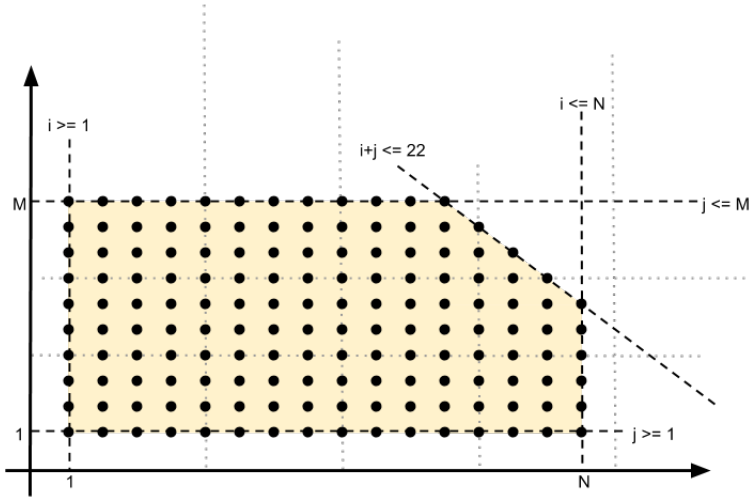
Universidad de Buenos Aires

Enhancing speculative code parallelization in VMAD with new patterns

Author	Juan Manuel Martinez Caamaño	*jmmartinez@dc.uba.ar*
Advisor	Philippe Clauss	*philippe.clauss@inria.fr*
Co-Advisor	Diego Garbervetsky	*diegog@dc.uba.ar*



Abstract

Los procesadores con múltiples núcleos de hoy en día requieren nuevas estrategias para sacar ventaja del hardware subyacente. Dos aspectos que afectan drásticamente el desempeño son el grado de *paralelismo* y la *localía de datos*. Una forma de mejorar estos aspectos es a través del uso de *compiladores* que puedan, automáticamente, paralelizar el código con muy poca intervención del programador.

Dado que los ciclos juegan un rol central en el tiempo de ejecución total de las aplicaciones, los compiladores prestan particular atención a ellos. Esta tesis se enfoca en el problema de paralelizar automáticamente ciclos. Un modelo matemático llamado *modelo del poliedro* cubre la teoría detrás del análisis y transformación de ciclos. Sin embargo, este modelo, originalmente, esta limitado a ciclos que pueden ser precisamente analizados en tiempo de ejecución, no puede tratar con ciclos con accesos a memoria a través de punteros o indirecciones.

VMAD es un framework que permite instrumentación en tiempo de ejecución y paralelización dinámica. Adaptando el modelo del poliedro a una técnica llamada paralelización especulativa, este framework puede tratar con ciclos que no pueden ser paralelizados utilizando las técnicas tradicionales.

Esta tesis continua el trabajo hecho en VMAD de la siguiente forma:

1. Extendiendo la cantidad de transformaciones soportadas por VMAD dando soporte para *tiling*, una transformación que mejora la localía de datos.

2. Proponemos una paralelización de ciclos optimizada, para ciclos que pueden ser paralelizados sin aplicar ningún tipo de transformación.

3. Presentamos una optimización para reducir la penalización impuesta por utilizar paralelización especulativa.

Evaluamos empíricamente la efectividad de las extensiones usando una serie de benchmarks bien conocidos.

Abstract

Today's multi-core processors impose new strategies to take advantage of the underlying hardware. Two important aspects that drastically affects the performance are the degree of *parallelism* and *data locality*. One way to improve these aspects is through the use of *parallelizing compilers*, which can automatically parallelize code with very little programmer intervention.

Since loops play an central role in the overall execution time of every application, compiler play particular attention to them. This thesis focuses on the problem of automatically parallelizing loops. A mathematical framework called the *polytope model* cover the theory behind loop analysis and transformation. Nevertheless, this theory is originally limited to loops which can be precisely analyzed at compile time, it cannot handle loops with memory accesses through pointers or indirections.

VMAD is a framework which enables runtime instrumentation and dynamic parallelization. By adapting the polytope model to a technique called *speculative parallelization*, this framework is able to handle loops which cannot be parallelized using the traditional techniques.

This thesis continues the work done in VMAD by:

1. Proposing an optimized loop parallelization, when the loop can be parallelized without applying any transformation.

2. Extending the transformation scope supported in VMAD by enabling *tiling*, a transformation which improves data locality.

3. Introducing an optimization to reduce the overhead imposed by speculative parallelization.

We empirically evaluate the effectiveness of the extensions using a series of well known benchmarks.

Acknowledgments

I would like to express my gratitude with all the people who helped me during the Computer Science degree and the development of my thesis.

I want to thank Philippe, Vincent and Aravind, for their guidance and help to finish successfully my internship at Strasbourg, which concluded in this work.

I would like to thank Alexandra Jimborean, former PhD student of Philippe Clauss and main author of VMAD. This thesis would not have been possible without her contributions.

Thanks to Diego, who guided and helped me through this past year, and showed me a bigger picture of computer science.

For their support during all these years, I want to thank my mom, dad and brother.

To my friends, Ari, Nico, Lean, March, Ivan, Mati, Fede, and GAP, with whom I shared most of the courses. They made the degree shorter and more enjoyable.

Contents

List of Figures

Chapter 1

Introduction

This work studies the problem of automatically parallelizing programs, in particular, focusing on loops. The approach taken in this thesis is called *speculative parallelization*. The techniques introduced in this work rely on the *polytope model*, a mathematical framework for reasoning about loop nests. This thesis continues previous work on a speculative parallelization framework called VMAD [18, 21] by extending the code optimizations available in it.

1.1 Context of the work

The emergence of the multi-core processors imposes new strategies for reaching good software performance and exploiting advantageously the underlying hardware. The key to success is now radically related to *parallelism*, where applications execute in such a way that many computations are performed simultaneously on several processor cores, and *data locality*, to reduce the penalty introduced by the memory accesses and communication between computing units.

There are several possible options to accomplish this: programs can be written by explicitly describing what can be run in parallel and what cannot, a compiler can synthesize parallel code from a formal specification, 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 optimizations. Nevertheless, although each option has been intensely studied, they all have some inherent limitations.

Even if many parallel programming languages and extensions are available, such as OpenMP, MPI, Cilk, TBB, HMPP, OpenCL or CUDA, parallel programming is in general difficult. The programmer is required to handle complex issues: selecting a convenient algorithm for parallelization, analyzing the dependences, ensuring

correct semantics and being aware of the underlying hardware characteristics.

Algorithmic skeletons exploit some common patterns in parallel programming, to hide some of the complexity of parallel programming. The provide some parallel primitives which can be combined to generate more complex constructs. Nevertheless, this algorithmic skeletons are restricted to the type of parallel model they try to express –for example data parallelism. Also, the programmer is forced to fit its solution into the available skeletons.

An other technique to take advantage of the multi-core processors is to automatically generate parallel code from a specification. In this approach, the programmer is released from the task of being aware of the underlying hardware. An example of this is line of work is FXML [41], a formal language for expressing concurrency and data precedence constraints. The FXML specification is intended to be transformed automatically, until actually executable code is obtained. This specification can be directly written by designers of the application, or be used as a semantic framework for other languages.

To aid the programmer in delivering parallel code, developing 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 [40], Polaris [3], PIPS [10] and Pluto [5], 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* [14, 15]. Chapter 2 presents a brief introduction about the polytope model.

However, its applicability is limited to array-based applications exhibiting explicit linearity of their loop bounds and array accesses. Loops exhibiting complex exit conditions and memory accesses through pointers or indirections cannot be handled at compile-time using these techniques, since crucial information can only be known at runtime. 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 some code regions on-the-fly. 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 cancelling 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 overhead 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) [37, 26, 32, 34, 39, 33, 6, 42, 23, 22]. 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.

Loop tiling[1, 5] is a key loop optimization for improving data locality. Tiling partitions a loop iteration space into smaller blocks, to take advantage of reuse in several directions.

The contributions of this work focus on three axis:

- Adding a new transformation which requires a reduced amount of verification and initialization code for each iteration. Chapters 4 covers this topics.
- Adding support for new code transformations, such as tiling. Chapter 5 focuses on this topic.
- To provide more information for the compile time code generation, a new static analysis is introduced. This information is used to avoid unnecessary verifications, during the execution of an optimized version of the code. This is covered in Chapter 6.

The work described in this thesis is implemented as a set of extensions of the VMAD framework. This framework adapts the polytope model to speculative parallelization.

The VMAD framework consists on several code transformations performed at compile time, implemented as compiler extensions, and a runtime system.

- The *compiler extensions* are in charge of performing static analysis and code generation.

- The *runtime system* gathers the data obtained from instrumentation, performs dependence analysis, selects which transformation is the most suitable to run, and orchestrates the execution of code versions thanks to a dedicated chunking system.

This framework it's not restricted by the type of code being optimized. Nevertheless, since it adapts the polytope model to speculative parallelization, it's restricted to loops exhibiting some phases of polyhedral behavior.

An explanation on the approach to speculative parallelization of VMAD is presented in chapter 3.

This thesis extends both components to allow new optimizations and to make use of the new static analysis results.

Discussion over the results and conclusions are presented in chapters 7 and 8.

Part I

Preliminars

Chapter 2

The polytope model

The polytope model provides a mathematical framework for loop analysis and optimization. A polytope is a geometric object with flat sides, which exists in any number of dimensions. In this case, we are interested only on convex polytopes. This model provides an homogeneous representation of the loop information necessary for performing precise dependence analysis.

An exhaustive presentation of applications of the polytope model in program optimizations are given by Feautrier in multiple works [12, 13, 14, 15] and has been addressed in many others works dedicated to compile-time [4, 5, 17] or runtime loop optimizations [31]. For a throughout presentation of the mathematical apparatus building the underlying background of the polytope model, the reader is referred to the monograph of Schrijver [36]. In what follows we present only an overview.

The polytope model can only be applied for loop nests where:

- The loop bounds are affine functions[1] of the surrounding loop indices and parameters.
- Each test predicate is an affine function of the surrounding loop indices and parameters.
- The data structures in the nest are arrays and scalars of any type.
- The array subscripts are affine functions of the surrounding loop counters and parameters.

2.1 Definitions

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

[1] See next page.

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

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

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 $\vec{x} \in K^n$ of the form:

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

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

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

$$\vec{a} \cdot \vec{x} \geq \text{b} \quad \text{and} \quad \vec{a} \cdot \vec{x} \leq \text{b}$$

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

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 $P \subset K^n$ can be expressed as a set of m affine constraints in $A \in K^{m \times n}$ and $\vec{b} \in K^m$:

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

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 $A \in K^{m \times n}$, a matrix of symbolic coefficients $B \in K^{m \times p}$, where p is the dimension of the vector of parameters $|\vec{p}| = p$ and a vector of constants $\vec{b} \in K^m$ as:

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

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

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.

```
for(i = 1; i ≤ N; ++i) {
    for(j = 1; j ≤ N; ++j) {
        S1: A[i][j] = A[i-1][j] + [i][j-1]
    }
}
```

Figure 2.1: Loop nest example for the iteration domain.

2.2 Representing loops in the polytope model

The representation of loop nests in the polytope model consists of three parts for each statement of the loop: the iteration domain, the access functions for each memory access and the schedule. These parts are explained in the following sections.

2.2.1 The iteration domain

Lets consider the loop nest in figure 2.1:

Notice the statement labeled as S_1. Even if S_1 is only a single statement, it has several statement instances along the life time of the loop, namely $S_{1_{(1,1)}}$, $S_{1_{(1,2)}}$, ..., $S_{1_{(N,N)}}$. Each statement instance has an iteration vector associated to it. The iteration vector for S_1 is (i, j), the set of all iteration vectors for a statement is called *Domain* or *Index set*. We can define the previous iteration domain as:

$$D_{S_1} = \{(i, j) \in Z^2 | 1 \leq i \leq N, 1 \leq j \leq N\}$$

The iteration domain is defined by a set of constraints. If these constraints are affine functions of the surrounding loop indices and parameters, the set of constraints defines a *polytope*. The matricial representation of D_{S_1} is the following:

$$\left(\begin{array}{cc|c} 1 & 0 & -1 \\ -1 & 0 & 10 \\ 0 & 1 & -1 \\ 0 & -1 & 10 \end{array}\right) * \begin{pmatrix} i \\ j \\ \hline 1 \end{pmatrix} \geq 0$$

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 $\mathcal{D}^S$.

2.2.2 Data access function

To model the memory accesses of a statement, a function corresponding to this statement has to be defined. These access functions have to be affine and must depend on loop indices and parameters.

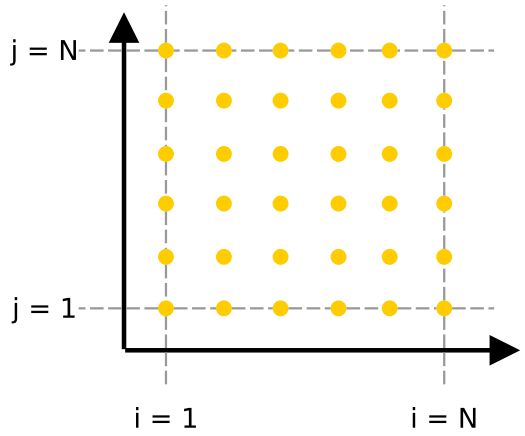


Figure 2.2: Representation of the iteration domain D_{S_1}.

The statement S_1 has one write operation corresponding to $A[i][j] = \ldots$ and two read operations corresponding to $A[i-1][j]$ and $A[i][j-1]$. The access functions corresponding to this memory accesses are:

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

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

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

2.2.3 Statement scheduling

The *iteration domain* and the *data access functions* do not represent the order of execution of each statement. The scheduling function gives a *time stamp* to each statement instance, expressing the order in which a statement instance has to be executed. In general, for a statement S its time stamp is defined by:

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

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

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

The scheduling function also expresses the ordering between statements.
The scheduling function for statement S_1 is:

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

The constants between the iterators allow to express the order between statements at the same loop level. If there was another statement executed after S_1 at the same loop level its scheduling function would be $\theta^{S_2}(i, j) = (0, i, 0, j, 1)$.

Applying the scheduling function $\theta(\vec{x}) = T\vec{x} + \vec{t}$ to the integer points of a iteration domain $\mathcal{D} = \{\vec{x} | A\vec{x} + \vec{b}\}$, is expressed as the polyhedron:

$$\left(\begin{array}{c|c} I & -T \\ \hline 0 & A \end{array}\right) \left(\begin{array}{c} \theta(\vec{x}) \\ \hline \vec{x} \end{array}\right) \begin{array}{c} = \\ \hline \geq \end{array} \left(\begin{array}{c} \vec{t} \\ \hline \vec{b} \end{array}\right)$$

Scheduling matrices

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

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

Canonical form of the scheduling matrices

To make the scheduling matrix more meaningful, Cohen *et al.* [2, 7] propose a normalized representation. The encoding purposed is suitable for expressing compositions of transformations, as it is decomposable in three sub-matrices:

1. *The iteration ordering matrix* $A^S \in \mathcal{M}_{d^S, d^S}(Z)$ representing the iteration vectors;

2. *The matrix of parameters* $\Gamma^S \in \mathcal{M}_{d^S, d_g p+1}(Z)$, where $d_g p$ denotes the number of global parameters;

3. *The statement ordering vector* $\beta \in N^{d^S+1}$, 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: affecting A^S it is possible to define a loop interchange, skewing or loop reversal; altering Γ^S can define shifting transformations; and modifying β^S redefines the execution order of the instructions, equivalent to performing loop fission or loop fusion.

The schedule for the statement S_1 its given by following matrices:

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

The time stamp for S_1 is given by:

$$\left(\begin{array}{cc|c|c} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right) * \left(\begin{array}{c} i \\ j \\ \hline N \\ 1 \end{array}\right) = \begin{pmatrix} 0 \\ i \\ 0 \\ j \\ 0 \end{pmatrix}$$

2.3 Dependency Analysis

To be valid, the loop transformations have to preserve the semantics of the sequential execution, to ensure that the resulting code preserves its correctness.

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, such that $S(\vec{x_S})$ and $R(\vec{x_R})$ access the same memory location and at least one is a write.

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 statements can be executed in arbitrary order.

Note that when the two statements are read operations, there is no dependence since the memory is not modified. However this accesses can be taken into account to improve data locality.

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*.

2.3.1 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* [1], 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 $\text{sig}(\vec{d}) = (\text{sig}(d_1), \text{sig}(d_2), ..., \text{sig}(d_m))$;
- 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.

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.

Schedule validation

Once the dependences between iterations are computed, one can apply polyhedral transformations (schedules) such that no dependence violations occur.

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. This first strictly positive component defines the depth of the loop which carries the dependence. The outermost parallel loop level is given by the first column in the resulting matrix for which no loop is carrying a dependence at this depth.

2.3.2 Example of a transformation

Recall the example of figure 2.1. For a given iteration (i, j), for example, to compute $A[i][j]$ it must read the values produced by iteration $(i-1, j)$ and $(i, j-1)$. This produce the dependences d_1 and d_2 and the dependence matrix d_M.

$$d_1 = \begin{pmatrix} 1 & 0 \end{pmatrix} d_2 = \begin{pmatrix} 0 & 1 \end{pmatrix}$$

$$d_M = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Notice that with this dependences there is no parallel loop. If we apply the transformation $T = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ we obtain:

$$d'_m = d_m * T = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

$$d'_1 = \begin{pmatrix} 1 & 1 \end{pmatrix} d'_2 = \begin{pmatrix} 1 & 0 \end{pmatrix}$$

The first non-null element of each new dependence is positive, so the transformation is valid. Also the outermost loop carries the dependence, thus, the innermost loop can be executed in parallel.

By applying this transformation we obtain a new iteration space with two new iterators, x and y such that $\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} i-j \\ i \end{pmatrix}$.

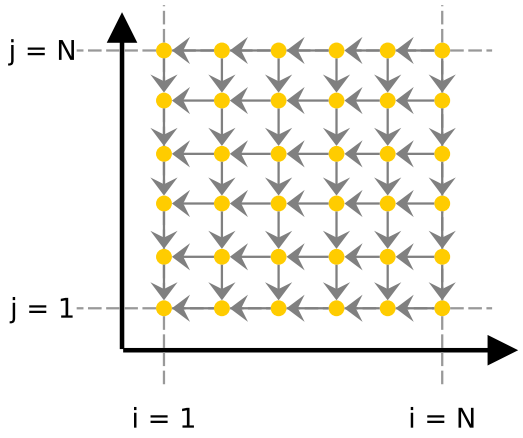


Figure 2.3: Representation of the iteration domain D_{S_1} with its dependences.

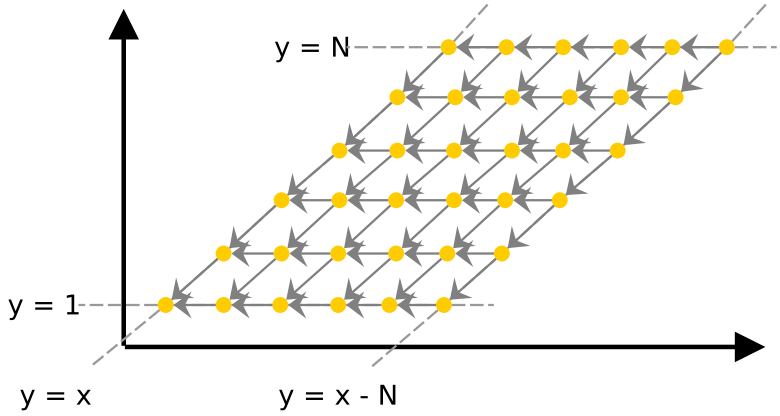


Figure 2.4: Representation of the new iteration domain for S_1 with its dependences.

```
for(x = 1; x ≤ 2N−1; ++x) {
    forall(y = 1; y ≤ min(x,N); ++y) {
        i = x − y
        j = x
        S1: A[i][j] = A[i−1][j] + [i][j−1]
    }
}
```

Figure 2.5: Resulting code after applying the code transformation.

Chapter 3

The VMAD framework

VMAD is a framework for speculative loop parallelization and instrumentation. In the following sections, a detailed overview of the VMAD framework is presented and how the polytope model is adapted for speculative parallelization.

3.1 VMAD Architecture

VMAD is implemented as a static-dynamic framework. We show its architecture in figure 3.2.

The procedure for using VMAD is the following: The programmer annotates the source code using a specific *pragma*, as shown in figure 3.1. This source code is compiled using the *clang* C compiler, producing as output its corresponding *llvm intermediate representation*[24]. In this intermediate representation, the previously annotated loop nests are identified by specific meta-data attached to the instructions of the loop nest.

The front-end can be easily interchanged with an other language front-end which outputs *llvm-ir*, making VMAD available to a wide set of programming languages.

```
#pragma speculative_parallelization
{
    loop 1
        loop 2
            ...
}
```

Figure 3.1: Loop nest tagged for speculative parallelization.

The *static component* of the framework is implemented as a series of compiler

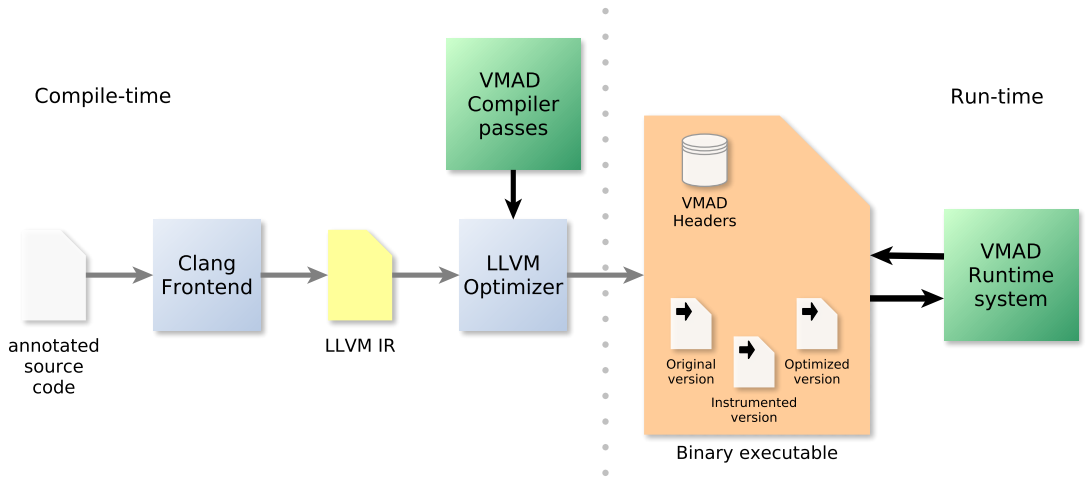


Figure 3.2: Overview of the architecture of the VMAD framework.

extensions. The main purpose of these components is to generate several code versions from the original loop nest:

- One version is dedicated to profiling (See section 3.3). This version includes code to capture dynamic data of the execution of the loop nest through instrumentation.
- Another version, corresponding to the original version of the loop nest.
- Additionally, several code-skeletons matching different kinds of code transformations. In section 3.4 we explain the construction of one of this code-skeletons.

Moreover, the VMAD static component embeds, in the binary executable, information that is statically available. This includes data about the loop nests, such as the loop hierarchy and depth, addresses of the callbacks in the code or values of some parameters. This information is stored in dedicated *Headers*.

The *Runtime System* is in charge of collecting dynamic information during the actual loop execution. Using this information, the runtime system builds interpolating linear functions to describe the behavior of that loop. Afterwards, dependence analysis is performed and an adequate code-version is selected to execute.

3.2 Multi-versioning

With the increasing complexity of the available hardware, including multi-cores and co-processors, multi-versioning has become a classical technique for using efficiently all available resources. It consists in compiling and embedding in the binary different versions of a region of code. It is particularly adequate for periodic instrumentation, adaptive version selection, and dynamic optimization in general. VMAD presents a mechanism for switching dynamically from a version of a code to another, without interrupting the execution of the program.

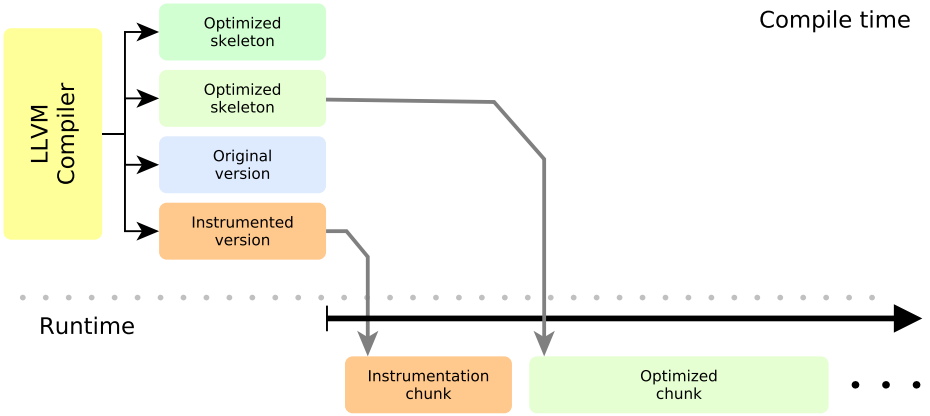


Figure 3.3: Alternating the execution of different versions, during one run of the loop nest.

In VMAD, multi-versioning is employed for generating three types of versions for each loop nest: an original version, an instrumented version and several optimized versions. At the moment of execution of the loop nest, the runtime system will switch between the different code versions to achieve different behaviors.

3.3 Instrumentation of loop nests

Typically, the polytope model relies on loops which can be statically analyzed and in which the data access patterns and loop bounds are affine functions of the enclosing loop indexes.

To be able to handle loops which cannot be fully analyzed at compile time, this model has to be extended.

Non-statically available information is captured by means of an *online profiling phase*, aimed at monitoring and instrumenting the data accesses and the loop

bounds. To limit the runtime overhead of the profiling, a suitable solution is to perform *instrumentation by sampling*. Instrumenting code usually induces a huge overhead, sampling tries to overcome this limitation by instrumenting a small number of iterations of each loop and switching to the non-instrumented version.

Provided that the target loop executes a high number of iterations, the cost of the instrumentation is negligible in practice.

During the instrumentation phase, VMAD monitors:

- Values taken by some scalars.
- The accessed memory locations.
- The number of iterations executed by the sub-loops.

From the values collected during instrumentation, interpolating linear functions are built if possible. Expressing data accesses and loop trip counts[1] as affine functions makes it possible to apply the polytope model dynamically.

3.4 Adapting the polytope model for speculative parallelization

As detailed in the previous chapter, the polytope model can be used for programs in which it is possible to obtain an exact characterization of the data dependences. The static analysis models program executions featuring loops by the aid of a mathematical formulation, particularly linear algebra and integer programming[2]. 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. Applying the traditional methodology at runtime will be inefficient because it is extremely time consuming. For dealing with this situation, VMAD proposes a light-weight system, able to fill-in the missing information at runtime, and perform polyhedral transformations dynamically and automatically to parallelize loops.

With the purpose of generating parallelized code at runtime, VMAD uses a technique known as *code skeletons* [19].

Code skeletons have already a long history of addressing runtime code specialization and simple optimizations. Noël *et al.* [27] 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 of our work for performing more advanced loop optimizations dynamically.

[1] Number of times in a row the loop branch is taken.

[2] A detailed description on how the model is applied in a research compiler is given in [5]

At compile time, several parallel code patterns are prepared, from which different code versions can be dynamically generated. The parallel code patterns are built to support the generation of distinct code versions, by applying various polyhedral transformations. A large set of parallelizing transformations can be implemented at execution-time, by patching some predefined 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.

Since the key contributions of this thesis rely on extending VMAD by adding new code patterns, a detailed description on how these patterns are constructed is given in this chapter.

For pedagogical purposes, we present the example in figure 3.4, to introduce how a pattern is built from the original source code.

```
example(float** A, float** B, unsigned N) {
  for(i = 1; i <= N; i++) {
    for(j = 1; j <= N j++) {
      A[i][j] = B[i-1][j] + B[i][j-1]
    }
  }
}
```

Figure 3.4: Code example before any transformation.

Notice that this function can be invoked as $example(M_1, M_2)$, which can make this code trivially parallel if there is no aliasing between rows of the matrices. Or we can invoke it as $example(M_1, M_1)$, which produces a case very similar to the example presented in chapter 2

3.4.1 Perfect nest

To allow a wider set of loop transformations, when generating a pattern, we convert the original nest into its equivalent perfect for-loop nest (See fig 3.5). A perfect loop holds one of two possible conditions, the body consists of a sequence of non-loop statements with one entry point and one exit point, or the body consist of a perfect loop. A perfect nest is a loop nest with all its loops perfect.

Guarding code is added to ensure that each statement is executed at the right time.

Since a contribution of this thesis concerns this topic in particular, a wider explanation is given in the second part of this manuscript, in section 6.1.

```
while(true){
        while(true) {
                //if loop first iteration
                if(condition_placeholder) {
                        if(i <= N) j = 1
                        else goto loop_exit //outermost loop exit
                }
                if(j <= N) {
                        A[i][j] = B[i-1][j] + B[i][j-1]
                        j++
                } else {
                       i++
                }
        }
}
```

Figure 3.5: Code example converted to a perfect nest.

3.4.2 Virtual iterators

To be able to handle loops that do not have iterators (as *do-while* or *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. These iterators provide a common base for performing the predictions, interpolating the loop trip counts, and the accessed memory addresses as affine functions of these iterators. One virtual iterator is inserted for each loop in the nest. The new bounds for the virtual iterations are obtained using the Fourier-Motzkin elimination algorithm [36].

Transformation matrix

The matrix T^{-1} represents the inverse of the loop transformation. This matrix is used to compute the values of the iterators in the original iteration space, from the transformed space iterators.

In the example of figure 3.6, vi_x and vi_y represent the *transformed space* virtual iterators, while vi_i and vi_j represent the value of the iterators in the *original iteration space*. The original iteration space is sliced into chunks: the outermost loop index vi_i is ranging from the lower to the upper bound of a chunk, which are set at runtime. The calls to $FM_{lb_x}()$, $FM_{ub_x}()$, $FM_{lb_y}(vi_x)$ and $FM_{ub_y}(vi_x)$ stand for the previous mentioned calls to the Fourier-Motzkin elimination algorithm, implemented in the FMlib library [29].

Original loop bounds

Items ub_{vi_i} and ub_{vi_j} represent the loop bounds in the original iteration space, for a specific iteration vector (vi_i, vi_j). The matrix UB_T and the vector b are patched by the runtime system. The value for ub_{vi_i} depend on the chunk size. The values of these bounds are required to perform the verification explained in section 3.4.5.

3.4.3 Transformation selection

In the polytope model, transformations are represented as matrices. Before launching an optimized chunk, we have to choose an appropriate transformation.

The strategy used for selecting a transformation matrix in VMAD consist of two stages:

1. During compile-time, we perform *partial static dependence analysis* to propose a set of polyhedral transformations, valid with respect to the information available.

   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 method could benefit significantly from an off-line profiler, aimed to exhibit the information statically unavailable.

2. After the instrumentation phase, the dependence [20] information is computed to select a suitable transformation. We iterate through the list of transformations statically proposed for the loop nest, verifying the transformations for validity[3], until a suitable one is found. The current implementation simply returns the first found valid schedule.

We select a transformation matrix, named T, and the runtime system patches a convenient code skeleton, using the information gathered by the instrumentation phase. By patching different values a different iteration schedule is obtained.

For pedagogical purposes, in figure 3.7 we show with which values the loop bounds and the transformation matrix are patched. We patch the code skeleton with the values corresponding to the case were a skewing transformation is applied[4].

Notice how the bounds of the loop depend on the bounds of the chunk being executed.

[3] As mentioned earlier, the first non-null component of each row of the dependency matrix is positive.

[4] The same as in chapter 2

```
for(vi_x = FM_lb_x(); vi_x <= FM_lb_x(); ++vi_x){
        for(vi_y = FM_lb_y(vi_x); vi_y <= FM_ub_y(vi_x); ++vi_y) {
                (vi_i, vi_j) = T^-1 * (vi_x, vi_y)
                (ub_vi_i, ub_vi_j) = UB_T * (vi_i, vi_j) + b
                if(vi_j == 0) {
                        if(i <= N) j = 1
                        else goto loop_exit //outermost loop exit
                }
                if(j <= N) {
                        A[i][j] = B[i-1][j] + B[i][j-1]
                        j++
                } else {
                        i++
                }
        }
}
```

Figure 3.6: Code example with virtual iterators inserted.

$$
\begin{aligned}
&\textbf{for}(vi_x = \text{chunk\_no} + 1;\ vi_x <= \text{chunk\_no} + \text{optimized\_chunk\_size} + N - 1;\ ++vi_x)\{\\
&\quad \textbf{for}(vi_y = 1;\ vi_y <= \min(x,N);\ ++vi_y)\ \{\\
&\qquad \begin{pmatrix} vi_i \\ vi_j \end{pmatrix} = \begin{pmatrix} 1 & -1 \\ 1 & 0 \end{pmatrix} * \begin{pmatrix} vi_x \\ vi_y \end{pmatrix}\\
&\qquad \begin{pmatrix} ub_{vi_i} \\ ub_{vi_j} \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} * \begin{pmatrix} vi_i \\ vi_j \end{pmatrix} + \begin{pmatrix} \text{chunk\_no} + \text{optimized\_chunk\_size} \\ N \end{pmatrix}\\
&\qquad \ldots\\
&\quad \}\\
&\}
\end{aligned}
$$

Figure 3.7: Code example with patched values.

3.4.4 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 [38, 30, 25, 11].

Note that the scalars which need to be initialized are the ones whose values depend on a previous iteration, i.e., which are read before being written in the current iteration. In the SSA form [8], these correspond to the ϕ nodes [5].

This practice ensures that the correct starting value is assigned in the beginning

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

of each iteration. These scalars are named *basic scalars*, because the values of all other variables can be deduced from them.

The initialization code is equivalent to privatization[6], since all values that depend on other iterations are redeclared locally in each thread and initialized with the interpolating linear functions.

3.4.5 Verification code

Verification code is required to validate or invalidate the speculations. Since the memory state and the control flow have been modified different types of verification code need to be introduced.

- To verify **memory accesses**: As some iterations might execute before being validated by the preceding iteration space(according to the sequential order), VMAD requires to verify all memory accesses, in the current iteration. Namely, it must ensure that each instruction that accesses memory, targets a location that has been predicted using the associated linear interpolating function. 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.

- To verify **basic scalars**: When the execution of the iteration completes, the value computed by the code and the value predicted by the speculation are compared. For this verification, the actual value with the one expected for the *next iteration in the sequential order* are compared. Notice that, since the values of the basic scalars at the end of one iteration corresponds to the initial value of the basic scalars of the next sequential iteration, each iteration verifies the next one in the sequential order, even if this iteration has been executed or not. Nevertheless, any dependence violation is necessarily detected at a given time, either before or after it occurs.

- To verify **loop bounds**: To ensure that no unpredicted iterations are executed, and that the exact number of iterations is executed (no more, no less), code verifying the exit bounds has to be introduced. This code verifies that the speculation on the loop bounds is correct. The verification code consist of a compare between the original exit condition and the speculation. Since in most non-statically analyzable nests it is impossible to predict the iteration in which the exit conditions is reached, a miss-prediction is signaled

[6] A variable inside a loop can be privatized if an assignment is performed to it before any use inside the loop.

if the exit conditions is reached, because extra iterations might have been executed.

To illustrate the scalar verification, figure 3.8 is presented. The first figure represents the original iteration order of the loop nest (important: the arrows don't represent dependences). The iteration vector (i, j) takes the following sequence of values: $(1,1)$, $(1,2)$, $(1,3)$, ..., $(2,1)$, $(2,2)$, ..., $(3,3)$.

After applying a skewing transformation to the loop nest, one obtains a new iteration space. Since the outermost loop of this new iteration space is parallelized, the execution of the entire iteration space is divided in two different sets, one for thread 1 and one for thread 2 (See the third picture). Furthermore, as a skewing transformation was applied, the iteration execution order changed: Thread 1 will execute the iterations in the following order $(1,1)$, $(2,1)$, $(1,2)$ and $(3,1)$. $(2,2)$, $(1,3)$, ... While thread 2 will execute $(3,2)$, $(3,3)$ and $(2,3)$.

Independently of the transformed iteration order, each iteration verifies the next one in the original sequential order. For example, iteration $(2,2)$ will verify the initial scalar values for iteration $(2,3)$, and iteration $(2,1)$ will verify the initial scalar values for iteration $(2,2)$.

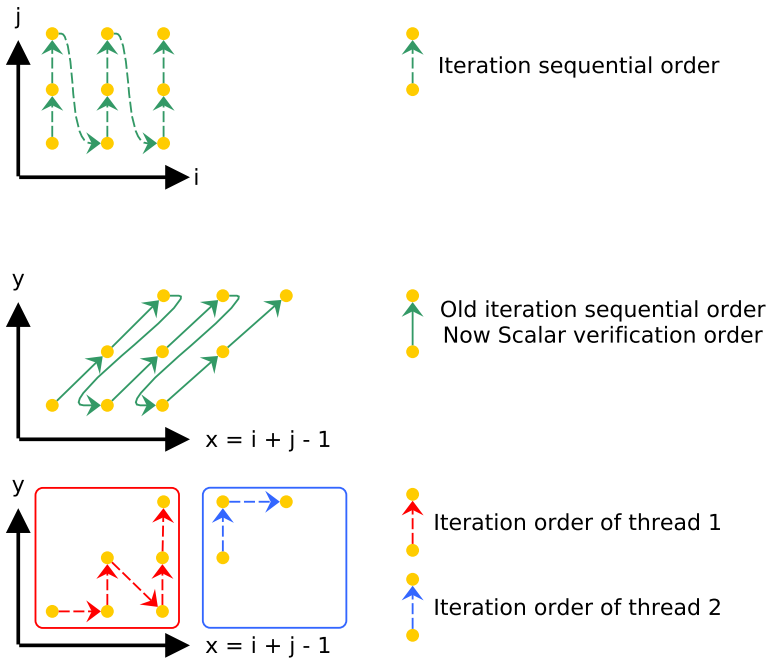


Figure 3.8: Verification order in a transformed iteration space

3.4.6 Chunking

To adapt to the current behavior of a code during execution, the *chunking system* is introduced. A loop chunk is defined by a slice of *consecutive iterations of the outermost loop* of the nest. As an example, the first chunk may contain the first 10 iterations, the next chunk the 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, VMAD can execute parallel chunks followed by sequential chunks, without missing any parallelization opportunities in *partially parallel loops*[9].

Inside each parallel chunk, *slices* of the chunk are scheduled for each thread. VMAD relies on *libGOMP* [16] for the parallel execution of slices. Notice that the chunks are scheduled sequentially, but inside a parallel chunk, slices of the chunk are executed by different threads.

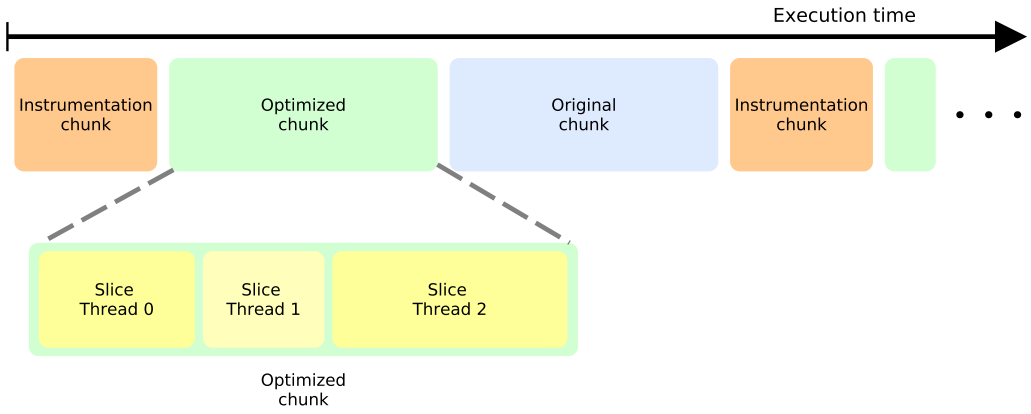


Figure 3.9: Loop chunking.

3.4.7 Miss-prediction handling

Once a miss-prediction is encountered, it is signaled 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.

Any speculative system relies on a mechanism to restore the execution upon a miss-speculation. Since faulty iterations can alter memory with incorrect values, an anticipated memory backup is performed before launching a chunk of code embedding a speculatively 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. Thus, for each write instruction, it is necessary 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*. The memory in this ranges is backed up, such that it can be restored by a reversed copy, if the iterations are canceled.

The completed code skeleton is shown in figure 3.10.

```
for(vi_x = FM_{lb_x}(); vi_x <= FM_{lb_x}(); ++vi_x){
    for(vi_y = FM_{lb_y}(vi_x); vi_y <= FM_{ub_y}(vi_x); ++vi_y) {
        (vi_i, vi_j) = T^{-1} * (vi_x, vi_y)
        (ub_{vi_i}, ub_{vi_j}) = UB_T * (vi_i, vi_j) + b

        //initialization code
        i = v_i * (vi_i, vi_j) + b_i
        j = v_i * (vi_i, vi_j) + b_j

        if(vi_j == 0) {
            if(i <= N) j = 1
            else rollback() //outermost loop exit
        }
        if(j <= N) {
            pred_A = v_{pred_A} * (vi_i, vi_j) + b_{pred_A}
            pred_{B_1} = v_{pred_{B_1}} * (vi_i, vi_j) + b_{pred_{B_1}}
            pred_{B_2} = v_{pred_{B_2}} * (vi_i, vi_j) + b_{pred_{B_2}}

            //memory access verification
            if(&A[i][j] != pred_A) rollback()
            if(&B[i-1][j] != pred_{B_1}) rollback()
            if(&B[i][j-1] != pred_{B_2}) rollback()

            A[i][j] = B[i-1][j] + B[i][j-1]
            j++
        } else {
            i++
        }

        //loop bounds verification
        if((j <= N) != (vi_j < ub_{vi_j})) rollback()

        //compute next iteration
        (vi_{i_next}, vi_{j_next}) = (vi_i + (vi_j == ub_{vi_j}), (vi_j + 1) * (vi_j != ub_{vi_j}))

        //scalar verification
        if((i != v_i * (vi_{i_next}, vi_{j_next}) + b_i) rollback()
        if((j != v_j * (vi_{i_next}, vi_{j_next})) + b_j) rollback()
    }
}
```


Figure 3.10: Code skeleton built from the code sample.

Part II

Contributions

Chapter 4

Straightforward code skeleton

In this chapter we introduce the *Straightforward code skeleton*. This pattern corresponds to the parallelization of a loop nest without any code transformation, by executing in parallel the outermost or second loop. Since no transformation is applied, important optimizations can be made compared to the other parallel patterns supporting transformations.

The transformation matrix which corresponds to this pattern is the identity matrix.

4.1 Motivation

Let us consider as a motivating example the loop nest in figure 4.1.

```
while(node != 0x0) {
    node->element = 2*node->element + 3
    node = node->next
}
```

Figure 4.1: Motivating example for the straightforward pattern.

This code performs a 'map' over a linked list. The compiler can't ensure that the nodes of this list don't alias –there may be loops in the linked list. Also, it is impossible to know in which iteration the exit condition of the loop will be reached. This code can't be parallelized using the traditional compile time techniques.

If the memory accesses of this code are linear, the straightforward skeleton can be used, only by making the outermost loop parallel. Since a speculation is performed verification code must be executed, but the amount of verification code is significantly reduced.

4.2 Loop bounds

When generating the parallel code, the linear functions interpolating the loop bounds are used directly. The coefficient values for this linear functions are patched in the code skeleton by the runtime system, and from this values, the bounds for each loop are computed.

This is in contrast with patterns where a transformation is applied, which perform calls to the Fourier-Motzkin algorithm, for getting new bounds for the loops.

Given this, for the nest in figure 4.2, the result of inserting the *virtual iterators* and the loop bounds for the case of straightforward parallelization is shown in figure 4.3.

```
while(...) {
    while(...) {
        /* body */
        ...
    }
}
```

Figure 4.2: Code example before inserting the virtual iterators.

```
for(vi_i = 0; vi_i < ub_vi_i; ++vi_i) {
    for(vi_j = 0; vi_j < ub_vi_j_i * i + ub_vi_j; ++vi_j) {
        /* body */
        ...
    }
}
```

Figure 4.3: Code example of figure 4.2 after inserting the virtual iterators and the upper bounds for the straightforward parallelization skeleton.

4.3 Basic scalars initialization and verification in sequential order

When running a chunk, slices of the chunk are scheduled for each thread. The basic scalars must be initialized at the beginning of each slice, using the linear functions obtained from the instrumentation phase.

Since no transformation is applied, inside each slice, the original sequential execution order is kept. This implies that, at the end of each iteration, the values of the basic scalars correspond to the initial values of the next iteration in the transformed version. Thus, there is no need to initialize the basic scalars for the next iteration.

Moreover, the basic scalar verification is only performed at the end of each slice, to validate the initial values of the following slice.

The initialization iterations and the verification iterations inside each slice is represented in the figure 4.4.

Notice that the values taken by the basic scalars may *not* be linear during the execution of a slice. However, the linearity must be kept at the edges of the slices. Even if the basic scalars are not linear, the linearity constraints over the memory accesses and the loop bounds must be verified on every iteration.

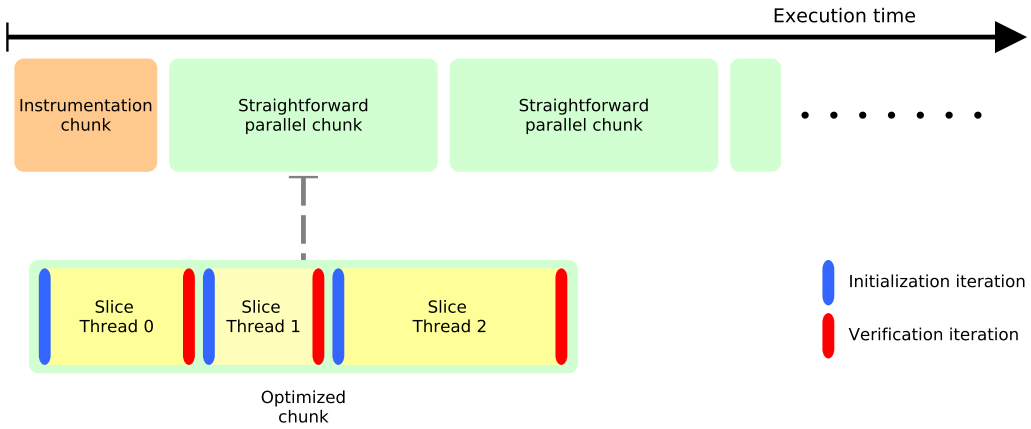


Figure 4.4: Straightforward parallel pattern scalar verification iterations.

Chapter 5

Tiled pattern

The code transformations which may be applied by VMAD, depend strictly on the patterns created at compile-time. For transformations which alters the nest structure, a new code skeleton must be created in order to handle this new transformation. In this chapter, a code skeleton, which enables a transformation named *tiling*, is explained.

5.1 Motivation

In parallel programs, poor data locality harms program scalability, since there may be contention over the memory bus.

The example on figure 5.1 corresponds to a Gaussian filter with a 3×3 stencil. Notice that element $A[i][j]$ accessed on one iteration, named (i', j'), would be referenced again by the next iteration, $(i', j' + 1)$, and by iteration $(i' + 1, j')$. Although this value is reused by the next iteration, the large volume of data in the upcoming iterations may replace the data from the cache before it can be reused by iteration $(i' + 1, j')$.

Tiling counteracts this by reordering the iterations such that iterations from the outer loops are executed before completing iterations from the innermost loop.

Tiling is a transformation which aims to improve data locality by exploiting reuse in several iteration directions.

The result of tiling the loop in figure 5.1, is listed in figure 5.2. Since the number of loops is duplicated, new iterators, named i_T and j_T, are introduced to iterate over each individual tile.

Notice the coefficients t_i and t_j which set the *maximum size of the tiles*. The values of these coefficients depend strictly on the loop being parallelized and on the underlying hardware, and must be chosen to maximize reuse. Moreover, due to the lack of information available at compile-time for most of the loop nests, it

```
for(i = 0; i ≤ N; ++i) {
    for(j = 0; j ≤ M; ++j) {
        B[i][j] = (A[i+1][j] + A[i][j-1] + A[i][j] + A[i][j+1] + A[i-1][j])/5 ;
    }
}
```

Figure 5.1: Gaussian filter nest.

```
for(i_T = 0; i_T ≤ N; i_T+=t_i) {
    for(j_T = 0; j_T ≤ M; j_T+=t_j) {
        for(i = i_T; i ≤ min(N, i_T+t_i-1); ++i) {
            for(j = j_T; j ≤ min(M, j_T+t_j-1); ++j) {
                B[i][j] = (A[i+1][j] + A[i][j-1] + A[i][j] + A[i][j+1] + A[i-1][j])/5 ;
            }
        }
    }
}
```

Figure 5.2: Tiled Gaussian filter nest.

is difficult to choose statically values for these coefficients without missing optimization opportunities. Taking this into account, in section 5.4 a mechanism for tuning dynamically the tile sizes is presented.

5.2 Loop transformation and tiling

Loop tiling transforms a n-depth loop nest in a 2n-depth loop nest. The innermost n loops execute the iterations within a given tile, while the outermost n loops iterate over the tile space.

In the tiling proposal of this thesis, the original loop is tiled and a loop transformation is applied over the outermost n loops. In this way, the original iteration order inside each tile is kept, but the order in which the tiles are scanned is modified.

As an example for this section, the nest in figure 3.4 from chapter 3 is reproduced in figure 5.3.

5.2.1 Loop nest construction

When constructing the code-skeleton which enables the tiling transformations, the loop first is made perfect in the same way as explained in section 3.4.1. Then, for a n-depth nest, the loop nest is surrounded by new n loops.

```
example(float** A, float** B, unsigned N) {
  for(i = 1; i <= N; i++) {
    for(j = 1; j <= N j++) {
      A[i][j] = B[i-1][j] + B[i][j-1]
    }
  }
}
```

Figure 5.3: Loop nest of figure 3.4 from chapter 3.

Recalling the notion of *virtual iterators*, which is explained in section 3.4.2, $2n$ virtual iterators are created in this pattern, n for the new loops, to iterate over each tile, while the other half is inserted at the innermost n loops, to scan each individual tile.

$$\begin{pmatrix} ts_{vi_i} \\ ts_{vi_j} \end{pmatrix} = T^{-1} * \begin{pmatrix} ts_{vi_x} \\ ts_{vi_y} \end{pmatrix}$$

```
for(vi_x = FM_lb_x(); vi_x <= FM_lb_x(); vi_x += ts_vi_x){
  for(vi_y = FM_lb_y(vi_x); vi_y <= FM_ub_y(vi_x); vi_y += ts_vi_y) {
    (vi_i0, vi_j0)^T = T^-1 * (vi_x, vi_y)^T
    for(vi_i = vi_i0; vi_i < min(vi_i0 + ts_vi_i, ub_vi_i); vi_i++) {
      for(vi_j = vi_j0; vi_j < min(vi_j0 + ts_vi_j, ub_vi_j * i + ub_vi_j); vi_j++) {
        if(vi_j == 0) {
          if(i <= N) j = 1
          else goto loop_exit //outermost loop exit
        }
        if(j <= N) {
          A[i][j] = B[i-1][j] + B[i][j-1]
          j++
        } else {
          i++
        }
      }
    }
  }
}
```

Figure 5.4: Code example with virtual iterators inserted.

In a similar way as in the pattern explained in chapter 3, the upper-bounds of the loops, which iterate on the transformed space, are computed using the Fourier-Motzkin elimination algorithm. In contrast with the non-tiled skeletons, the outermost loop iterators are incremented with a step depending on the loop tile

size. The values taken by the tile sizes are adjusted and patched by the runtime system.

The initial values for the tile iterators, $< vi_i, vi_j >$, are computed by multiplying the inverse of the transformation matrix, T^{-1}, with the value of the transformed space iterators. In a similar manner, since the innermost loops iterate in the original sequential order, while the outermost iterate on the transformed iteration space, new tile sizes must be computed by multiplying the outermost tile sizes with T^{-1}.

In the loops which iterate over one tile, the upper-bound is the minimum between the tile bound (the initial value of the corresponding iterator plus the tile size) and the original loop bound, computed by the dot product from the iterator values, and the coefficients interpolated by the runtime system.

5.2.2 Scalar initialization and verification

One particularity of how tiling is implemented in this code skeleton, is that the iteration order inside each tile, keeps the original sequential order. Notice that, at the end of an iteration of the innermost loop, the values of the basic scalars correspond to the initial values of the next sequential iteration. This is only true for the innermost loop, since the slices executed by the innermost loops may finish before reaching their original exit condition; it can reach the tile bound. This is illustrated in figure 5.5.

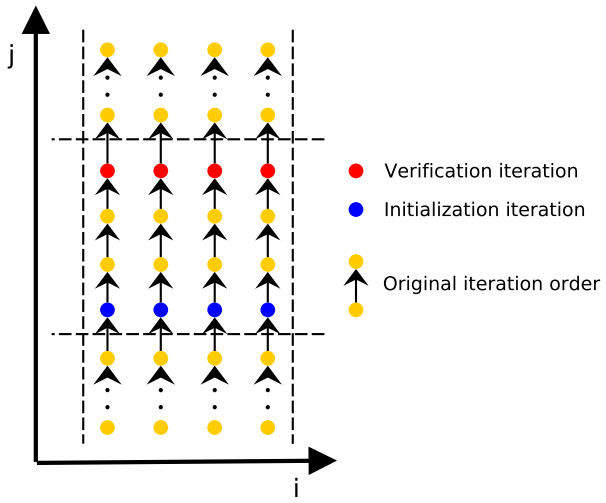


Figure 5.5: Scalar verification iterations in the tiling code skeleton.

In this way, the initialization code must be executed before entering the innermost loop, while the verification code is executed after exiting the innermost loop.

The memory access verification code and the loop bounds verification is inserted in the same way as in the other code skeletons.

5.3 Dependency analysis

In order to decide if the tiling transformation can be applied to parallelize a given loop nest, the dependency analysis mechanism was extended.

First, a set of dependence distance vectors is obtained from the instrumentation phase, they have the form $\vec{d_N} = \begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{pmatrix}$, where n is the depth of the loop nest. By decomposing each vector $\vec{d_N}$ in its canonical form, one obtains the dependence vectors for the n-loops which iterate over the tiles.

$$\vec{d_N} = \begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{pmatrix} \rightarrow \left\{ \begin{pmatrix} \frac{d_1}{|d_1|} \\ 0 \\ \vdots \\ 0 \end{pmatrix} \quad \begin{pmatrix} 0 \\ \frac{d_2}{|d_2|} \\ \vdots \\ 0 \end{pmatrix} \quad \dots \quad \begin{pmatrix} 0 \\ 0 \\ \vdots \\ \frac{d_n}{|d_n|} \end{pmatrix} \quad \forall d_i, d_i \neq 0 \right\}$$

.

Using this new set of dependences, we compute the scalar product between each dependence and the transformation matrix T . Then, we verify for each transformation if the first-non null component of each resultant vector is positive. Since each first non-null component identifies a loop carrying a dependence, the parallel loop is the outermost which is not carrying any dependence. If this parallel loop exists and is the outermost or the second of the loop nest, then, it is possible to launch a chunk with the tiled skeleton, patched in the appropriate way.

As an example, lets suppouse that the dependence vector obtained from instrumentation are $d_1 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$ and $d_2 = \begin{pmatrix} 0 \\ 2 \end{pmatrix}$.

The new dependences for the loops which iterate over the tiles are:

$$\left\{ \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\} \bigcup \left\{ \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\}$$

If we apply the transformation $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ we obtain the following dependences. Notice that the second loop is parallel, so we are able to launch a tiled chunk.

$$\left\{ \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \end{pmatrix} \right\}$$

5.4 Tile size adjustment

Selecting the right values for the tile sizes is mandatory to achieve good performance when tiling. Choosing the wrong values for this coefficients may result in pollution of the data in cache, harming the advantages obtained by the tiling transformation. Furthermore, these values depend strongly on the underlying hardware (cache size, associativity, etc...).

In the considered loop nests, there is little information available at compile-time to select the right values for these coefficients accurately. Nevertheless, we got an advantage over static analysis, since we can actually run and measure the execution time of the code.

The approach that is presented is guided by:

- *Dynamically adjusting* the tile sizes, without making assumptions about the underlying hardware or the loop-nest.
- Being *lightweight*, to avoid incurring a large overhead to the chunk execution or the decision making mechanism.

The approach chosen for adjusting the tile sizes is the following:

1. Before executing the first chunk with a tiled skeleton, we assign random values to the tile sizes for each thread. The idea is to execute with a different tile size configuration, on each thread.
2. The chunk is launched, and per-thread information about the execution is stored. In particular, we are interested in information regarding the number of *executed tiles* and the *running time*. This information will be used to approximate the performance of a single thread with a given tile-size configuration.
3. Between each chunk, the tile sizes are adjusted. By multiplying the tile sizes $ts_x * ts_y * \dots$, it is possible to bound the number of iterations executed by one tile. By multiplying this value with the number of executed tiles $(ts_x * ts_y * \dots) * executed\_tiles$ the number of executed iterations by a thread is approximated.

We obtain a *per-thread score* by dividing the approximation of the number of executed iterations with the time it took to execute these iterations.

$$score = \frac{(ts_x * ts_y * \dots) * executed\_tiles}{running\_time}$$

4. Using the *per-thread score*, we select the winner tile size configuration. This configuration is replicated for each thread, and slightly adjusted. Then, a new chunk is launched with this new tile size configuration.

Steps 3 and 4 are repeated until a misprediction occurs or the loop exit is reached. After this situation, all the counters are restarted, and all the information regarding the tile sizes is discarded –A change of the behavior of the loop occurred, so all the information collected is invalidated.

Using this approach, the tile sizes are adjusted dynamically, converging eventually to a 'good' set of tile-size values.

Performance counters

For collecting the previously mentioned information about each thread execution, two counters are inserted. One for measuring time, and another for counting the number of executed tiles. These counters are initialized to zero before launching a chunk, as shown in figure 5.6.

```
time_before = time()
for(vi_x = FM_lb_x(); vi_x <= FM_lb_x(); vi_x += ts_vi_x){

    executed_tiles = executed_tiles + FM_ub_y(vi_x) - FM_lb_y(vi_x)

    for(vi_y = FM_lb_y(vi_x); vi_y <= FM_ub_y(vi_x); vi_y += ts_vi_y) {
        ...
        for(vi_i = vi_i0; vi_i < min(vi_i0 + ts_vi_i, ub_vi_i ); vi_i++) {
            for(vi_j = vi_j0; vi_j < min(vi_j0 + ts_vi_j, ub_vi_j_i * i + ub_vi_j ); vi_j++) {
                ...
            }
        }
    }
}
time_after = time()
total_time = total_time + time_after - time_before
```

Figure 5.6: Code example with performance counters inserted.

Tile size adjustment results

In figures 5.7 and 5.8 we show the experimental results of performing the tile size adjustment. Notice how as the chunks execute, the average score of the threads converge to the best score of each iteration.

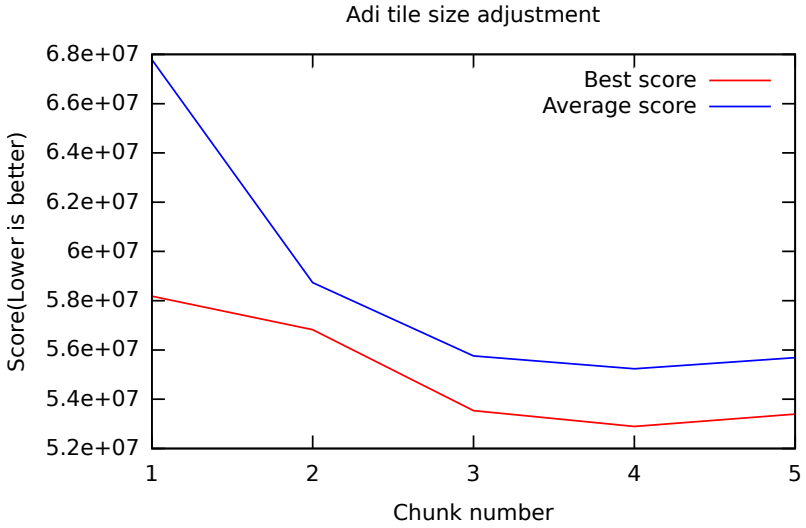


Figure 5.7: Tile size adjustment results for Adi benchmark.

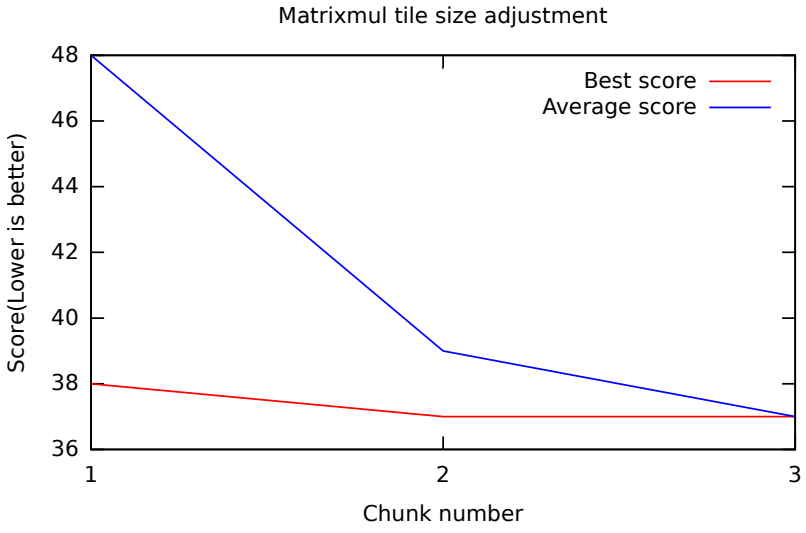


Figure 5.8: Tile size adjustment results for Matrixmul benchmark.

Chapter 6

Optimizations and corrections

6.1 Guarding code

For VMAD to handle a wider set of loop nests and transformations, when generating the different code skeletons, the loop nests are transformed to a perfect nest. This allows the code-skeletons to handle transformations, such as loop interchange, by patching a given set of scalars in the code skeleton.

For making a loop perfect, we need to move the code which is originally situated before and after the contained loop inside the innermost loop. Also, we want to keep the original behavior of the loop nest, thus, we must guard the code moved inside the innermost loop to ensure that it is executed at the right time. Notice that the code after the inner loop will be executed only after the exit condition of the inner loop is reached. In a similar way, the code before the inner loop is executed before the first iteration of this loop.

To check if the current iteration is the last one of the contained loop nest, we use the inner loop exit condition. When this conditions becomes false, we execute the code after the contained loop.

For checking if the current iteration is the first one of the contained loop nest, the process is slightly different, since, in the more general case, without adding any extra code, there is no way to identify if the current iteration is the first one of the contained loop. To overcome this limitation, when generating the skeletons, we use the *virtual iterators* for this purpose. But at the moment of making the nest perfect, the *virtual iterators* are not inserted in the skeleton. For this reason, we insert a *placeholder* for the guard condition. Once the *virtual iterators* are inserted in the loop nest, we replace the *placeholder* with the corresponding condition.

For a n-depth loop nest, where $i_1, i_2, \ldots, i_n$ are the virtual iterators of the nest, the guard condition for the $i-th$ loop is $(i_{i+1} == 0) \wedge (i_{i+2} == 0) \wedge \ldots (i_n == 0)$.

An example of transforming a loop nest to perfect as described in this section

is presented in figures 6.1 and 6.2.

```
while(cond1) {
  ... //Body1
  while(cond2) {
    ... //Body2
  }
  ... //Body3
}
```

Figure 6.1: Loop nest example before making it perfect.

```
while(true) {
  while(true) {
    if(condition_placeholder) {
      if(cond1) {
        ... //Body1
      } else break; //Outermost exit
    }
    if(cond2) {
      ... //Body2
    } else {
      ... //Body3
    }
  }
}
```

Figure 6.2: Loop nest example after making it perfect.

```
for(vi_x = FM_lb_x(); vi_x <= FM_lb_x(); ++vi_x){
        for(vi_y = FM_lb_y(x); vi_y <= FM_ub_y(vi_x); ++vi_y) {
                (vi_i, vi_j) = T^-1 * (vi_x, vi_y)
                if(vi_j == 0) {
                        if(cond_1) {
                          ... //Body_1
                        } else rollback(); //Outermost exit
                }
                if(cond_2) {
                  ... //Body_2
                } else {
                  ... //Body_3
                }
        }
}
```

Figure 6.3: Loop nest example after making it perfect with the placeholder replaced.

6.2 Memory access verification avoidance

As we seen before, the amount of verification code that is inserted in the different code skeletons can introduce a significant overhead in the loop execution.

Is there any situation where we can avoid the execution of such verification code ?

Let us consider the example in figures 6.4, 6.5 and 6.6.

```
for(...) {
    *(ptr+0) = ...
    *(ptr+1) = ...
    *(ptr+2) = ...
          ⋮
    *(ptr+7) = ...
}
```

Figure 6.4: Loop nest after an unroll transformation was applied.

The example in figure 6.4 corresponds to a loop nest, after a code transformation known as loop unrolling was applied. In this transformation, several loop iterations are written as a sequence of similar independent statements. Notice that

after the first memory access to *ptr* is verified to be linear, we can ensure that the remaining accesses to $ptr + 1$, $ptr + 2$ and $ptr + 3$, are linear too.

```
while(...) {
    ... = node->b + ...
    node->a = ...
}
```

Figure 6.5: Loop nest were the operations are performed by accessing fields of a structure.

The case presented in figure 6.5 is similar to the last one. To access one field of a structure, we should add the *offset* which corresponds to the field we want to access, to the base memory position in which the structure is stored. In this example, the memory accesses are to the address $node + offset_b$ and $node + offset_a$. In the example, if the memory access to $node + offset_b$ is linear, the access to $node + offset_a$ will be necessarily linear too.

```
while(...) {
    for(ptr = node->data; ptr != node->data + length; ++ptr) {
        *ptr = ...
    }
}
```

Figure 6.6: Loop nest were the memory access address correspond to a basic scalar value.

In the last example, of figure 6.6, the memory access is performed using as a pointer the value of a *basic scalar*. The values taken by this scalar would be interpolated with linear functions by the runtime system, and initialized using linear functions. Thus, if a memory access is a linear transformation from the basic scalars, we are sure it will behave as a linear function of the virtual iterators. This last paragraph is not true for the skeletons which implements the tiling and straightforward parallelization patterns. For the tiling skeletons it is only valid if the basic scalar is outside the innermost loop.

We developed an analysis for identifying this situations. The analysis verifies for each memory access in the loop, if it is a linear transformation of:

- The *basic scalars* of the loop nest.
- Other *memory accesses* inside the loop nest, which *strictly dominates* [1] the

[1] An instruction A **dominates** an instruction B, if every path from a *start node* to B must go through A. Trivially, B dominates itself.

memory access being verified.

The results of this analysis are encoded in the *llvm intermediate representation* as meta-data attached to the memory access instructions. For a particular memory access, we attach the depth of the innermost loop nest, where a basic scalar intervenes in the memory address computation. For example, in the nest in figure 6.6, the value attached to the unique memory access instruction would be 2. If the memory address is computed as a linear transformation from other memory addresses, known to be linear or verified at runtime, the value attached to the instruction is 0. If the memory access is not linear from the basic scalars or other memory accesses, we don't attach any meta-data to the instruction.

When generating the multiple optimized versions, different criteria are taken into account for deciding to insert or not verification code.

- **Transformed parallel**: We avoid inserting verification, only if meta-data identifying the access as linear is attached to the instruction.
- **Straightforward parallel**: We insert verification code excepting when the value attached to the memory access instruction is equal to 0.
- **Tiling**: We avoid inserting verification code if the value attached to the instruction is different from the original loop nest depth (The depth of the innermost loop in the original nest).

This analysis is implemented as a separate compiler pass. By doing a depth first search through the instructions, starting with the memory address referenced by the memory instruction, we traverse the data flow graph, until a conclusive result is obtained. For a reference on the llvm instruction set the reader can refer to [24].

Each instruction type must be handled separately in the following way:

- *Constant* or *Instruction outside the loop nest*: the analysis returns $(linear, 0)$.
- For a *Basic scalar* return $(linear, basic\ scalar\ depth\ in\ the\ nest)$.
- If the instruction is a *Memory address known linear which dominates* the one being analyzed return $(linear, 0)$.
- For a *Cast instruction*, if the cast keeps the linearity, for example, from i32 to i64 signed cast, the analysis result is the same as with the operand being casted.
- For a *Binary operator instruction*:

  - For addition or subtraction the analysis is the same. First analyze both operands, if both are linear return $(linear, max(depth_{op_1}, depth_{op_2}))$, where $depth_{op_i}$ is the second component of the tuple returned by the i-th operand.
  - For multiplication, one of the operands must stay constant during the execution of the loop, and the other must be linear. The same result as with the linear operand is returned.
  - For every other case, including division, we return $non - linear$.

- The *Get element pointer* is an instruction for the computation of memory addresses. Its operands consist of one base pointer, and several indexes. The address computation is performed in the following way: $base\_ptr + cte_0 * idx_0 + cte_1 * idx_1 + \cdots + cte_n * idx_n$, where the constants $cte_0, \ldots, cte_n$ are known at compile time. In a similar manner to the binary addition and subtraction, if every operand is linear, the result is $(linear, max(depth_{base\_ptr}, depth_{idx_0}, \ldots, depth_{idx_n}))$.

- In every other case, return $non - linear$.

Part III

Results and conclusions

Chapter 7

Evaluation of the optimizations

In the previous chapters we introduced a technique for runtime optimization known as speculative code parallelization. To enable adaptive behavior during execution, several code-skeletons are generated: one specialized for instrumentation, one matching the original loop behavior, and several skeletons matching different optimizing code transformations.

In this thesis, two new code skeletons are presented, one which enables the tiling transformation and one which parallelizes the code, but keeps the original sequential iteration order. Section 7.1 presents the evaluation of the different code skeletons when parallelizing code and varying the number of threads.

The configuration on which the tests were run embeds two AMD Opteron Processors 6172, of 12 cores each, at 2.1 Ghz, running Linux 3.2.0-36-generic x86_64, and 32 Gb of ram memory.

The results obtained from running the benchmarks are presented in Appendix A.

7.1 Parallel, Straightforward and Tiled code skeletons

In this section we present the results we obtained with speculative parallelization, while using different code skeletons on codes which cannot be parallelized using the traditional compile time techniques. The set of benchmarks comes from different sources: The polyhedral benchmark suite [28], the Rosetta code website [35] and some codes implementing linear algebra algorithms. Since some of these codes can be handled statically, we have modified them to use dynamically allocated arrays or pointers, preventing static analysis.

In figures 7.1, 7.3, 7.5, 7.7 and 7.9 we shows performance comparisons between

the original code[1], the previous existing parallel code skeleton, the straightforward parallelization skeleton and the tiling skeleton.

Opposed to the performance comparisons, to evaluate the scalability of the code skeletons, we measured the execution time of the loop, *ignoring the time spent to perform memory backup*. The results obtained are shown in figures 7.2, 7.4, 7.6, 7.8 and 7.10.

The benchmarks were run using the identity as the transformation matrix, to be able to compare with the straightforward parallelization pattern. Additionally, the *gauss 2d kernel* was run with a loop interchange.

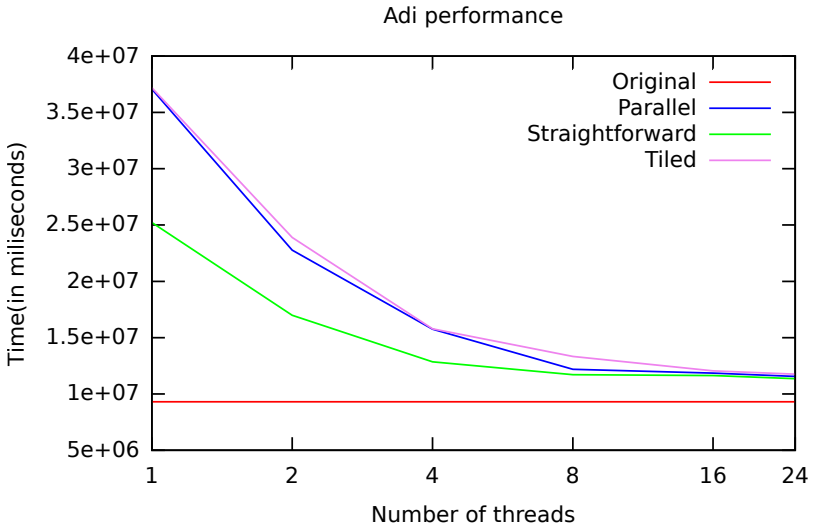


Figure 7.1: *Adi-polybench kernel* performance results.

[1]Compiled with gcc with the -O3 option.

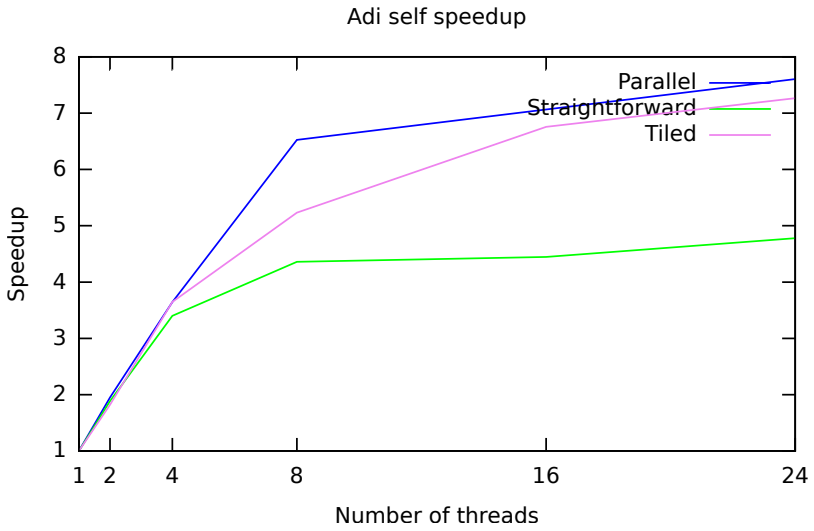


Figure 7.2: *Adi-polybench kernel* self speedup results.

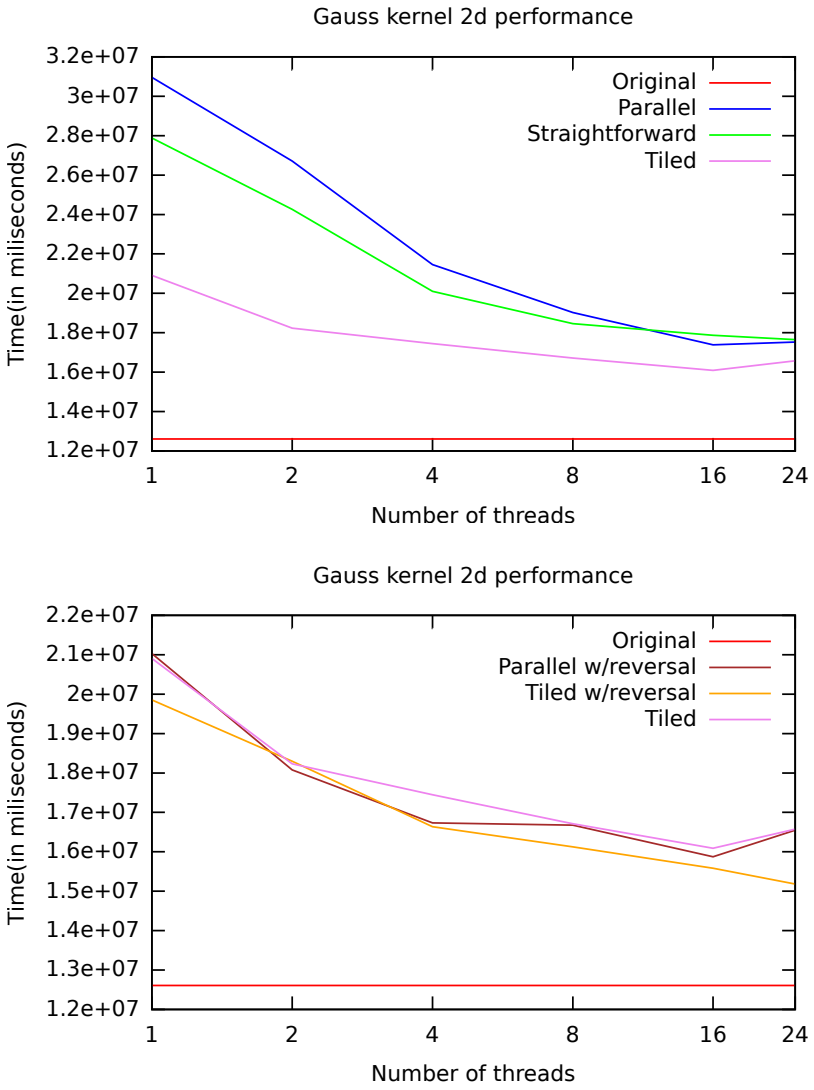


Figure 7.3: *Gauss 2d kernel* performance results.

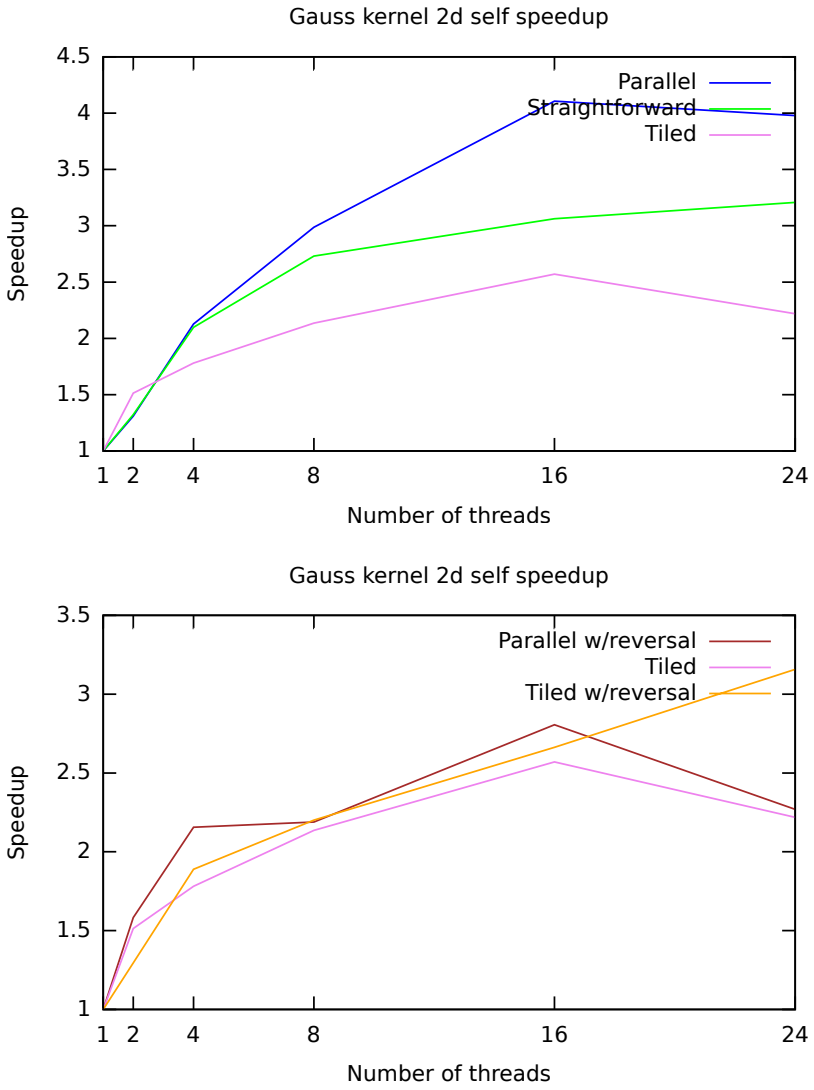


Figure 7.4: *Gauss 2d kernel* self speedup results.

For the kernels adi-polybench (See figures 7.1 and 7.2) and gauss 2d (See figures 7.3 and 7.4) speedup over the original code was never reached, even after increasing the number of threads.

A further analysis shows that the time spent doing the memory backup corresponds to the 82% and 103% of the time executing the original code. Thus, the penalty imposed by the memory backup harms any benefits of speculative parallelization.

The results obtained by applying a loop interchange outperformed the ones parallelized with the identity matrix as transformation. Nevertheless, the tiled code

skeleton reduced the performance penalty imposed by using the identity schedule.

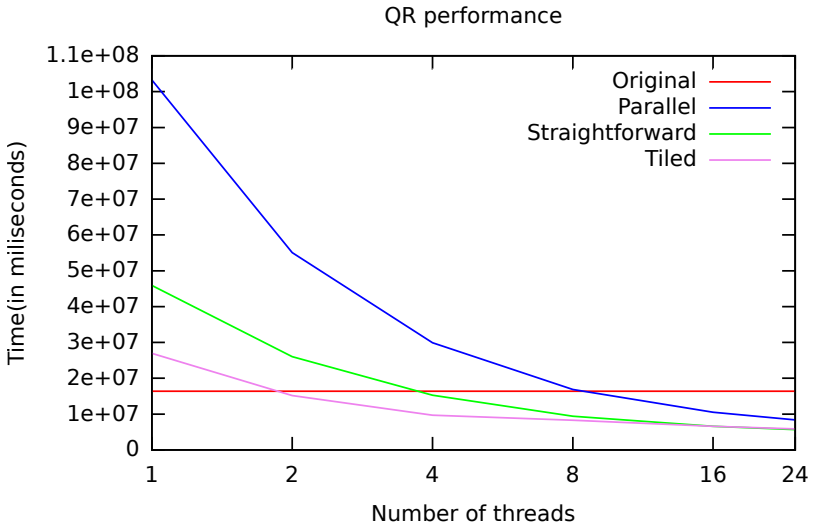


Figure 7.5: *QR decomp kernel* performance results.

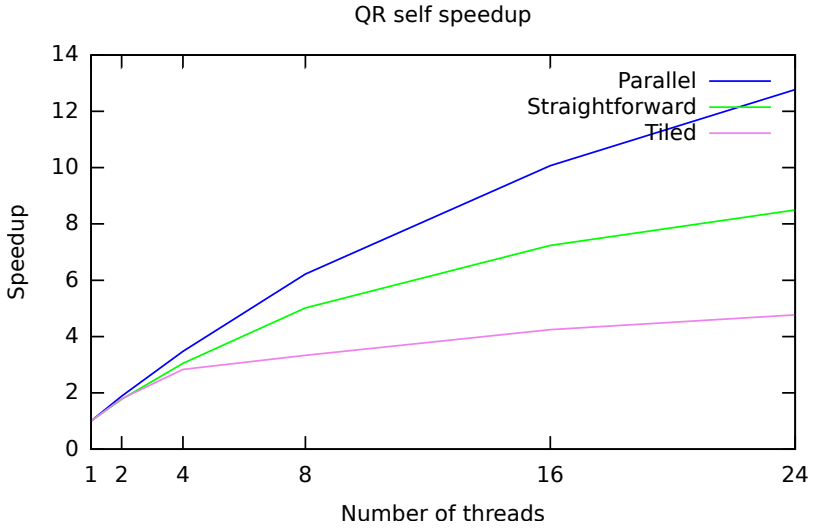


Figure 7.6: *QR decomp kernel* self speedup results.

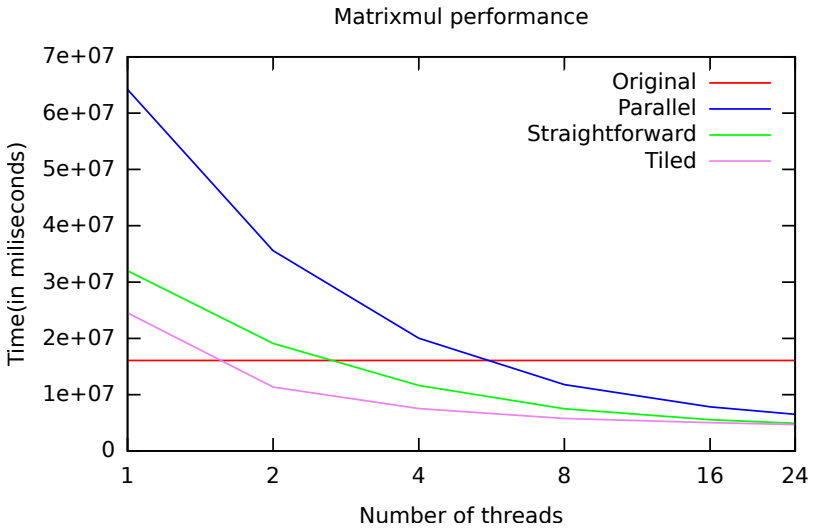


Figure 7.7: *Matrixmul kernel* performance results.

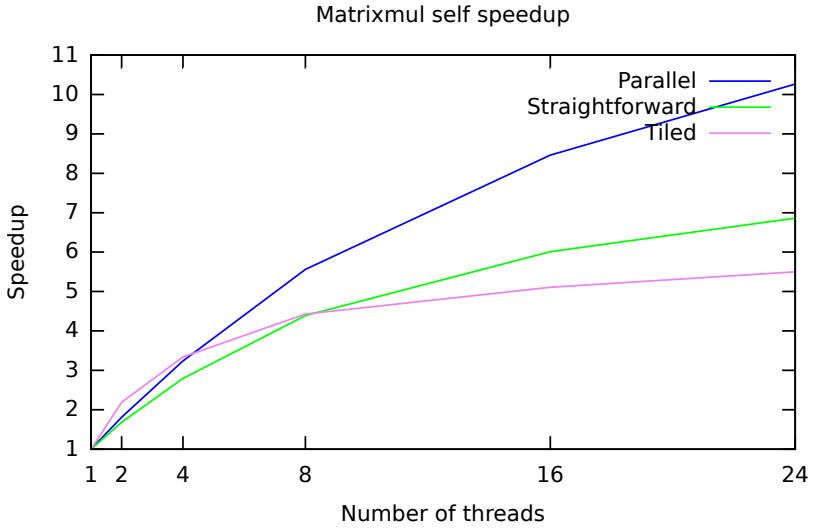


Figure 7.8: *Matrixmul kernel* self speedup results.

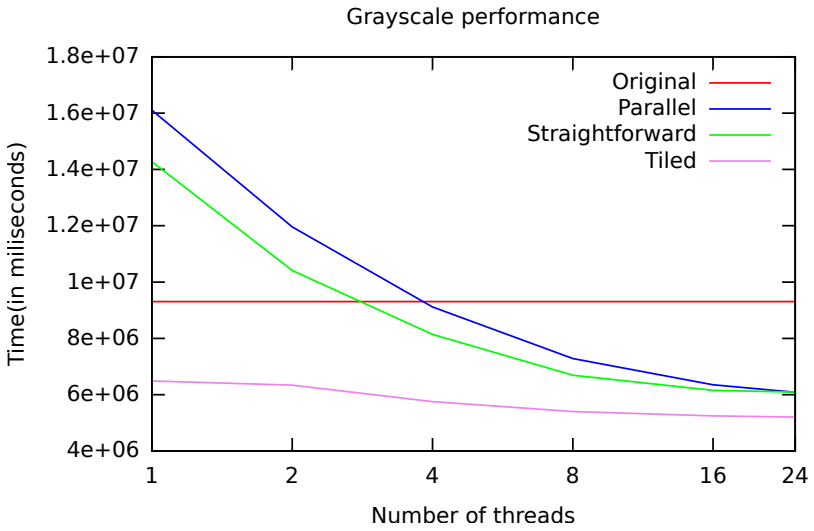


Figure 7.9: *Grayscale kernel* performance results.

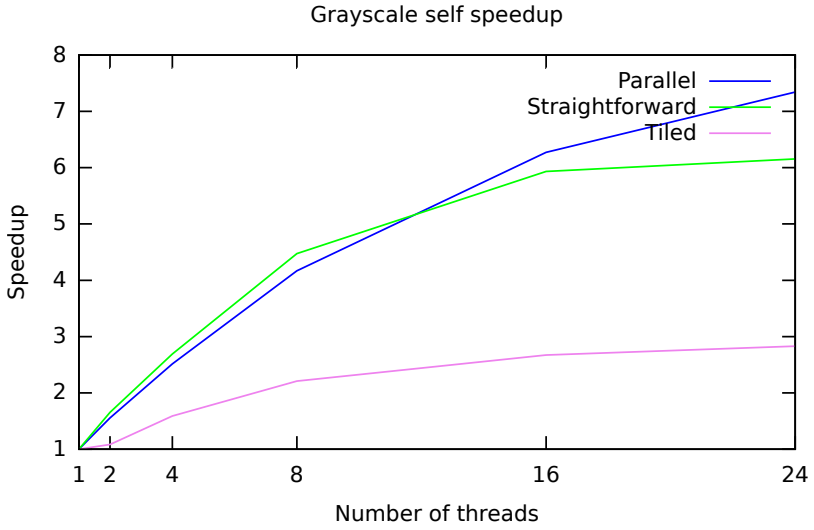


Figure 7.10: *Grayscale kernel* self speedup results.

For kernel QR, Matrixmul and the Grayscale, we obtained benefits from using speculative parallelization.

The tiled skeleton, provided better performance results than the other skeletons, except for adi-polybench, were the straightforward parallelization pattern got better results. On the other hand, the performance obtained by the straight-

forward parallelization skeleton over the parallel code skeleton was noticeable on all the benchmarks.

In the grayscale kernel, the tiled pattern got speedup over the original version without parallelization (See fig 7.9). This is because this kernel traverses a big image, by columns instead of by rows. This results in poor data locality.

The speedup results obtained for all the skeletons with a small number (less than 8) of threads are similar. On the other hand, the performance results were different in favor of the tiled code skeleton.

As the number of threads increases the speedup increase ratio drops, especially for the skeletons which obtain better performance results. For the selected problem sizes it's impossible to obtain speedup after a certain number of threads, since the time spent on the sequential fraction of the program –Transformation selection, instrumentation, backup, rollback, original version execution– started dominating the time spent on the parallel fraction. It is expected that, as the problem size increases, the parallel region fraction of the program execution will increase, leading to better speedups.

7.2 Conclusions over the results

The new code skeletons purposed in this thesis succeed in improving the performance over the original code transformations available.

Tiling results a key transformation to achieve good performance. For most benchmarks, applying the tiling transformation achieved better performance than the rest of the code skeletons. One advantage of this code skeleton is that it can counteract the penalty imposed by selecting a loop transformation with poor data locality.

Also the straightforward parallelization skeletons outperformed the previous existing code skeleton.

Nevertheless, the strategy implemented in VMAD is limited by the time spent performing the memory backup. This penalty can ban any benefit obtained by using speculative parallelization.

Chapter 8

Conclusions

The work done in this thesis contribute to the topic of speculative parallelization, by extending and improving VMAD, giving support for new transformations and enabling new optimizations.

We extend the number of transformations available at runtime with two new code versions:

- One reflecting the original code with the outermost or second loop parallel. Since the original execution order is kept, the verification code executed on each iteration is noticeably reduced.

- To enable the tiling transformation, we purpose a new code skeleton. The main purpose of this transformation is to increase the data locality by reordering the iteration execution such that iterations from the outer loops are executed before completing iterations from the innermost loops. Additionally, we purposed a mechanism for adapting the sizes of the tiles at runtime.

This new code skeletons were implemented as extensions to the static component of VMAD. Also, the runtime system and the dependence analysis was extended to be aware of this new skeletons.

We successfully parallelized loops using this new code skeletons. We evaluated the effectiveness of this contributions by running a set of benchmarks. The results obtained confirm our idea that improving data locality and reducing the amount of verification code are key to obtain good performance when applying speculative parallelization. We obtained promising results using the tiling code skeleton, which outperformed the results obtained using other code skeletons, or executing the original code, for several benchmarks. The new code skeletons successfully outperformed the previous code skeleton available in VMAD by improving data

locality and reducing verification code. Nevertheless, for some codes it was impossible to achieve a performance benefit by using speculative parallelization; the penalty imposed by the memory backup limits this strategy.

To reduce the impact of the verification code, embedded in the code skeletons, this thesis purposes an analysis to avoid unnecessary verification when there exists a linear relationship between a value to be verified and a value known to be linear. This analysis captures several common cases where the amount of verification code can be reduced: Loops where loop unroll was applied, accessing different fields of a `struct`, reading several fields of the same array. This analysis is implemented as an LLVM optimization pass, whose results are used at compile time by VMAD.

Many aspects we expect to enhance in our system are presented in the next section.

8.1 Future work

The VMAD framework succeeds in enabling the polytope model to general purpose codes, that require dynamic analysis and transformations. Nevertheless, there are several topics that have to be revisited to achieve good performance.

One strategy for improving the performance of the frameworks, is to reduce the time spent on the memory backup launched before each optimized chunk. Currently, this is performed by several calls to the `memcpy` function. Another approach would be to perform the copy of each memory location as needed, inside each thread. Additionally, if a rollback is needed, restore the memory in parallel.

Improving the code generation of VMAD is key to achieve good performance. To generate the optimized code, we are using a naive algorithm to select a suitable code transformation. It is mandatory to develop an algorithm to select a suitable code transformation which takes into account efficiency issues, such as data locality.

The number of transformations available at runtime is limited by the code skeletons generated at compile time. An interesting strategy is to generate dynamically a new code version by composing different loop basic blocks. In this way, we could be able to build loop nest whose structure is not preserved by the transformation, such as loop fission.

From scalability measurements obtained from the results, we can conclude that after a certain point assigning more cores to the execution of an optimized chunk does not result in a noticeable performance improvement. To obtain a better use of the machine resources, we could limit the number of cores assigned to the chunk execution, while other cores perform other tasks, such as anticipating a memory backup or searching for better transformations. On the other hand, we could optimize VMAD to reduce the power consumption by shutting down processor cores when increasing the parallelism doesn't provide performance benefits.

Appendices

Appendix A

Raw results

In this section we present the results obtained from the different benchmark runs. The measurements are in μ-seconds (10^{-6} of a second) and correspond to the average measurement of three runs.

adi-polybench			
Original code execution time: 9310370			
Backup average time: 7688955			
Threads	Parallel skeleton	Straightforward skeleton	Tiled skeleton
1	37066182	25221893	37189056
2	22763116	16982922	23890959
4	15753914	12847902	15775522
8	12191040	11709426	13325941
16	11847743	11633639	12055411
24	11551530	11358191	11749281

QR decomp			
Original code execution time: 16398426			
Backup average time: 307030			
Threads	Parallel skeleton	Straightforward skeleton	Tiled skeleton
1	103256160	45905225	26930602
2	55049555	26052874	15188196
4	29923610	15280970	9708889
8	16867327	9396928	8298510
16	10533665	6607966	6579476
24	8366709	5674621	5888677

matrixmul			
Original code execution time: 16086987			
Backup average time: 282860			
Threads	Parallel skeleton	Straightforward skeleton	Tiled skeleton
1	64185481	31992624	24514220
2	35576364	19105865	11352990
4	20026163	11642170	7535176
8	11772054	7507076	5754477
16	7835561	5557538	5027725
24	6508784	4905244	4690190

grayscale			
Original code execution time: 9304649			
Backup average time: 4502757			
Threads	Parallel skeleton	Straightforward skeleton	Tiled skeleton
1	16104522	14280905	6487343
2	11957698	10409356	6337701
4	9113262	8137345	5752809
8	7285303	6688697	5400806
16	6352299	6151061	5245322
24	6083029	6091061	5203952

Gauss 2d filter					
Original code execution time: 12609505					
Backup average time: 13017175					
Threads	Parallel skeleton	Straight skeleton	Tiled skeleton	Parallel with reversal	Tiled with reversal
1	30960145	27885923	20908908	21031597	19855323
2	26714235	24260954	18231533	18079417	18299501
4	21454785	20103593	17448212	16734101	16637507
8	19023425	18463514	16712471	16677955	16124230
16	17386607	17873544	16088021	15873514	15584978
24	17526994	17651687	16573776	16548641	15183278

Bibliography

[1] Utpal K. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[2] 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.

[3] 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.

[4] 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.

[5] Uday Kumar Reddy Bondhugula. *Effective automatic parallelization and locality optimization using the polyhedral model*. PhD thesis, Ohio State University, Columbus, OH, USA, 2008. AAI3325799.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988.

[13] P. Feautrier. Dataflow analysis of scalar and array references. *Int. J. of Parallel Programming*, 20(1):23–53, 1991.

[14] 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.

[15] Paul Feautrier. Some efficient solutions to the affine scheduling problem, part 2 : multidimensional time. *International Journal of Parallel Programming*, 21(6), 1992.

[16] GOMP — An OpenMP implementation for GCC - GNU Project. `http://gcc.gnu.org/projects/gomp`.

[17] 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.

[18] Alexandra Jimborean. *Adapting the polytope model for dynamic and speculative parallelization*. Phd thesis, Université de Strasbourg, September 2012.

[19] Alexandra Jimborean, Philippe Clauss, Jean-François Dollinger, Vincent Loechner, and Juan Manuel Martinez Caamaño. Dynamic and speculative polyhedral parallelization using compiler-generated skeletons. *International Journal of Parallel Programming*, 2014. to appear.

[20] Alexandra Jimborean, Philippe Clauss, Juan Manuel Martinez, and Aravind Sukumaran-Rajam. Online dynamic dependence analysis for speculative polyhedral parallelization. In Felix Wolf, Bernd Mohr, and Dieter Mey, editors, *Euro-Par 2013 Parallel Processing*, volume 8097 of *Lecture Notes in Computer Science*, pages 191–202. Springer Berlin Heidelberg, 2013.

[21] 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.

[22] 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.

[23] 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.

[24] Chris Lattner and Vikram Adve. LLVM language reference manual. `http://llvm.org/docs/LangRef.html`.

[25] 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.

[26] 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.

[27] 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.

[28] Polybenchs 1.0. http://www-rocq.inria.fr/pouchet/software/polybenchs., 2010.

[29] Louis-Noël Pouchet. FM: the Fourier-Motzkin library. `http://www.cse.ohio-state.edu/ pouchet/software/fm`.

[30] 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.

[31] Benoit Pradelle. *Static and dynamic methods of polyhedral compilation for an efficient execution in multicore environments*. PhD thesis, University of Strasbourg, Strasbourg, France, 2011.

[32] 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.

[33] 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.

[34] 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.

[35] Rosetta Codes. `http://rosettacode.org/wiki/Rosetta_Code`.

[36] Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, Inc., New York, NY, USA, 1986.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] Sergio Yovine, Assayad Ismail, Francois-Xavier Defaut, Marcelo Zanconi, and Ananda Basu. Formal approach to derivation of concurrent implementations in software product lines. Technical Report TR-2007-12, Verimag Research Report.

[42] 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.