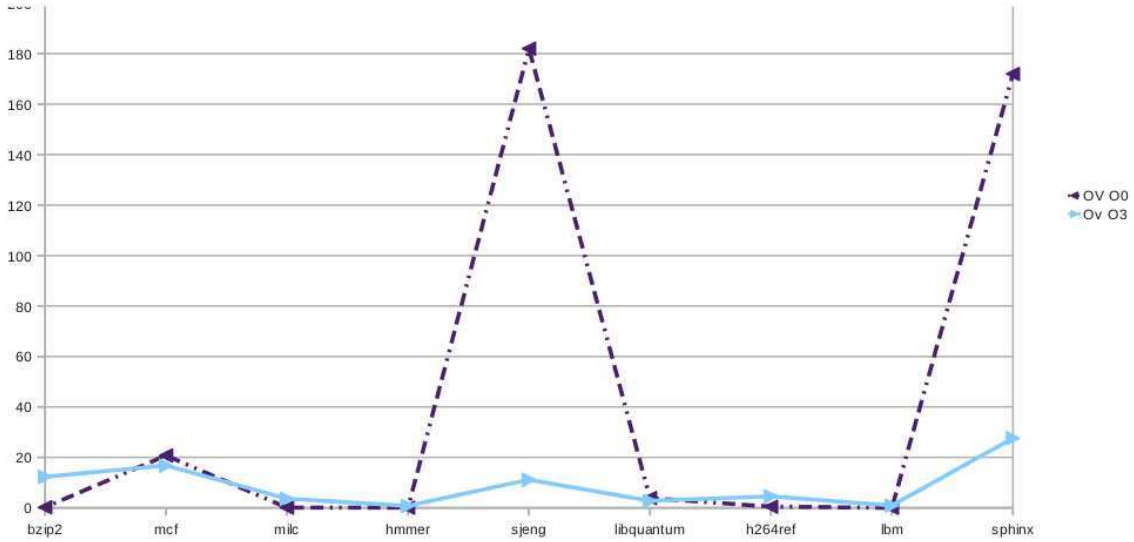


Figure 6.6: Runtime overhead with O0 and O3 optimization levels

However, the profiling strategy is improved in order to manage such cases by deactivating the instrumentation after a few calls (with O3). Program `milc` shows an opposite behavior since a few memory instructions are executed many times. In such a case the runtime overhead is very low. For the Pointer-Intensive benchmarks, the execution times are too small – of the order of milliseconds – to get relevant overhead measurements: either a large runtime overhead is obtained since VMAD inevitably induces a fixed minimum overhead (`bc`), or even a speedup is obtained (`ft`), which may be explained by cache locality, new alignments or new optimization opportunities. In some situations, the overhead is higher when instrumenting optimized code (-O3), since modifying optimized code impacts its performance, still, the execution time is better than with O0. The graphical representation of the overhead obtained with O0 and with O3 optimization levels is given in Figure 6.6. The X-axis lists the instrumented programs from the SPEC CPU 2006, while the Y-axis represents the overhead in percentage. The values are taken from Table 6.1. The dotted line illustrates the overhead with O0 (second column) and the continuous line, the overhead with O3, respectively (third column).

We also noticed that this particular instrumentation process increases the size of a program's binary file by 400 bytes per instrumented memory instruction, on average. However, the code size variation strongly depends on the depth of the loop nests and on the percentage of code selected for instrumentation, compared to the total size. For instance, the program `milc` instrumented version is 267 kbytes versus 185 kbytes for the original version.

In Figure 6.7 it is shown an extract from the program `ks` (Pointer-Intensive benchmark suite), and some interpolation functions that were computed by our profiling process. For instance, one of the memory accesses can be modeled as $64i + 24881848$ where i denotes the virtual outer loop index.

```
# pragma instrument_mem_add{
   for (mrA = groupA.head, mrPrevA = NULL;{
                  ....
      for (mrB = groupB.head, mrPrevB = NULL;
           mrB != NULL;
            mrPrevB = mrB, mrB = (*mrB).next) {
                   ...
    gp = D[(*mrA).module] + D[(*mrB).module]
                                  - CAiBj(mrA, mrB);
                  ....
                  if (gp > gpMax) {
                     gpMax = gp;
                     maxA = mrA; maxPrevA = mrPrevA;
                     maxB = mrB; maxPrevB = mrPrevB;
                  }
```

```
  0 |  0 | 140736985202616 |
  0 | 64 | 24881880 |
  0 | 140736985202632 |
 64 | 0 | 24881848 |
  4 | 0 | 6345856 |
  4 | 6345896 |
  0 | 0 | 140736985202244 |
  0 | 0 | 4234836 |
  0 | 0 | 140736985202244 |
  0 | 0 | 140736985202572 |
  0 | 0 | 140736985202568 |
  0 | 64 | 24881872 |
  0 | 0 | 140736985202616 |
```

Figure 6.7: Code extract from `ks` and its corresponding interpolation functions

To identify whether the analyzed loops are good candidates for speculative parallelization using polyhedral transformations, we monitored closely the code examples from the SPEC CPU 2006, during their whole execution. Figure 6.8 displays our findings:

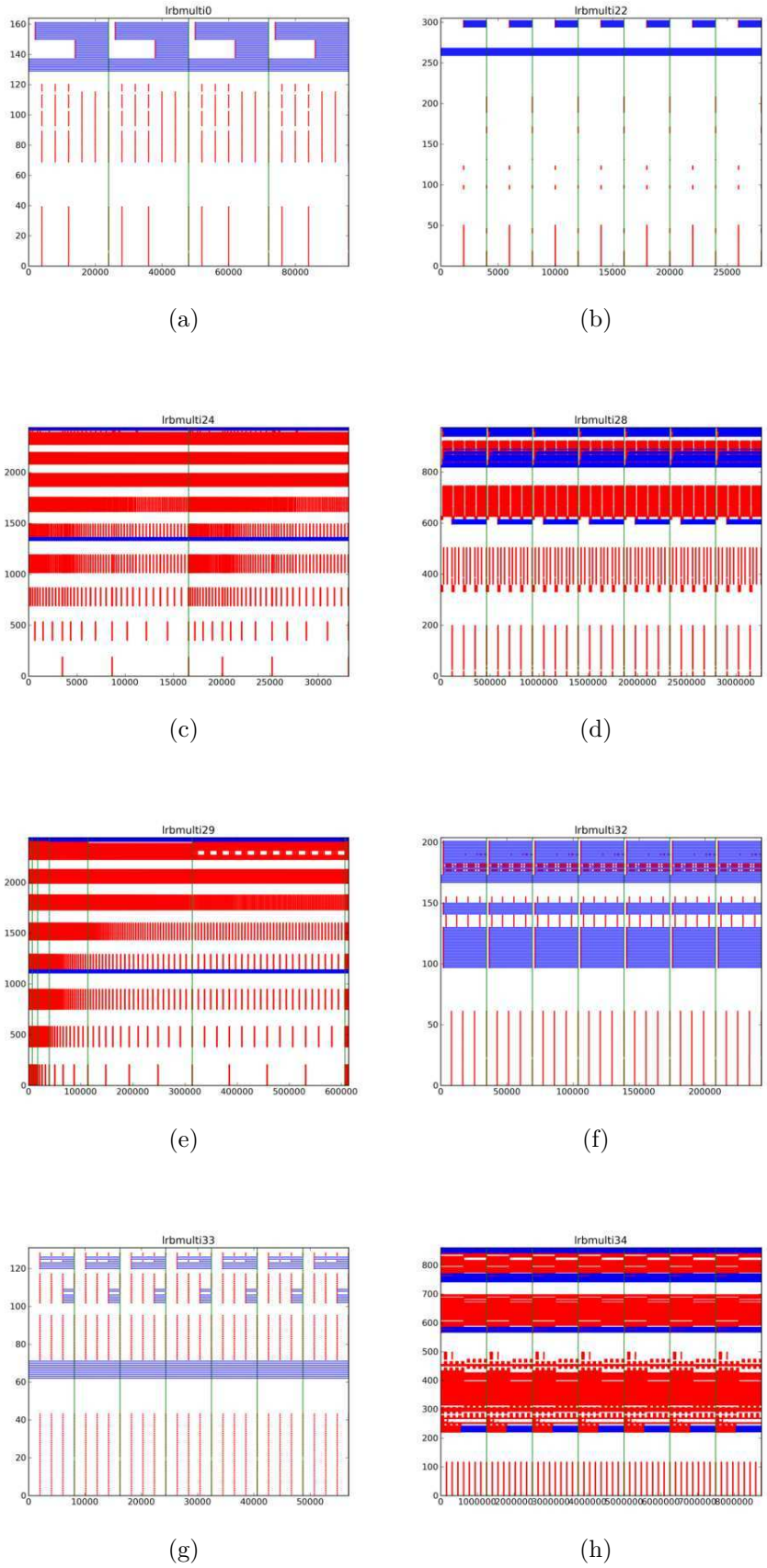
- the horizontal axis represents the iteration number of the outermost loop;
- the vertical axis lists all memory accesses;
- the vertical green bars indicate the starting point of a subloop;

Horizontally, for each memory access, color blue indicates a memory access following a linear function, red - unlinear function, whereas white signifies that no memory access was performed by the monitored instruction in that particular iteration.

The graphs show that some of the monitored instructions exhibit a linear memory accessing behavior during the whole execution of the code extracts, and in some situations, one can identify phases that repeat during one run. These are promising results, showing that general purpose codes could exhibit the linear behavior that our system targets.

Conclusions This application illustrated VMAD as an infrastructure for dynamic profiling, where advanced instrumentation, such as interpolating memory accesses, can be implemented with almost negligible runtime overhead, of less than 4% in most cases, with -O0 optimization level; and varying between 0,5% and 27% with -O3 optimization level. In addition to a reduced overhead, this evaluation proves that the strategy of instrumenting by sampling the first ten iterations of each loop is viable, and can be successfully integrated in any system that requires advanced dynamic analysis.

The next section, presents the results of speculative and polyhedral parallelization with VMAD, relying on the results of this instrumentation.



(a) (b)

(c) (d)

(e) (f)

(g) (h)

Figure 6.8: Instrumenting memory accesses in SPEC CPU 2006

6.2 Speculative parallelization

In this section we present the experiments we conducted to evaluate our approach of applying the polyhedral model at runtime, on non-statically analyzable codes, in the view of speculatively parallelizing the loop nests. Our benchmarks were run on two architectures. The first platform embeds two AMD Opteron Processors 6172, of 12 cores each, at 2.1 Ghz, running Linux 3.2.0-27-generic x86_64, while the second platform is an Intel Xeon X5650 at 2.67GHz, with 12 cores hyper-threaded, running Linux 3.2.0-24-generic x86_64. We have selected a set of benchmarks from different sources: the polyhedral benchmark suite [95], the Rosetta code website [107], the Rodinia benchmark suite [24] and the DSPstone benchmarks [36]. Additionally, we have written some codes implementing classic linear algebra algorithms. Notice that although some of these codes could have been handled statically, they are used to show the effectiveness of VMAD. We have modified them to use dynamically allocated arrays or pointers, which would prevent static analysis. Our findings are indicated graphically in Figures 6.9 and 6.10, and the absolute values are given in the tables from Figures 6.12, 6.13 and 6.14. We compare the speed-up of our system relatively to manual parallelization using OpenMP, when possible, on both architectures. Also, to assess the scalability of our system with respect to the number of threads, we provide the values obtained by comparing the multi-threaded execution time of a code parallelized by VMAD, to the single-threaded execution of the same code (VMAD self).

One of the interesting outcomes is that the behavior of the same code on the two different architectures is very different, in terms of scalability with the numbers of threads. We observed that this behavior is not specific to VMAD, as we obtained similar results when parallelizing the codes manually, with OpenMP. Each processor is better adapted for a particular type of applications, and the codes benefit differently from the hardware support such as the hardware prefetcher, the branch predictor, etc. This is well illustrated by `adi` in Figure 6.9a, where VMAD's speed-up is close to the one obtained from manual parallelization on the Intel processor, but it is outperformed on the AMD processor.

In some situations, we observed that even without any transformation other than straightforward parallelization of the outermost loop, VMAD outperforms OMP, thanks to the execution in chunks of the loops, which is similar to strip-mining, having a positive effect on data locality. With `adi` on the Intel processor, the execution with 16 threads of VMAD is slightly better than the code parallelized with OMP. The gains of the chunking are masked by the inherent overhead of VMAD. However, this improvement is visible when analyzing the scalability of VMAD in the same situation, as one can remark that VMADs self speed-up reaches values of over 15, while OpenMP code' speed-up is lower than 5. Also, the benefits of chunking and of the parallelization are hidden by the overhead when running `adi` on the AMD processor. We analysed the overhead and we concluded that the bottleneck is the strategy of saving the data in advance using `memcpy`. More details on the overhead of VMAD are given in the end of this section.

Figure 6.9b illustrates the behavior of the `backprop` code, which can be parallelized in its original form using OpenMP. This code is handled similarly by any traditional

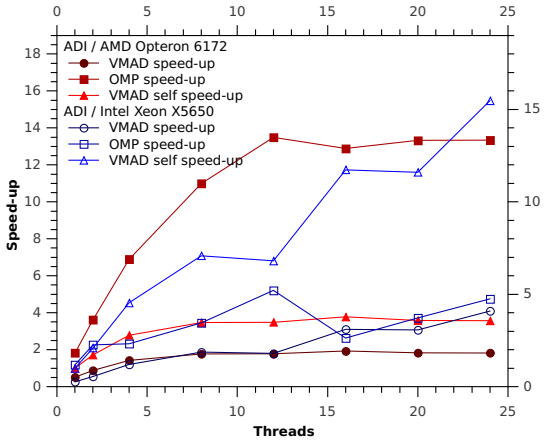
TLS system, by parallelizing the outermost loop. On the other hand, VMAD discovers that a loop interchange is possible, which brings significant performance gains, by improving data locality. VMAD even outperforms OMP code on the AMD processor, up to 16 threads. On the Intel architecture, both VMAD and OpenMP show low scalability with the number of threads. This benchmark underlines the fact that VMAD can significantly improve even embarrassingly parallel codes, unlike traditional TLS systems, even outperforming manual parallelization. Additionally, it shows that the runtime overhead of the system is hidden by the gains provided by applying the polyhedral transformation.

Another example highlighting this contribution to the state of the art is `cholesky` (Figure 6.9c), which is not parallel in its original form. Therefore, previous TLS systems cannot handle this code, nor can it be manually parallelized with OpenMP, since every loop carries dependences. In contrast, VMAD analyses the runtime behavior of the code and finds a suitable polyhedral transformation which allows the loop from the second level to be executed in parallel. The selected transformation is indicated in the tables from Figures 6.12, 6.13 and 6.14. Although the speed-up obtained currently is not very high, one can remark that VMAD self speed-up scales with the number of threads, up to 12 threads. Our finding is that the code transformation has a negative impact on data locality, a problem that we can address by tiling the transformed loop. Building patterns that support tiling is one of the first targets of our future work.

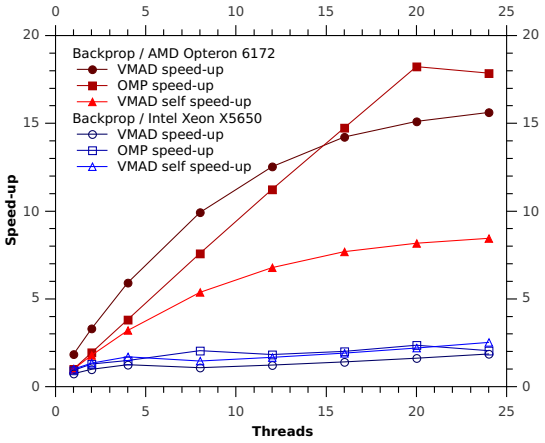
The example in Figure 6.9f, `floyd`, illustrates the capability of our system to adapt dynamically to the behavior of the code and to exploit partial parallelism. This benchmark embeds a conditional that does not allow parallelization because it does not have a predictable behavior during the first iterations. Nevertheless, VMAD executes a sequential chunk and monitors again the loop. The second profiling phase identifies that one branch of the conditional is always executed, thus parallelization is possible. Moreover, the result of the dependence analysis indicates the second loop level as being parallel. This represents another example where VMAD can exploit parallelization opportunities which are not accessible by manual parallelization or by traditional TLS systems, which generally simply parallelize the outermost loop.

In Figure 6.10a, we depict the benchmark `fir2dim` which contains a loop nest of depth 3 performing memory accesses via pointers. Arrays are represented as dynamically allocated pointers and their parsing is performed by using pointer arithmetic. OpenMP fails to parallelize such codes, due to the impossibility of predicting the starting value of the pointers for each thread. On the contrary, VMAD is successful in parallelizing these examples, thanks to its instrumentation phase, which builds interpolating linear functions. Thus VMAD successfully predicts the starting values of the pointers. Our technique is to privatize the pointers and to initialize them with the predicted value, such that all iterations can be run in parallel.

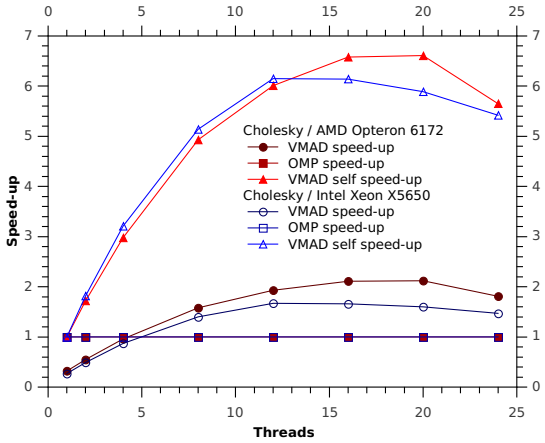
Other examples, such as `covariance` and `correlation` in Figures 6.9e and 6.9d show that codes parallelized with VMAD have overall a good performance. Nevertheless, `grayscale` and `QRdecomp` in Figures 6.10b and 6.10c show that in some situations, although VMAD codes scale with the number of threads, the overhead of the system could be reduced.



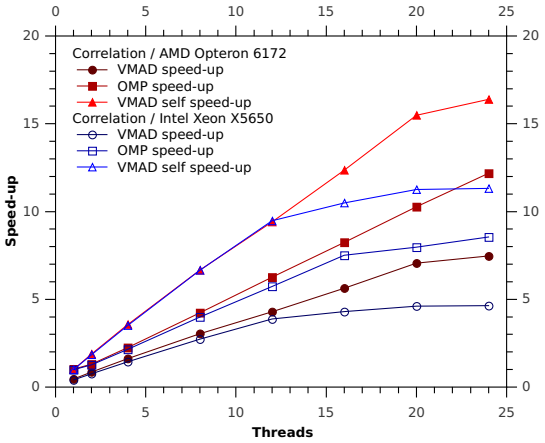
(a) Adi



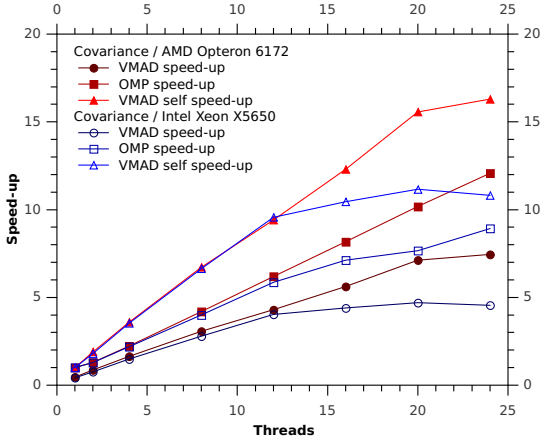
(b) Backprop



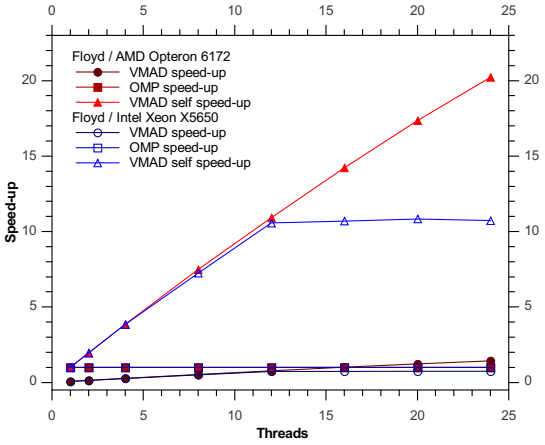
(c) Cholesky



(d) Correlation

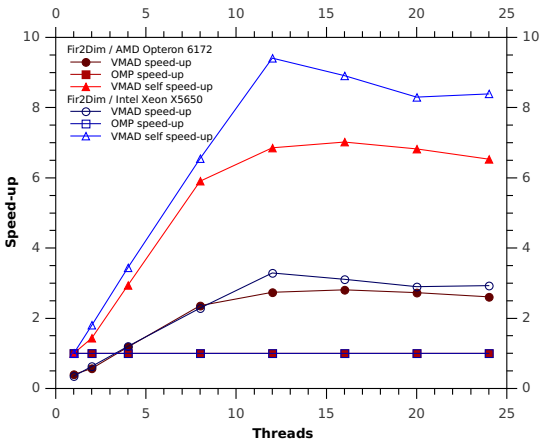


(e) Covariance

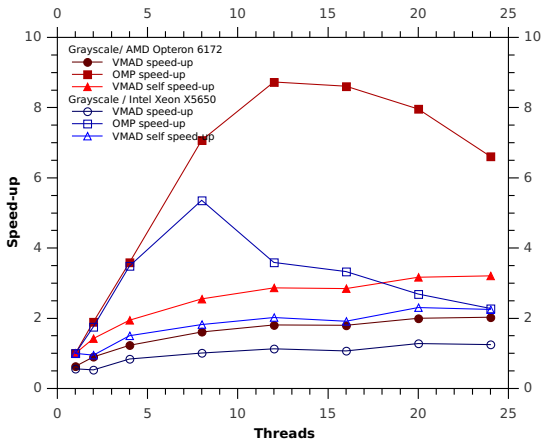


(f) Floyd

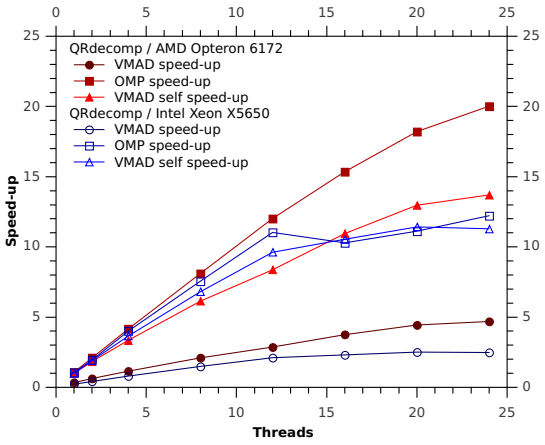
Figure 6.9: Speculative parallelism results I



(a) Fir2Dim

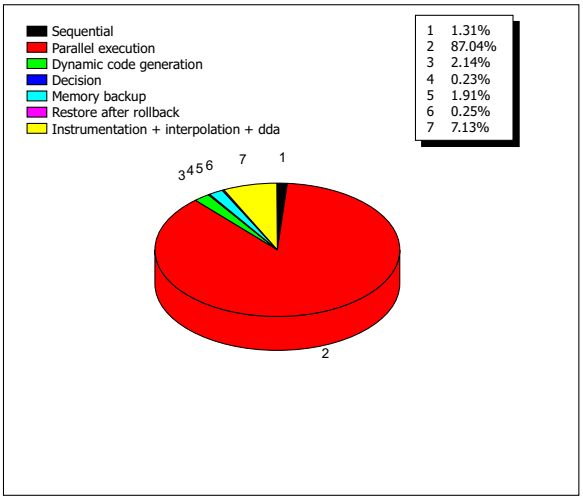


(b) Grayscale

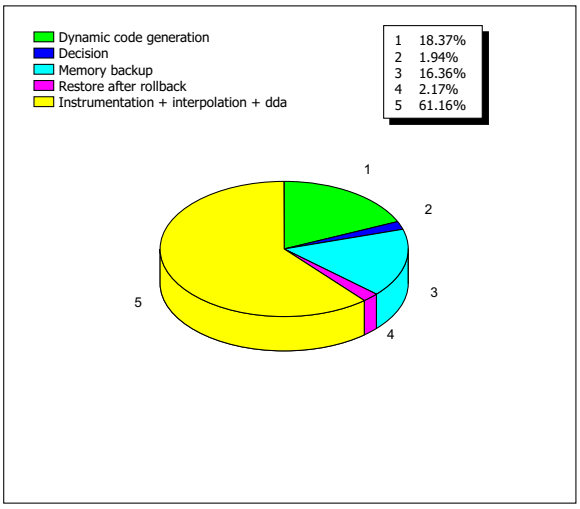


(c) QRdecomp

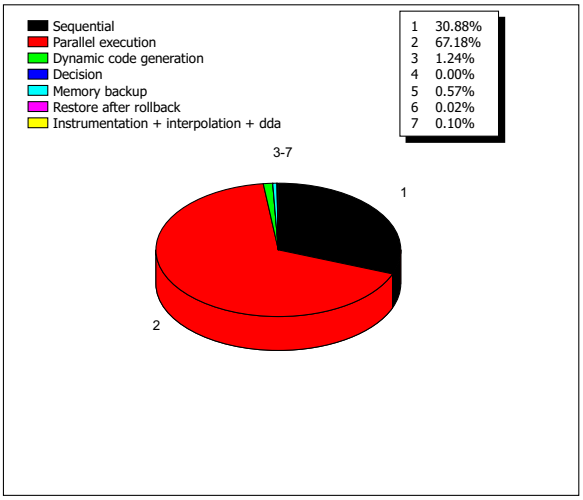
Figure 6.10: Speculative parallelism results II



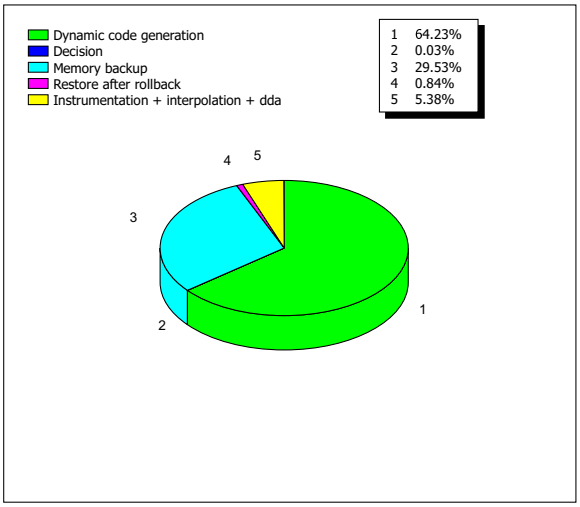
(a) Covariance



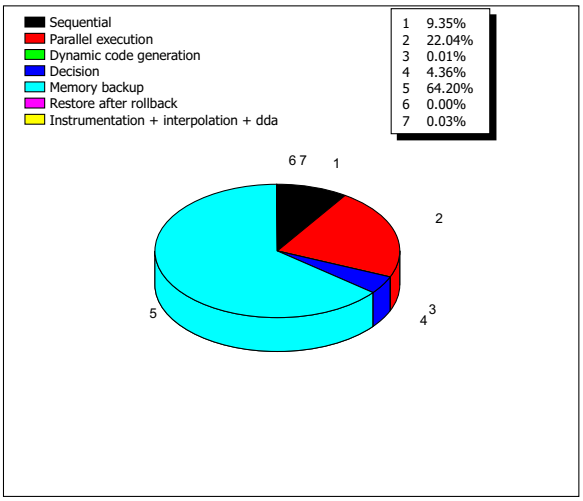
(b) Covariance overhead



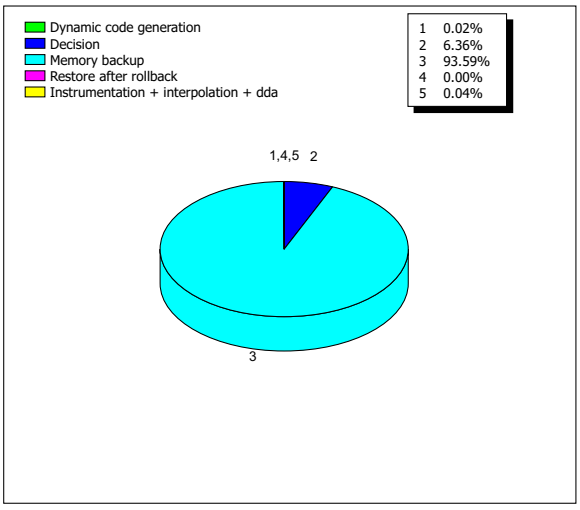
(c) Backprop



(d) Backprop overhead



(e) Adi



(f) Adi overhead

Figure 6.11: Percentage of time spent by VMAD in different execution phases

TRANSFORMATION $= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$: identity, 1st loop is parallel

	THREADS	VMAD SELF SPEED-UP	VMAD SPEED-UP	OMP SPEED-UP
ADI / AMD Opteron 6172	24	3.57	1.82	13.34
	20	3.58	1.83	13.33
	16	3.78	1.93	12.89
	12	3.48	1.78	13.49
	8	3.47	1.77	10.99
	4	2.78	1.42	6.89
	2	1.72	0.88	3.62
	1	1.00	0.51	1.82
ADI / Intel Xeon X5650	THREADS	VMAD SELF SPEED-UP	VMAD SPEED-UP	OMP SPEED-UP
	24	15.48	4.09	4.75
	20	11.60	3.07	3.72
	16	11.73	3.10	2.63
	12	6.81	1.80	5.21
	8	7.08	1.87	3.45
	4	4.54	1.20	2.32
	2	2.09	0.55	2.26
	1	1.00	0.26	1.18

TRANSFORMATION $= \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$: loop interchange, 1st loop is parallel

	THREADS	VMAD SELF SPEED-UP	VMAD SPEED-UP	OMP SPEED-UP
Backprop / AMD Opteron 6172	24	8.45	15.62	17.86
	20	8.17	15.11	18.23
	16	7.69	14.22	14.74
	12	6.78	12.53	11.24
	8	5.37	9.92	7.59
	4	3.20	5.92	3.82
	2	1.79	3.31	1.94
	1	1.00	1.85	1.01
Backprop / Intel Xeon X5650	THREADS	VMAD SELF SPEED-UP	VMAD SPEED-UP	OMP SPEED-UP
	24	2.52	1.86	2.05
	20	2.20	1.62	2.36
	16	1.91	1.41	2.01
	12	1.67	1.23	1.83
	8	1.46	1.08	2.05
	4	1.70	1.25	1.49
	2	1.34	0.99	1.29
	1	1.00	0.74	0.93

TRANSFORMATION $= \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$: polyhedral transf., 2nd loop is parallel

	THREADS	VMAD SELF SPEED-UP	VMAD SPEED-UP	OMP SPEED-UP
Cholesky / AMD Opteron 6172	24	5.65	1.81	N/A
	20	6.61	2.12	N/A
	16	6.58	2.11	N/A
	12	6.01	1.93	N/A
	8	4.93	1.58	N/A
	4	2.98	0.96	N/A
	2	1.72	0.55	N/A
	1	1.00	0.32	N/A
Cholesky / Intel Xeon X5650	THREADS	VMAD SELF SPEED-UP	VMAD SPEED-UP	OMP SPEED-UP
	24	5.42	1.47	N/A
	20	5.89	1.60	N/A
	16	6.14	1.66	N/A
	12	6.15	1.67	N/A
	8	5.14	1.40	N/A
	4	3.21	0.87	N/A
	2	1.82	0.49	N/A
	1	1.00	0.27	N/A

Figure 6.12: Code speculatively parallelized with VMAD, compared to OpenMP (I)

TRANSFORMATION $= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$: identity, 1st loop is parallel

	THREADS	VMAD SELF SPEED-UP	VMAD SPEED-UP	OMP SPEED-UP
Correlation / AMD Opteron 6172	24	16.39	7.47	12.18
	20	15.48	7.06	10.27
	16	12.35	5.63	8.24
	12	9.41	4.29	6.26
	8	6.66	3.04	4.23
	4	3.56	1.62	2.25
	2	1.89	0.86	1.31
	1	1.00	0.46	1.02
	THREADS	VMAD SELF SPEED-UP	VMAD SPEED-UP	OMP SPEED-UP
Correlation / Intel Xeon X5650	24	11.32	4.64	8.55
	20	11.26	4.61	7.97
	16	10.49	4.30	7.51
	12	9.48	3.88	5.73
	8	6.65	2.73	3.99
	4	3.51	1.44	2.15
	2	1.84	0.76	1.27
	1	1.00	0.41	0.99

TRANSFORMATION $= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$: identity, 1st loop is parallel

	THREADS	VMAD SELF SPEED-UP	VMAD SPEED-UP	OMP SPEED-UP
Covariance / AMD Opteron 6172	24	16.30	7.45	12.07
	20	15.56	7.12	10.18
	16	12.30	5.62	8.17
	12	9.41	4.30	6.19
	8	6.71	3.07	4.19
	4	3.59	1.64	2.23
	2	1.90	0.87	1.30
	1	1.00	0.46	1.01
	THREADS	VMAD SELF SPEED-UP	VMAD SPEED-UP	OMP SPEED-UP
Covariance / Intel Xeon X5650	24	10.81	4.55	8.92
	20	11.16	4.70	7.66
	16	10.45	4.40	7.12
	12	9.56	4.03	5.86
	8	6.63	2.79	4.00
	4	3.53	1.49	2.19
	2	1.81	0.76	1.29
	1	1.00	0.42	1.00

TRANSFORMATION $= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$: identity, 2nd loop is parallel

	THREADS	VMAD SELF SPEED-UP	VMAD SPEED-UP	OMP SPEED-UP
Floyd / AMD Opteron 6172	24	20.22	1.43	N/A
	20	17.35	1.23	N/A
	16	14.22	1.01	N/A
	12	10.91	0.77	N/A
	8	7.49	0.53	N/A
	4	3.85	0.27	N/A
	2	1.96	0.14	N/A
	1	1.00	0.07	N/A
	THREADS	VMAD SELF SPEED-UP	VMAD SPEED-UP	OMP SPEED-UP
Floyd / Intel Xeon X5650	24	10.73	0.74	N/A
	20	10.83	0.74	N/A
	16	10.69	0.73	N/A
	12	10.57	0.73	N/A
	8	7.24	0.50	N/A
	4	3.84	0.26	N/A
	2	1.95	0.13	N/A
	1	1.00	0.07	N/A

Figure 6.13: Code speculatively parallelized with VMAD, compared to OpenMP (II)

TRANSFORMATION $= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$: identity, 1st loop is parallel

	THREADS	VMAD SELF SPEED-UP	VMAD SPEED-UP	OMP SPEED-UP
Fir2Dim / AMD Opteron 6172	24	6.53	2.61	N/A
	20	6.82	2.73	N/A
	16	7.02	2.81	N/A
	12	6.86	2.74	N/A
	8	5.90	2.36	N/A
	4	2.94	1.18	N/A
	2	1.44	0.57	N/A
	1	1.00	0.40	N/A
	THREADS	VMAD SELF SPEED-UP	VMAD SPEED-UP	OMP SPEED-UP
Fir2Dim / Intel Xeon X5650	24	8.39	2.93	N/A
	20	8.30	2.90	N/A
	16	8.91	3.11	N/A
	12	9.41	3.29	N/A
	8	6.55	2.29	N/A
	4	3.44	1.20	N/A
	2	1.80	0.63	N/A
	1	1.00	0.35	N/A

TRANSFORMATION $= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$: identity, 1st loop is parallel

	THREADS	VMAD SELF SPEED-UP	VMAD SPEED-UP	OMP SPEED-UP
Grayscale / AMD Opteron 6172	24	3.21	2.03	6.61
	20	3.17	2.00	7.96
	16	2.85	1.80	8.61
	12	2.87	1.81	8.73
	8	2.55	1.61	7.07
	4	1.94	1.23	3.59
	2	1.42	0.90	1.90
	1	1.00	0.63	1.00
	THREADS	VMAD SELF SPEED-UP	VMAD SPEED-UP	OMP SPEED-UP
Grayscale / Intel Xeon X5650	24	2.25	1.25	2.28
	20	2.31	1.28	2.69
	16	1.92	1.07	3.33
	12	2.02	1.13	3.59
	8	1.82	1.01	5.36
	4	1.50	0.84	3.49
	2	0.95	0.53	1.76
	1	1.00	0.56	1.00

TRANSFORMATION $= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$: identity, 1st loop is parallel

	THREADS	VMAD SELF SPEED-UP	VMAD SPEED-UP	OMP SPEED-UP
QRdecomp / AMD Opteron 6172	24	13.70	4.69	20.02
	20	12.97	4.44	18.21
	16	10.95	3.75	15.35
	12	8.37	2.87	12.02
	8	6.13	2.10	8.11
	4	3.35	1.15	4.15
	2	1.84	0.63	2.10
	1	1.00	0.34	1.09
	THREADS	VMAD SELF SPEED-UP	VMAD SPEED-UP	OMP SPEED-UP
QRdecomp / Intel Xeon X5650	24	11.29	2.48	12.22
	20	11.42	2.51	11.12
	16	10.54	2.31	10.29
	12	9.62	2.11	11.03
	8	6.81	1.49	7.56
	4	3.65	0.80	4.00
	2	1.91	0.42	1.95
	1	1.00	0.22	1.05

Figure 6.14: Code speculatively parallelized with VMAD, compared to OpenMP (III)

6.2.1 VMAD's runtime overhead

As expected, the overhead of the system strongly depends on the characteristics of the code, since it is relative to the time of the total execution. Thus, for loop nests in which the outermost loop has a high number of iterations, the profiling phases, consisting in instrumentation, interpolation of memory accesses and dynamic dependence analysis, has almost a negligible overhead. In practice, we noticed that in many of the situations, this overhead did not pose significant problems. In Figures 6.11a, 6.11e and 6.11c we depict the time taken by each phase of executing codes with VMAD, relative to the total execution. The `Sequential` phase refers to the execution of the last chunk, which is always executed sequentially, to ensure that all iterations of the loop were executed. `Dynamic code generation` is the time taken to specialize the patterns, using runtime information. `Decision` is the time taken by the runtime system to select the type of the next chunk to be launched and to set its size. `Memory backup` is the safe copy that is performed before launching a speculative chunk. `Restore after rollback` is the time required to restore the correct state of the memory, upon a misprediction. And finally, the time taken to instrument the code, interpolate the results and run the dynamic dependence analysis is depicted as `Instrumentation + interpolation + dda`. The figures are shown in pairs. On the left is illustrated the total execution time divided in the time taken by each action, while the figure on the right focuses only on the overhead incurred by the runtime system. Thus, on the right we illustrate the amount of time taken by the runtime system to perform additional actions, before executing the loop or in between the chunks.

The first pair of figures 6.11a and 6.11b depicts the execution of the `covariance` example. As one can notice, the execution of the parallel chunks governs the total execution, and the overhead is rather limited. Most notable, the time taken for instrumentation and dependence analysis, represents 7.13% of the total execution, as we instrument 10 iterations of the outermost loop's 1500 iterations.

The dynamic code generation represents a part of the overhead, however, compared to the total execution, its overhead is negligible. this argues in favor of using code patterns.

The second pair of figures 6.11c and 6.11d shows the overhead of the system when executing the `backprop` benchmark. One can notice that the overall overhead is rather small, most notable the dynamic code generation on this example is more costly, due to numerous memory accesses that must be verified. Nevertheless, even in this situation, its overhead is only 1.24%.

The last pair, is an extract from `adi`, on which our system has a considerable slowdown. By analyzing the overhead over all benchmarks, we noticed that in many situations, the bottleneck is the back-up of the memory, using `memcpy`. This routine is highly optimized, depending on the architecture and we have noticed differences from one machine to another. But it has a considerable impact on performance, since the volume of data we require to save between the chunks is in general proportional with the chunk size. Therefore, our next goal is to lower the overhead of VMAD, by optimizing our strategy to back-up data. Some preliminary ideas are presented in the last chapter, presenting our short-term perspectives.

All in all, VMAD provides important contributions and advancements to the state of

the art and is successful in optimizing and paralellizing general purpose codes, that are not accessible to traditional TLS systems. The system can handle codes in any form and is not hindered by the type of memory allocation, being capable of handling pointers, static array accesses, multi-dimensional or linearized arrays. Moreover, unlike OpenMP codes, VMAD can handle multiple exit loops and pointer-chasing loops. In contrast, VMAD does not yet readily support reductions, however, this does not represent an important cornerstone for the future extensions, as one can consider a dependence carried by the loop which contains the reduction.

We conclude by reminding the main contributions, underlined by the benchmarks:

1. VMAD is able to parallelize codes which do not exhibit parallelism in the original form. Thus they cannot be handled efficiently by existing TLS systems (due to numerous rollbacks) and cannot be parallelized by hand, unless a transformation is applied.

2. VMAD can discover optimization opportunities in codes that can already be parallelized in the original form. By applying such optimizing transformations prior to parallelization, the performance of the generated code is significantly boosted.

3. Given that the parallel code does not have the expected performance gains, the overhead of VMAD is, in some situations, cancelled by the performance improvement provided by the polyhedral transformation.

Chapter 7

Other applications

In addition to performing speculative parallelization, VMAD has been designed as a generic platform suitable for various types of advanced code analyses and transformations. Since the modules of the runtime system are decoupled, the strength of our system resides in its extensibility, as one can add support for new profiling or optimization strategies, independently of the existing modules. In what follows we present two other applications that rely on some of VMAD's mechanisms, such as multi-versioning or chunking. The first application performs a very simple but efficient dynamic dependence analysis of codes, using the interpolating linear functions presented in Chapter 6, Section 6.1. The second application is dedicated to dynamically adapting to the current execution context and environment, by selecting at runtime the optimal version, among several statically generated versions.

7.1 Simple dynamic dependence analysis

To underline the usability of the instrumentation presented in Chapter 6, Section 6.1, we developed a very simple, yet efficient method for dynamic dependence analysis, based on value range analysis and the greatest common divisor (gcd) tests, as depicted in Figure 7.1.

The analyzer *pp* [72] is one of the earliest work that proposed hierarchical dependence testing to estimate the parallelism in loop nests. Some recent works are Alchemist [136] and SD3 [68] where runtime and memory overhead is reduced through the use of parallelization and compression.

For the purpose of this application, we added a module to VMAD to determine, for a loop nest, which are the loop levels that might be parallelized, according to the memory behavior observed during profiling. Such information can be a useful indication for a developer in order to identify and further analyze such loops, to decide whether they can be parallelized. Our framework identifies the candidate loops by speculatively analyzing dependences between iterations, based on the linear functions interpolating the memory addresses accessed during profiling. The module considers each couple of memory instructions and their associated linear functions, where at least one is a write.

We use a simple value range analysis method to determine if the two referenced address spaces can overlap, using the linear functions and the loop bounds to com-

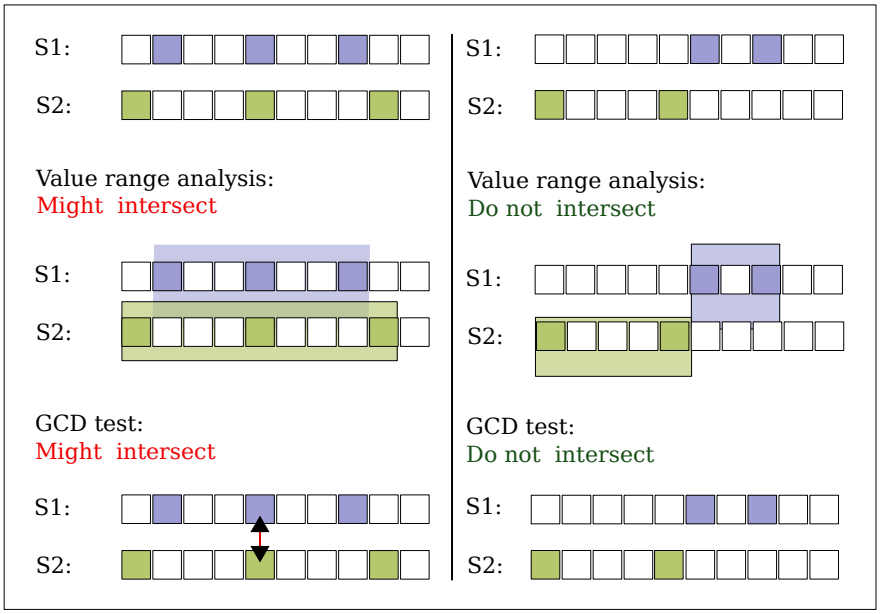


Figure 7.1: Dynamic dependence analysis with *value range analysis* and *gcd* tests

pute the minimal and the maximal values of the memory addresses accessed by each instruction. Each write instruction is also considered solely since it can carry an output self-dependence. A loop level not carrying any dependence is then identified as a candidate for parallelization. We used the OmpSCR benchmark suite [91] for our experiments, a set of scientific kernels that are already manually parallelized by the programmer using OpenMP pragmas. Even if these have been deactivated for our runs, they indicate loops being effectively parallel. Loops inside these kernels contain memory references through pointers, through parameterized array accesses and references to dynamically allocated arrays. Such memory references cannot be handled statically by a compiler. Results are shown in table 7.1. The columns respectively show the benchmarks, the number of loop nests originally marked with OMP pragmas, the number of loop nests we identified as having a linear behavior, and the number of loop nests detected as parallel. Finally, the last column indicates for each parallel loop nest which is the parallel loop level. For instance, in FFT6 we detect 4 loop nests suitable for parallelization: in the first nest, the 1st loop is parallel; in the second nest - the 3rd loop level; in the third nest both the 1st and 2nd loops can be parallelized; and the same in the fourth loop nest. We target for instrumentation and analysis all loop nests of the program, thus, for two benchmarks, FFT6 and LUreduction, more loop nests than the ones marked with OpenMP pragmas were detected as parallel. When less parallel loop nests are detected, it is due to dependences induced by reductions.

Remarks. In a nutshell, the dependence analysis test we developed, although simple, proved to be very efficient. Moreover, the instrumentation built for the purpose of computing interpolating linear functions of the memory accesses showed its utility in other goals than speculative parallelization. Similarly, it could be engaged in data prefetching, to boost performance, or involved in analyses targeting different types of

Table 7.1: Simple dynamic dependence analysis and parallel loop detection in the OmpSCR benchmark suite.

Benchmark	#OMP pragmas	#Linear loop nests	#Detected as parallel	Parallel loop levels
FFT	2	2	0	
FFT6	3	10	4	1 / 3 / 1,2 / 1,2
Jacobi	2	4	1	1,2
LUreduction	1	2	2	1,2 / 2,3
Mandelbrot	1	2	1	1
Md	2	2	1	1,2
Pi	1	1	0	
QuickSort	1	2	1	1

predictions and estimations on the codes behaviors.

7.2 Runtime version selection

Frequently, the execution context has a great impact on the performance of the code. In consequence, even highly optimized codes might perform under expectations on different architectures. Similarly, among several optimized versions of the same code, the optimal version could differ, with respect to the current context (data size, processor load) and execution platform.

Several studies proposed a runtime selection between various algorithms, or code extracts, or versions of a function. PetaBricks [3] provides a language and a compiler where having multiple implementations of multiple algorithms to solve a problem is the natural way of programming. Mars and Hundt's [84] static/dynamic SBO framework consists in generating at compile-time several versions of a function that are related to different dynamic scenarios. The STAPL [116] adaptive selection framework runs a profiling execution at install time to extract architectural dependent information. Pradelle *et al.* [102] propose a framework to select between versions of loop nests resulting from various polyhedral transformations.

A loop nest can be optimized using different kinds of transformations such as loop fusion/fission, interchange, skewing, tiling, unrolling, etc. A subset of those transformations can be applied, in different order, or with different parameters (unrolling factor, tile size, ...) to generate distinct versions. Hence many versions can be obtained in this way, and each of them may be the best performing one in some execution contexts, while being slower in some others. Such a phenomenon can occur, for example, when the amount of accessed data generates a lot of cache misses if the computation size exceeds a given threshold. Another case is when the locality of the data accesses depends on some input parameters, or when the control flow traverses costly branches in some circumstances depending on intermediate computations. More exactly, it is a combination of such phenomena that impacts the global performance. Hence, it is in general impossible to predict in advance which version would yield the best execution time.

Table 7.2: Dynamic code selection with VMAD.

Benchmark	#Versions	Best exec. time	Worst exec. time	Average exec. time	VMAD exec. time	Gap to the average version
2mm	6	2.68	19	8.29	4.80	-42.09%
adi	7	32.99	34.17	33.24	33.10	-0.42%
covariance	6	9.71	145.55	55.81	17.54	-68.5%
gemm	6	7.21	57.10	15.79	9.94	-37.04%
jacobi-1d	6	8.34	11.05	9.70	9.72	0.2%
jacobi-2d	6	2.74	5.24	4.12	4.22	2.42%
lu	6	3.94	51.26	12.11	6.31	-47.89%
matmul	7	4.96	31.49	16.90	6.96	-58.81%
matmul-init	6	3.29	27.04	7.38	4.72	-36.04%
mgrid	6	11.58	16.50	13.45	13.03	-3.12%
seidel	6	76.59	87.71	85.07	86.66	1.86%

The implemented runtime mechanism consists in first measuring the time per iteration when executing a small chunk of each version, and then running the fastest one for the remaining iterations. Different versions are provided in the source code, delimited by dedicated pragmas. Each version contains an additional condition in the outermost loop, which constrains the iterator between a lower and an upper bound, required for the chunking mechanism.

At compile-time our modified LLVM compiler identifies the multiple versions and a callback to the dedicated runtime selector module is automatically added, as well as the mechanism to switch between the versions.

The runtime module performs the following operations: for each version, one by one, it sets the chunk bounds such that each new chunk will continue the execution of the previous one, it gets the processor's time stamp counter using the RDTSC instruction, launches the version, gets the new CPU time information, computes the execution time per iteration and stores a reference to the fastest version so far. Finally, when all versions have been evaluated, the fastest version is launched to complete the execution. This naive approach already selects the best version in most cases, but the algorithm can be further refined. Similarly to the sampling rate in the first example, the size of the instrumented chunk can be set by a parameter.

The benchmark programs contain 11 benchmarks. The code `2mm` consists of two matrix multiplications ($D = A \times B \times C$), `adi` is the ADI kernel provided as an example with the automatic optimizer Pluto [15], `covariance` is a covariance matrix computation, `gemm` is taken from BLAS [12], `jacobi-1d` and `jacobi-2d` are the 1D and 2D versions of the Jacobi kernel, `lu` is a LU decomposition kernel, `matmul` is a simple matrix multiplication, `matmul-init` is a matrix multiply combined with the initialization of the result matrix, `mgrid` is a kernel extracted from the `mgrid` code in SPECOMP [5] and `seidel` is a Gauss-Seidel kernel also provided with Pluto.

Such loops are good candidates for loop optimizations such as skewing, loop interchange or tiling. We generated 6 or 7 different versions for each benchmark, either using Pluto or manually. Some versions are tiled, some others are tiled two times in two levels, some others are just skewed or their loops have been interchanged, and

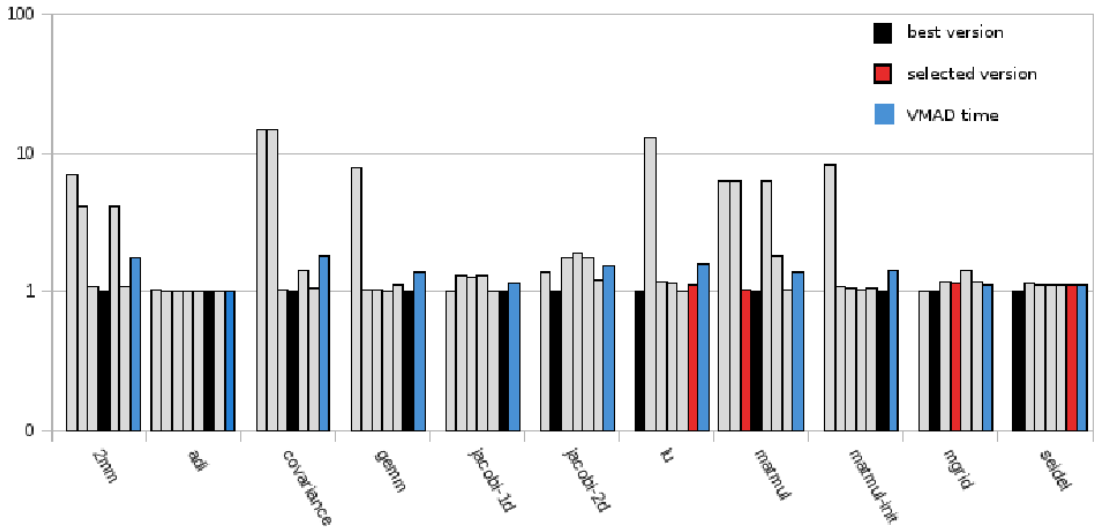


Figure 7.2: Dynamic code selection with VMAD, Logarithmic scale (courtesy: Ph. Clauss)

finally some are the result of a combination of these transformations. All versions, as well as VMAD's code selector, have been run in sequential on an Intel Xeon W3520 at 2.67Ghz under Linux 2.6.38. Numerical results are shown in table 7.2 and a graphical representation is given in Figure 7.2. For each benchmark, it shows the execution time of the best and of the worst version, the average execution time of all versions, the time when executing with VMAD, and finally a comparison between VMAD and the average execution time. The histogram shows in gray the execution time of each version, in blue the execution time of VMAD, in red the selected version and in black the best one. When the red bar is missing, the selected and the best versions coincide.

Remarks. To conclude, VMAD selects the best version, or one of the best if the execution times are very similar. Also, its execution time is close to the best execution times. The overhead is higher when some versions are very slow compared to others, but negligible when the versions have very similar execution times. Hence, this makes our framework suitable for detecting at runtime the best performing version according to the current execution context, among several optimized versions that already exhibit good performance.

7.3 Conclusions

In addition to the examples presented above, other applications are also envisaged. VMAD can find its applications in distributed debugging or instrumentation, among multiple users. Thanks to the sampling approach and the multiple versions, the selection mechanism can be adjusted such that the version chosen for execution differs from one user to another. Moreover, each version contains only a subpart of the instrumenting or debugging instructions, which ensures a very low overhead, but together, the

instrumentation inserted in all versions cover the entire targeted code. Distributed debugging or instrumentation becomes attractive when there is a high number of testers, as each version is executed at least by one user. Moreover, since the overhead is negligible, users are not hindered from executing the versions multiple times. On the other hand, when overhead is not a concern, the framework can be employed for fully tracing the behavior of the code. This can be achieved by setting the chunk size to a maximal value and selecting the instrumented version.

Chapter 8

Conclusions

The contributions presented in this dissertation belong to the sphere of speculative parallelization of loops, by applying polyhedral transformations at runtime and dynamically tuning the optimizations, with respect to the current behavior of the code, in order to maximize performance. Our proposal takes the shape of a fine-grain TLS system, which provides the following advancements to the state of the art.

- To our knowledge, VMAD is the first proposal that performs **dynamic optimizations of loops**, prior to **parallelization**. In this manner, one can address non-statically analyzable codes and significantly boost their performance, by using information that becomes available only during execution.
- In contrast to previous TLS systems, VMAD **discovers parallelization opportunities** in codes that could not be parallelized in their original form, and applies a polyhedral transformation to explore such opportunities. This is a key contribution to automatic parallelization of loops, as the responsibility of performing dependence analysis and verification of the speculations is transferred to the system. Moreover, VMAD succeeds to parallelize loops that are not targeted by state of the art tools, or which would be tedious to be analyzed and parallelized by a programmer.
- Our proposal is adapted to **exploit partial parallelism** in loops with a changing behavior, such that our optimization and parallelization strategies are not hindered by execution phases when the loop nest can only be run in sequential order.
- Additionally, not only the system can identify different phases of a loop, but it is designed to optimize the code according to the current phase. Thus, during one execution of a loop, the system links different versions, each of them generated by applying a different polyhedral transformation on the original code, such that the generated code performs best under the current conditions. This makes the code optimized with VMAD invulnerable to variations of the input data or to the execution environment.

Our proposal has been implemented as a platform for dynamic profiling, optimizations and speculative parallelization. The framework consists of a static part, which

prepares the code at compile time, and a dynamic component, which orchestrates the execution. We extended the LLVM compiler to handle specific `pragmas`, allowing the developer to initiate automatic speculative parallelization from selected parts of the source code. Speculative transformations rely on advanced instrumentation and analyses, which in practice show almost negligible runtime overhead, since VMAD does not use software dynamic translation like most of the dynamic profiling tools. Similarly, runtime code generation of the parallel code is achieved by means of a set of statically prepared code patterns, that are specialized at runtime, with respect to the observed behavior of the code. This approach makes our system very fast and efficient. The runtime system has the responsibility of orchestrating the execution, by launching successive chunks of the loop embedding a corresponding version, in accordance with the memory accessing behavior of the loop. Should the loop exhibit several phases during one execution, the runtime system automatically detects each phase and launches a new, adapted code version. The parallel versions are generated from the code patterns, by applying a suitable polyhedral transformation. In contrast to many previous TLS proposals which divide the outermost loop in parallel chunks for a speculative execution, we target to execute the chunks one after the other, but speculatively parallelize each chunk with a different schedule, if necessary. This strategy allows us to identify partial parallelism in loops and to apply the most suitable polyhedral transformation for each chunk.

To evaluate our proposal, we conducted experiments by speculatively parallelizing non-statically analyzable loops and applying polyhedral transformations, to optimize the code or to exhibit new parallelization opportunities. Our findings were that the system introduces a limited overhead, which is masked by the benefits brought by loop parallelization and polyhedral transformations. Moreover, several other applications were presented, which emphasize VMAD as a general platform, suitable for various types of code analyses and optimizations.

This thesis brings significant contributions in several very well-researched areas such as dynamic tracing of memory accesses, dynamic dependence analysis and automatic speculative parallelization, by applying the polyhedral model at runtime. Our experiments yielded promising outcomes on the applications we evaluated. Although the system already proves to be efficient, we are aware of some limitations imposed by our design choices. Imposing a linear memory accessing behavior of loops could significantly reduce the class of codes which can be handled by VMAD. Building patterns at compile time has an impact on the type of polyhedral transformations that can be applied dynamically in order to generate optimized parallel versions. Algorithms, such as the one dedicated to the selection of the polyhedral transformation to be applied, require further refinement for optimal results. All these and many other aspects that we envisaged to enhance in our system are presented in the next chapter.

Chapter 9

Perspectives

In our opinion, VMAD already is a first proof that the benefits of the polyhedral model are now accessible to general purpose codes, that require dynamic analysis and transformations. We have fully implemented a working prototype, yet, among the short term goals, we would like to start by revisiting some of the implementation details, in the view of reducing the runtime overhead of the framework.

To mention a few strategies, since the runtime system currently has a sequential implementation, one could focus on parallelizing some of its operations. For instance, during the instrumentation phase of a loop, all instructions place the values to be interpolated in a designated buffer, and at the end of each iteration, the runtime system accesses the buffer and computes the interpolating linear functions. Since each instruction accesses its specific location in the buffer and they are all independent, the computation of the linear functions at the end of each iteration could be easily parallelized.

Next, in the process of speculatively parallelizing a loop, the runtime system creates a safe-copy of the memory that is predicted to be accessed by each parallel chunk. This is performed before launching the chunk, by several calls to the `memcpy` function, to save ranges of memory. Nevertheless, this is a costly operation. Another approach would be to perform a copy of each memory location, inside the threads, just before it is modified by the thread. Thus, the copies are performed in parallel, writing each accessed location into its unique corresponding position in a buffer. Moreover, upon a rollback, the runtime system can restore the memory state from the buffer, again, copying the data in parallel for each location. Similarly, several other aspects of VMAD could be improved, by polishing the implementation.

Inspecting several suites of benchmarks, we noticed that often, the most time consuming regions of codes are loop nests which not only may have a considerable number of iterations, but also, they are contained in frequently invoked functions. Consequently, should the loop nests exhibit a similar behavior during several consecutive invocations of their enclosing functions, the runtime system could use the *history* of previously valid transformations. In this manner, the runtime system can perform the instrumentation phase, dependence analysis and selection of transformation during the first few invocations, and maintain a history of transformations that proved to be legal during the execution of the parallel version. Next, future invocations could be

speculatively parallelized by applying transformations from the *history* list. Moreover, the runtime system can store the value of the iteration starting a faulty chunk, such that, in the next invocations it can directly compute the maximal valid chunk size and launch a parallel version that executes (almost) all valid iterations, directly followed by an original, sequential chunk. Thus the rollback costs are eliminated, given that the loop preserves its behavior from one invocation to another.

Reducing the overhead of the framework is an important goal, but it does not suffice to guarantee maximal performance. In what follows, we present a few other techniques we envisage for boosting the performance of the generated code. Currently, we have implemented only a naive algorithms for the selection of the polyhedral transformation. Namely, we apply the first transformation we detect to be valid, according to the set of loop carried dependences. However, consistent research has been devoted to find the optimal polyhedral transformation, given criteria such as data locality, parallelism *etc.* In this respect, we consider of utmost importance to design a fast algorithm dedicated to selecting at runtime a suitable and efficient transformation. One of the first techniques we investigated, was to apply several of the most promising transformations on small chunks and using our runtime version selection to evaluate these chunks and select the best transformation. Nevertheless, this approach shows its limits soon, in case of transformations that perform suboptimal on very small chunks, but become more efficient once the number of iterations executed in parallel increases.

Another aspect that makes the purpose of our future work is to extend the types of polyhedral transformations that can be applied. In the current version of VMAD, the patterns are limited to transformations which preserve the structure of the loops inside the nest, such as loop skewing or loop interchange. Still, there is ongoing work on building patterns which support tiling of loops, a very promising transformation that can be easily implemented since the structure of the transformed loop nest is known at compile time. In a nutshell, VMAD requires one pattern for each class of polyhedral transformations. Another interesting strategy is to compose dynamically a new code version by stitching together several code patterns, to build up loop nests whose structures are not preserved by the polyhedral transformations. For generating code versions for which one cannot build a pattern at compile time, such as loop fission, fusion, splitting, one could use JIT, as an alternative to patterns.

There is work in progress for integrating the LLVM JIT in VMAD, to apply optimizations on patterns, prior to JIT-ing and executing them. We consider this a technique worthy to be tested, since patterns abound in expressions and computations of affine functions. Once all values are known at runtime, the JIT can apply simple but efficient optimizations such as constant propagation or strength reduction. The key factor is to evaluate the cost of optimizing, JIT-ing and executing the optimized version of the pattern, in comparison to the execution of the simple, unoptimized pattern.

As our long term goals, we would like to extend VMAD such that, by relaxing the constraints, it can handle more general-purpose codes. For instance, one strategy is to allow the parallel code to violate dependences, as long as the execution remains correct. If some dependences appear to be violated, but the dependent iterations are executed by the same thread, the result is still correct, since intra-thread iterations are executed sequentially.

One of the main constraints we currently impose is that codes exhibit a linear memory accessing behavior, during some phases of execution. For example, a linked list allocated contiguously in memory which is accessed in a loop nest one element after the other makes a good candidate for parallelization, although it is not detected at compile time. Given that the linked list is large enough, even if, let us say, some of its elements are eliminated thus breaking the linearity, it is still efficient to parallelize chunks of the loop accessing the list. Nevertheless, requiring a linear behavior is a strong constraint, which does not fit more general codes, for instance manipulating graphs. We would like to extend our framework, by proposing a new modeling of the programs' behavior. Similar to the current approach, the instrumentation phase would capture the accessed memory locations and values required to be interpolated, however, instead of computing interpolating linear functions, the runtime system would compute the convex hull of the acquired points. By imposing constraints on the density of the convex hulls, more general codes could be efficiently optimized and parallelized, by applying polyhedral transformations.

On another note, from the developers perspective, extending VMAD could be a challenge, since the compile-time code preparation must shake hands with the runtime system. The developer receives little information from the semantics of the `pragma` inserted in the source code, and performs a palette of code manipulations totally transparent to the user, however specific only to the particular `pragma`. To build more general and re-usable passes that prepare the code statically, one could design a set of `pragma`s that describe in more detail and give rich information to the compiler on what actions it must take for preparing the code. As an example, for instrumenting the memory accesses, as presented in this dissertation, the user should insert a pair of `pragma`s. One to *define* the entire mechanism on how the compiler should prepare the code and what are the modules invoked in the runtime system. And a second `pragma`, similar to the one already existing in VMAD, that *uses* the newly defined `pragma` to mark a loop nest. Following a standard API for defining the `pragma`, a unique pass could interpret any new `pragma` and generate the corresponding code.

However, this would transfer the responsibility to the user, which is forced to explicitly include implementation details in the `pragma`. Similar to the *pintools* of PIN, it is a viable technique for extending VMAD for various new types of profiling, instrumentation or optimizations. But it is reserved only to experienced users, unlike the current implementation in which our platform is entirely automatic, once the user specifies the loop nest of interest.

Note that VMAD has already a generic design, as all modules dedicated to different tasks are decoupled and can operate independently, provided the required input is available. From this standpoint, one can envisage future applications of the platform, such as optimizing code for energy efficiency, distributed computing *etc.* Moreover, the generated parallel code could be prepared for hybrid target architectures of CPUs and GPUs.

Bibliography

[1] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 26–37, New York, NY, USA, 2006. ACM.

[2] Randy Allen and Ken Kennedy. Automatic translation of fortran programs to vector form. *ACM Trans. Program. Lang. Syst.*, 9(4):491–542, October 1987.

[3] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. Petabricks: a language and compiler for algorithmic choice. In *PLDI '09*, pages 38–49. ACM, 2009.

[4] Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 168–179, New York, NY, USA, 2001. ACM.

[5] Vishal Aslot, Max J. Domeika, Rudolf Eigenmann, Greg Gaertner, Wesley B. Jones, and Bodo Parady. Specomp: A new benchmark suite for measuring parallel computer performance. In *WOMPAT '01*, pages 1–10. Springer-Verlag, 2001.

[6] Moshe (Maury) Bach, Mark Charney, Robert Cohn, Elena Demikhovsky, Tevi Devor, Kim Hazelwood, Aamer Jaleel, Chi-Keung Luk, Gail Lyons, Harish Patil, and Ady Tal. Analyzing parallel programs with pin. *Computer*, 43(3):34–41, March 2010.

[7] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 1–12, New York, NY, USA, 2000. ACM.

[8] Utpal K. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[9] C. Bastoul. Generating loops for scanning polyhedra. Technical Report 2002/23, PRiSM, Versailles University, 2002. Related to the CLooG tool.

[10] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam. Putting polyhedral transformations to work. In *LCPC'16 Intl. Workshop on Languages and Compilers for Parallel Computers, LNCS 2958*, pages 209–225, College Station, October 2003.

[11] Cédric Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, pages 7–16, Juan-les-Pins, France, September 2004.

[12] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28:135–151, 2001.

[13] Bill Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David Padua, Paul Petersen, Bill Pottenger, Lawrence Rauchwerger, Peng Tu, and Stephen Weatherford. Polaris: The next generation in parallelizing compilers. In *Workshop on Languages and Compilers for Parallel Computing*, pages 10–1. Springer-Verlag, Berlin/Heidelberg, 1994.

[14] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th international conference on Compiler construction*, CC'08/ETAPS'08, pages 132–146, Berlin, Heidelberg, 2008. Springer-Verlag.

[15] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI, 2008.

[16] Uday Kumar Reddy Bondhugula. *Effective automatic parallelization and locality optimization using the polyhedral model*. PhD thesis, Ohio State University, Columbus, OH, USA, 2008. AAI3325799.

[17] Gary Brooks, Gilbert J. Hansen, and Steve Simmons. A new approach to debugging optimized code. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI, 1992.

[18] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *CGO '03: Proc. of the Int. Symp. on Code Generation and Optimization*, 2003.

[19] Derek Bruening, Srikrishna Devabhaktuni, and Saman Amarasinghe. Softspec: Software-based speculative parallelism. In *ACM Workshop on Feedback-Directed and Dynamic Optimization*, Monterey, California, Dec 2000.

[20] Misha Brukman, Brad Jones, Nate Begeman, and Chris Lattner. Extending LLVM: Adding instructions, intrinsics, types, etc. `http://llvm.org/docs/ExtendingLLVM.html`.

[21] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *Int. J. High Perform. Comput. Appl.*, 14(4), 2000.

[22] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5):46–58, September 2008.

[23] John Cavazos and Michael F. P. O'Boyle. Method-specific dynamic compilation using logistic regression. *SIGPLAN Not.*, 41(10):229–240, October 2006.

[24] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 44–54, Washington, DC, USA, 2009. IEEE Computer Society.

[25] Ding Kai Chen, Josep Torrellas, and Pen Chung Yew. An efficient algorithm for the run-time parallelization of doacross loops. In *Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, Supercomputing '94, pages 518–527, New York, NY, USA, 1994. ACM.

[26] Xuan Chen and Shun Long. Adaptive multi-versioning for OpenMP parallelization via machine learning. In *15th Int. Conf. on Parallel and Distributed Systems (ICPADS)*, 2009.

[27] T Chilimbi and M Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI, 2002.

[28] Marcelo Cintra and Diego R. Llanos. Toward efficient and robust software speculative parallelization on multiprocessors. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '03, pages 13–24, New York, NY, USA, 2003. ACM.

[29] A C language family frontend for LLVM. `http://clang.llvm.org`.

[30] Philippe Clauss. Counting solutions to linear and nonlinear constraints through ehrhart polynomials: applications to analyze and transform scientific programs. In *Proceedings of the 10th international conference on Supercomputing*, ICS '96, pages 278–285, New York, NY, USA, 1996. ACM.

[31] Albert Cohen, Sylvain Girbal, and Olivier Temam. A polyhedral approach to ease the composition of program transformations. In *in: Euro-Par'04, no. 3149 in LNCS*, pages 292–303. Springer-Verlag, 2004.

[32] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.

[33] Francis H. Dang, Hao Yu, and Lawrence Rauchwerger. The r-lrpd test: Speculative parallelization of partially parallel loops. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, IPDPS '02, pages 318–, Washington, DC, USA, 2002. IEEE Computer Society.

[34] Alain Darte, Yves Robert, and Frederic Vivien. *Scheduling and Automatic Parallelization*. Birkhauser Boston, 1st edition, 2000.

[35] Laurent Daverio, Corinne Ancourt, Fabien Coelho, Stéphanie Even, Serge Guelton, François Irigoin, Pierre Jouvelot, Ronan Keryell, and Frédérique Silber-Chaussumier. PIPS – An Interprocedural, Extensible, Source-to-Source Compiler Infrastructure for Code Transformations and Instrumentations. Tutorial at PPoPP, Bengalore, India, January 2010; Tutorial at CGO, Chamonix, France, April 2011. `http://pips4u.org/doc/tutorial/tutorial-no-animations.pdf` presented by François Irigoin, Serge Guelton, Ronan Keryell and Frédérique Silber-Chaussumier.

[36] http://www.ice.rwth-aachen.de/research/tools-projects/entry/detail/dspstone/.

[37] Zhao-Hui Du, Chu-Cheow Lim, Xiao-Feng Li, Chen Yang, Qingyu Zhao, and Tin-Fook Ngai. A cost-driven compilation framework for speculative parallelization of sequential programs. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, PLDI '04, pages 71–81, New York, NY, USA, 2004. ACM.

[38] Bruno Dufour, Barbara G Ryder, and Gary Sevitsky. Blended analysis for performance understanding of framework-based applications. In *Int. Symposium on Software Testing and Analysis, ISSTA'07*. ACM, 2007.

[39] Andrew Edwards, Amitabh Srivastava, and Hoi Vo. Vulcan binary transformation in a distributed environment. Technical report, Microsoft Research, 2001.

[40] Non-Hermitian Eigenvalue Problem Collection . `http://math.nist.gov/MatrixMarket/data/NEP`.

[41] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988.

[42] P. Feautrier. Dataflow analysis of scalar and array references. *Int. J. of Parallel Programming*, 20(1):23–53, 1991.

[43] Paul Feautrier. Array expansion. In *In ACM Int. Conf. on Supercomputing*, pages 429–441, 1988.

[44] Paul Feautrier. Some efficient solutions to the affine scheduling problem, part 1 : one dimensional time. *International Journal of Parallel Programming*, 21(5):313–348, 1992.

[45] Paul Feautrier. Some efficient solutions to the affine scheduling problem, part 2 : multidimensional time. *International Journal of Parallel Programming*, 21(6), 1992.

[46] Paul Feautrier. Automatic parallelization in the polytope model. In *Laboratoire PRiSM, Université de Versailles St-Quentin en Yvelines, 45, avenue des États-Unis, F-78035 Versailles Cedex*, pages 79–103. Springer-Verlag, 1996.

[47] Eric Jay Feigin. A case for automatic run-time code optimization, 1999.

[48] Min Feng, Rajiv Gupta, and Yi Hu. Spicec: scalable parallelism via implicit copying and explicit commit. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 69–80, New York, NY, USA, 2011. ACM.

[49] Grigori Fursin, Albert Cohen, Michael O'Boyle, and Olivier Temam. A practical method for quickly evaluating program optimizations. In *In Proceedings of the International Conference on High Performance Embedded Architectures and Compilers (HiPEAC 2005*, pages 29–46. Springer Verlag, 2005.

[50] Grigori Fursin, Cupertino Miranda, Sebastian Pop, Albert Cohen, and Olivier Temam. Practical Run-time Adaptation with Procedure Cloning to Enable Continuous Collective Compilation. In *GCC Developers' Summit*, 2007.

[51] Grigori Fursin and Olivier Temam. Collective optimization: A practical collaborative approach. *ACM Trans. Archit. Code Optim.*, 7, Dec. 2010.

[52] Xiaofeng Gao, M. Laurenzano, B. Simon, and A. Snavely. Reducing overheads for acquiring dynamic memory traces. In *Workload Characterization Symposium*, 2005.

[53] The GNU Compiler Collection. `http://gcc.gnu.org`.

[54] GOMP — An OpenMP implementation for GCC - GNU Project. `http://gcc.gnu.org/projects/gomp`.

[55] Tobias Grosser, Hongbin Zheng, Raghesh A, Andreas Simbürger, Armin Grösslinger, and Louis-Noël Pouchet. Polly - polyhedral optimization in llvm. In *First International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*, Chamonix, France, April 2011.

[56] Matthias Hauswirth and Trishul M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *11th Int. Conf. on Architectural support for programming languages and operating systems*, ASPLOS-XI. ACM, 2004.

[57] Matthieu Herrmann. *Machine virtuelle d'analyse et d'optimisation dynamique de programmes*. Master thesis, Université de Strasbourg, June 2010.

[58] Martin Hirzel and Trishul Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling. In *4th ACM Workshop on FeedbackDirected and Dynamic Optimization FDDO4*, 2001.

[59] Yuanjie Huang, Liang Peng, Chengyong Wu, Yuriy Kashnikov, Jörn Rennecke, and Grigori Fursin. Transforming GCC into a research-friendly environment: plugins for optimization tuning and reordering, function cloning and program instrumentation. In *2nd Int. Workshop on GCC Research Opportunities (GROW'10)*, Pisa Italy, 2010. Google Summer of Code'09.

[60] Interactive Compilation Interface. `http://ctuning.org/ici`.

[61] Clara Jaramillo, Rajiv Gupta, and Mary Lou Soffa. Fulldoc: A full reporting debugger for optimized code. In *7th International Symposium on Static Analysis*, SAS '00. Springer-Verlag, 2000.

[62] Alexandra Jimborean, Matthieu Herrmann, Vincent Loechner, and Philippe Clauss. Vmad: A virtual machine for advanced dynamic analysis of programs. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '11, pages 125–126, Washington, DC, USA, 2011. IEEE Computer Society.

[63] Troy A. Johnson, Rudolf Eigenmann, and T. N. Vijaykumar. Speculative thread decomposition through empirical optimization. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '07, pages 205–214, New York, NY, USA, 2007. ACM.

[64] A. Ketterlin and P. Clauss. Prediction and trace compression of data access addresses through nested loop recognition. In *Proc. of the sixth annual IEEE/ACM Int. Symp. on Code generation and optimization*, pages 94–103, Boston, MA, USA, 2008.

[65] Minhaj Ahmad Khan, H.-P. Charles, and D. Barthou. Improving performance of optimized kernels through fast instantiations of templates. *Concurr. Comput. : Pract. Exper.*, 21(1):59–70, January 2009.

[66] Hanjun Kim, Nick P. Johnson, Jae W. Lee, Scott A. Mahlke, and David I. August. Automatic speculative doall for clusters. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 94–103, New York, NY, USA, 2012. ACM.

[67] Jungsoo Kim, Sungjoo Yoo, and Chong-Min Kyung. Program phase and runtime distribution-aware online dvfs for combined vdd/vbb scaling. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 417–422, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[68] Minjang Kim, Hyesoon Kim, and Chi-Keung Luk. Sd3: A scalable approach to dynamic data-dependence profiling. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 535–546, Washington, DC, USA, 2010. IEEE Computer Society.

[69] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the java hotspot client compiler for java 6. *ACM Trans. Archit. Code Optim.*, 5, May 2008.

[70] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. *Commun. ACM*, 52(9):89–97, September 2009.

[71] Naveen Kumar, Bruce Childers, and Mary Lou Soffa. Transparent debugging of dynamically optimized code. In *Int. Symp. on Code Generation and Optimization*, CGO '09. IEEE Computer Society, 2009.

[72] J. R. Larus. Loop-level parallelism in numeric and symbolic programs. *IEEE Trans. Parallel Distrib. Syst.*, 4:812–826, July 1993.

[73] Chris Lattner and Vikram Adve. LLVM language reference manual. `http://llvm.org/docs/LangRef.html`.

[74] M. Laurenzano, M. Tikir, L. Carrington, and A. Snavely. PEBIL: Efficient static binary instrumentation for linux. In *ISPASS-2010: IEEE Int. Symp. on Performance Analysis of Systems and Software*, 2010.

[75] Xiao-Feng Li, Chen Yang, Zhao-Hui Du, and Tin-fook Ngai. Exploiting thread-level speculative parallelism with software value prediction. In *Proceedings of the 10th Asia-Pacific conference on Advances in Computer Systems Architecture*, ACSAC'05, pages 367–388, Berlin, Heidelberg, 2005. Springer-Verlag.

[76] Wei Liu, James Tuck, Luis Ceze, Wonsun Ahn, Karin Strauss, Jose Renau, and Josep Torrellas. POSH: a TLS compiler that exploits program structure. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 158–167, New York, NY, USA, 2006. ACM.

[77] The LLVM compiler infrastructure. `http://llvm.org`.

[78] Register allocation in the LLVM compiler. `http://llvm.org/docs/CodeGenerator.html#regalloc`.

[79] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[80] Lianjie Luo, Yang Chen, Chengyong Wu, Shun Long, and Grigori Fursin. Finding representative sets of optimizations for adaptive multiversioning applications. In *International Workshop on Statistical and Machine learning approaches to ARchitectures and compilaTion*, Paphos Chypre, 2009.

[81] Yangchun Luo, Venkatesan Packirisamy, Wei-Chung Hsu, Antonia Zhai, Nikhil Mungre, and Ankit Tarkas. Dynamic performance tuning for speculative threads. *SIGARCH Comput. Archit. News*, 37(3):462–473, June 2009.

[82] Jonathan Mak and Alan Mycroft. Limits of parallelism using dynamic dependency graphs. In *Proceedings of the Seventh International Workshop on Dynamic Analysis*, WODA '09, pages 42–48, New York, NY, USA, 2009. ACM.

[83] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. Literace: effective sampling for lightweight data-race detection. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI, 2009.

[84] Jason Mars and Robert Hundt. Scenario based optimization: A framework for statically enabling online optimizations. In *CGO '09*, pages 169–179. IEEE Computer Society, 2009.

[85] Mojtaba Mehrara, Jeff Hao, Po-Chun Hsu, and Scott Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. *SIGPLAN Not.*, 44(6):166–176, June 2009.

[86] Michael Philippsen, Nikolai Tillmann, and Daniel Brinkers. Double inspection for run-time loop parallelization. In *Proceedings of the 24th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2011)*, 2011.

[87] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI '07: Proc. of ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 2007.

[88] François Noël, Luke Hornof, Charles Consel, and Julia L. Lawall. Automatic, template-based run-time specialization: Implementation and experimental study. In *In International Conference on Computer Languages*, pages 132–142. IEEE Computer Society Press, 1998.

[89] Cosmin E. Oancea and Alan Mycroft. Languages and compilers for parallel computing. In José Nelson Amaral, editor, *LCPC*, chapter Set-Congruence Dynamic Analysis for Thread-Level Speculation (TLS), pages 156–171. Springer-Verlag, Berlin, Heidelberg, 2008.

[90] Cosmin E. Oancea, Alan Mycroft, and Tim Harris. A lightweight in-place implementation for software thread-level speculation. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, SPAA '09, pages 223–232, New York, NY, USA, 2009. ACM.

[91] OmpSCR: OpenMP source code repository. `http://sourceforge.net/projects/ompscr`.

[92] Eunjung Park, Sameer Kulkarni, and John Cavazos. An evaluation of different modeling techniques for iterative compilation. In *Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems*, CASES '11, pages 65–74, New York, NY, USA, 2011. ACM.

[93] Devang Patel and Lawrence Rauchwerger. Implementation issues of loop-level speculative run-time parallelization. In Stefan Jähnichen, editor, *Compiler Construction*, volume 1575 of *Lecture Notes in Computer Science*, pages 1–99. Springer Berlin / Heidelberg, 1999.

[94] Pointer-intensive benchmark suite. `http://pages.cs.wisc.edu/~austin/ptr-dist.html`.

[95] Polybenchs 1.0. http://www-rocq.inria.fr/pouchet/software/polybenchs., 2010.

[96] Polylib. `http://icps.u-strasbg.fr/PolyLib`.

[97] Louis-Noël Pouchet. FM: the Fourier-Motzkin library. `http://www.cse.ohio-state.edu/ pouchet/software/fm`.

[98] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, P. Sadayappan, and Nicolas Vasilache. Loop transformations: convexity, pruning and optimization. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 549–562, New York, NY, USA, 2011. ACM.

[99] PPL: The parma polyhedra library. `http://www.cs.unipr.it/ppl/`.

[100] Manohar K. Prabhu and Kunle Olukotun. Using thread-level speculation to simplify manual parallelization. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '03, pages 1–12, New York, NY, USA, 2003. ACM.

[101] Benoit Pradelle. *Static and dynamic methods of polyhedral compilation for an efficient execution in multicore environments*. PhD thesis, University of Strasbourg, Strasbourg, France, 2011.

[102] Benoit Pradelle, Philippe Clauss, and Vincent Loechner. Adaptive Runtime Selection of Parallel Schedules in the Polytope Model. In *High Performance Computing Symposium*, Boston, United States, April 2011. ACM/SIGSIM.

[103] Carlos García Quiñones, Carlos Madriles, Jesús Sánchez, Pedro Marcuello, Antonio González, and Dean M. Tullsen. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 269–279, New York, NY, USA, 2005. ACM.

[104] Arun Raman, Hanjun Kim, Thomas R. Mason, Thomas B. Jablin, and David I. August. Speculative parallelization using software multi-threaded transactions. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 65–76, New York, NY, USA, 2010. ACM.

[105] Easwaran Raman, Neil Va hharajani, Ram Rangan, and David I. August. Spice: speculative parallel iteration chunk execution. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, pages 175–184, New York, NY, USA, 2008. ACM.

[106] Lawrence Rauchwerger and David Padua. The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, PLDI '95, pages 218–232, New York, NY, USA, 1995. ACM.

[107] Rosetta Codes. `http://rosettacode.org/wiki/Rosetta_Code`.

[108] Gabe Rudy, Malik Murtaza Khan, Mary Hall, Chun Chen, and Jacqueline Chame. A programming language interface to describe transformations and code generation. In *Proceedings of the 23rd international conference on Languages and compilers for parallel computing*, LCPC'10, pages 136–150, Berlin, Heidelberg, 2011. Springer-Verlag.

[109] Alexander Schrijver. *Theory of linear and integer programming.* John Wiley & Sons, Inc., New York, NY, USA, 1986.

[110] Timothy Sherwood, Erez Perelman, Greg Hamerly, Suleyman Sair, and Brad Calder. Discovering and exploiting program phases. *IEEE Micro*, 23(6):84–93, November 2003.

[111] Frederick Smith, Dan Grossman, Greg Morrisett, Luke Hornof, and Trevor Jim. Compiling for template-based run-time code generation. *J. Funct. Program.*, 13(3):677–708, May 2003.

[112] SPEC CPU2006. `http://www.spec.org/cpu2006`.

[113] Vasileios Spiliopoulos, Stefanos Kaxiras, and Georgios Keramidas. Green governors : A framework for continuously adaptive dvfs. In *Proc. International Green Computing Conference and Workshops : IGCC 2011*, pages 1–8. IEEE, 2011.

[114] J. Greggory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. A scalable approach to thread-level speculation. In *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, pages 1–12, New York, NY, USA, 2000. ACM.

[115] Martin Süsskraut, Stefan Weigert, Ute Schiffel, Thomas Knauth, Martin Nowack, Diogo Becker Brum, and Christof Fetzer. Speculation for parallelizing runtime

checks. In *Proceedings of the 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, SSS '09, pages 698–710, Berlin, Heidelberg, 2009. Springer-Verlag.

[116] Nathan Thomas, Gabriel Tanase, Olga Tkachyshyn, Jack Perdue, Nancy M. Amato, and Lawrence Rauchwerger. A framework for adaptive algorithm selection in stapl. In *PPoPP '05*, pages 277–288. ACM, 2005.

[117] Chen Tian, Min Feng, and Rajiv Gupta. Speculative parallelization using state separation and multiple value prediction. In *Proceedings of the 2010 international symposium on Memory management*, ISMM '10, pages 63–72, New York, NY, USA, 2010. ACM.

[118] Chen Tian, Min Feng, and Rajiv Gupta. Supporting speculative parallelization in the presence of dynamic data structures. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 62–73, New York, NY, USA, 2010. ACM.

[119] Chen Tian, Min Feng, Vijay Nagarajan, and Rajiv Gupta. Copy or discard execution model for speculative parallelization on multicores. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 330–341, Washington, DC, USA, 2008. IEEE Computer Society.

[120] Chen Tian, Min Feng, Vijay Nagarajan, and Rajiv Gupta. Speculative parallelization of sequential loops on multicores. *Int. J. Parallel Program.*, 37(5):508–535, October 2009.

[121] Chen Tian, Changhui Lin, Min Feng, and Rajiv Gupta. Enhanced speculative parallelization via incremental recovery. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 189–200, New York, NY, USA, 2011. ACM.

[122] Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael F.P. O'Boyle. Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 177–187, New York, NY, USA, 2009. ACM.

[123] Abhishek Udupa, Kaushik Rajan, and William Thies. Alter: exploiting breakable dependences for parallelization. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 480–491, New York, NY, USA, 2011. ACM.

[124] A. van Deursen, P. Klint, and F. Tip. Origin tracking. *J. Symb. Comput.*, 15, May 1993.

[125] Hans Vandierendonck, Sean Rul, and Koen De Bosschere. The paralax infrastructure: automatic parallelization with a helping hand. In *Proceedings of the 19th*

international conference on Parallel architectures and compilation techniques, PACT '10, pages 389–400, New York, NY, USA, 2010. ACM.

[126] Nicolas T. Vasilache. *Scalable Program Optimization Techniques in the Polyhedral Model*. PhD thesis, Université Paris Sud XI, Orsay, September 2007.

[127] Sven Verdoolaege. ISL: An integer set library for the polyhedral model. In *Lecture Notes in Computer Science*, 2010.

[128] Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe. Counting integer points in parametric polytopes using barvinok's rational functions. *Algorithmica*, 48(1):37–66, March 2007.

[129] Christoph von Praun, Rajesh Bordawekar, and Calin Cascaval. Modeling optimistic concurrency using quantitative dependence analysis. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 185–196, New York, NY, USA, 2008. ACM.

[130] Michael J. Voss and Rudolf Eigemann. High-level adaptive program optimization with ADAPT. In *PPoPP '01*, pages 93–102. ACM, 2001.

[131] R. P. Weicker and J. L. Henning. Subroutine profiling results for the CPU2006 benchmarks. *SIGARCH Comput. Archit. News*, 35(1), 2007.

[132] Robert Wilson, Robert French, Christopher Wilson, Saman Amarasinghe, Jennifer Anderson, Steve Tjiang, Shih Liao, Chau Tseng, Mary Hall, Monica Lam, and John Hennessy. The suif compiler system: a parallelizing and optimizing research compiler. Technical report, Stanford University, Stanford, CA, USA, 1994.

[133] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. Parallel Distrib. Syst.*, 2(4):452–471, October 1991.

[134] Peng Wu, Arun Kejariwal, and Călin Caşcaval. Languages and compilers for parallel computing. In José Nelson Amaral, editor, *LCPC*, chapter Compiler-Driven Dependence Profiling to Guide Program Parallelization, pages 232–248. Springer-Verlag, Berlin, Heidelberg, 2008.

[135] Jun Yao, Hajime Shimada, Yasuhiko Nakashima, Shin-ichiro Mori, and Shinji Tomita. Program phase detection based dynamic control mechanisms for pipeline stage unification adoption. In *Proceedings of the 6th international symposium on high-performance computing and 1st international conference on Advanced low power systems*, ISHPC'05/ALPS'06, pages 494–507, Berlin, Heidelberg, 2008. Springer-Verlag.

[136] Xiangyu Zhang, Armand Navabi, and Suresh Jagannathan. Alchemist: A transparent dependence distance profiling infrastructure. In *Proceedings of the 7th*

annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '09, pages 47–58, Washington, DC, USA, 2009. IEEE Computer Society.

[137] Hongtao Zhong, Mojtaba Mehrara, Steven A. Lieberman, and Scott A. Mahlke. Uncovering hidden loop level parallelism in sequential applications. In *HPCA*, pages 290–301, 2008.

[138] Chuan-Qi Zhu and Pen-Chung Yew. A scheme to enforce data dependence on large multiprocessor systems. *IEEE Trans. Softw. Eng.*, 13:726–739, June 1987.

[139] Lenore Zuck, Amir Pnueli, Benjamin Goldberg, Clark Barrett, Yi Fang, and Ying Hu. Translation and run-time validation of loop transformations. *Form. Methods Syst. Des.*, 27(3):335–360, November 2005.