ABSTRACT

Title of dissertation: AUTOMATIC PARALLELIZATION OF AFFINE LOOPS USING DEPENDENCE AND CACHE ANALYSIS IN A BINARY REWRITER

Aparna Kotha, Doctor of Philosophy, 2013

Dissertation directed by: Professor Rajeev Barua
Department of Electrical and Computer Engineering

Today, nearly all general-purpose computers are parallel, but nearly all software running on them is serial. Bridging this disconnect by manually rewriting source code in parallel is prohibitively expensive. Automatic parallelization technology is therefore an attractive alternative.

We present a method to perform automatic parallelization in a binary rewriter. The input to the binary rewriter is the serial binary executable program and the output is a parallel binary executable. The advantages of parallelization in a binary rewriter versus a compiler include (i) compatibility with all compilers and languages; (ii) high economic feasibility from avoiding repeated compiler implementation; (iii) applicability to legacy binaries; and (iv) applicability to assembly-language programs.

Adapting existing parallelizing compiler methods that work on source code to work on binary programs instead is a significant challenge. This is primarily because symbolic and array index information used in existing compiler parallelizers is not available in a binary. We show how to adapt existing parallelization methods to achieve equivalent parallelization from a binary without such information. We have also designed a *affine cache reuse model* that works inside a binary rewriter building on the parallelization techniques. It quantifies cache reuse in terms of the number of cache lines that will be required when a loop dimension is considered for the innermost position in a loop nest. This cache metric can be used to reason about affine code that results when affine code is transformed using affine transformations. Hence, it can be used to evaluate candidate transformation sequences to improve run-time directly from a binary.

Results using our x86 binary rewriter called SecondWrite on a suite of dense-matrix regular programs from *Polybench* suite of benchmarks shows an geomean speedup of 6.81X from binary and 8.9X from source with 8 threads compared to the input serial binary on a x86 Xeon E5530 machine; and 8.31X from binary and 9.86X from source with 24 threads compared to the input serial binary on a x86 E7450 machine. Such regular loops are an important component of scientific and multi-media workloads, and are even present to a limited extent in otherwise non-regular programs.

Further in this thesis we present a novel algorithm that enhances the past techniques significantly for loops with unknown loop bounds by guessing the loop bounds using only the memory expressions present in a loop. It then inserts run-time checks to see if these guesses were indeed correct and if correct executes the parallel version of the loop, else the serial version executes. These techniques are applied to the large affine benchmarks in SPEC2006 and OMP2001 and unlike previous

methods the speedups from binary are as good as from source. We also present results on the number of loops parallelized directly from a binary with and without this algorithm. Among the 8 affine benchmarks among these suites, the best existing binary parallelization method achieves an geomean speedup of 1.33X, whereas our method achieves a speedup of 2.96X. This is close to the speedup from source code of 2.8X.

AUTOMATIC PARALLELIZATION OF AFFINE LOOPS USING DEPENDENCE AND CACHE ANALYSIS IN A BINARY REWRITER

by

Aparna Kotha

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2013

Dedication

Dedicated to my family

To my *father* for the *inspiration* to begin

To my *mother* for the *motivation* to continue

To my *sister* for the *long chats* to keep me going

To my *husband* for lots of *love and stability* to finish

Acknowledgments

I owe my gratitude to all the people who have made this thesis possible and because of whom my graduate experience has been one that I will cherish forever.

First and foremost I'd like to thank my advisor, Dr. Rajeev Barua for giving me an invaluable opportunity to work on challenging and extremely interesting projects over the past six years. In times when I felt lost, he was always there to give me a fresh perspective and look at the problem. His words have inspired me in many ways.

I would also like to thank my committee members, Dr. Gang Qu, Dr. Shuvra Bhattacharyya, Dr. Donald Yeung and Dr. Alan Sussmann for agreeing to be on my committee. Your feedback during my proposal and dissertation talks is invaluable.

My labmates through the years at the SCAL lab, University of Maryland, College Park have taught me many things related to academics and many a times otherwise. I would specially like to thank Kapil Anand, Khaled ElWazeer, Timothy Creech, Matthew Smithson, Greeshma Yellareddy, Mincy Mathew, Kyungjin Yoo, Fady Ghanim and many more. Thank you for all the time we spent together and for everything you taught me.

I would also like to acknowledge the help and support from some of the staff members at the ECE department. Melanie Prange, Carrie Hilmer, Kristie Little, Maria Hoo and everyone at the helpdesk and ECE graduate office for being there to solve my non-thesis related problems that arose during the jouney, hence making it a more enjoyable journey.

I owe my deepest thanks to my parents - my mother and father who have always stood by me and supported me all along my career. They have always encouraged me to live my dreams. Their love for me is unparalleled.

I'd like to express my gratitude to my housemates through the years Suma Babu, Purva Ghate, Sarah Khurian, Aparajita Bal and Khushboo Maheshwari for their friendship and support.

Lastly the journey would not have been any fun without my friends. Thanks all of you for all the crazy and fun times: Kapil Anand, Nitesh Shroff, Shalabh Jain, Kaustubh Jain, Harita Tenneti, Rashi Jain, Srikanth Vishnubhotla, Jishnu Keshavan, Surabhi Chouhan, Ishwar Bhati, Raghuraman Gopalan, Satish C Reddy, Silpa Billa, Anshu Sarje and many others. Special thanks goes out to my friends Vijayalakshmi Ethirajulu, Ashwin Rishinaramangalam, Ashwin Pejaver and Sreekanth Annapureddy from IITM days with whom a vacation a year was compulsary and those times are part of my best memories in the last few years.

It is impossible to remember all, and I apologize to those I've inadvertently left out.

Lastly, thank you all and thank God!

Table of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Introduction

Since about 2004, semiconductor trends show that the astonishing improvements in clock speeds over the last few decades have come to end. However improvements in silicon area, as per Moore's law, are still being realized. As a natural consequence of these trends, microprocessor vendors have turned to multi-core processors to remain competitive. For *e.g.*, Intel Corporation has replaced its Pentium line of uniprocessors with the Intel *Core* processor family, virtually all of which have multiple cores. AMD Corporation offers the Athlon Dual-core and Phenom Quad-core processors. By the end of 2009, multi-cores accounted for 100% of all new desktop and notebook processors [1]. The CPU road maps of both Intel and AMD show trends towards further multiple cores. Today even handheld devices such as mobile phones and tablets have multiple cores in them. Embedded system processors are being realized with multiple cores in them.

With the advent of multi-core hardware in every sphere it is essential to have parallel software to run on them. There are three methods to obtain parallel software:

- The first way to obtain parallel software is to use parallel language directives such as OpenMP [2, 3] to implicitly specify parallelism using comments in high-level language programs.

- The second way to obtain parallel software is to write programs in an explicitly parallel manner. This is done using a set of APIs, such as MPI [4], posix complaint *pthreads* or Intel's TBB, to extend existing languages such as C, C++ and Fortran.

- The third way to obtain parallel software is to use an automatic parallelizing compiler. This is a tool that takes as input serial code and produces as output parallel code

The first two methods need human intervention whereas the third method is an automatic tool. Although the use of explicitly parallel programming using the first two methods is increasing, their adoption has been slowed by the following factors:

- Huge amounts of serial code represent most of the world's programs.

- Rewriting code manually in parallel is time consuming and expensive.

- Dearth of engineers trained in parallel programming and algorithms.

- Parallel programming productivity per line of code is lower than for serial [5].

For this reason, except for the most performance-critical code, it is not likely that most of the world's existing serial code will be rewritten in parallel.

The third way to obtain parallel software using an automatically parallelizer overcomes the above-mentioned drawbacks of explicitly parallel code. Indeed, since the introduction of parallel machines in the early 1970s, many strides have been made in parallelizing compiler technology. Most efforts to date have focused on parallelism in loops, primarily in regular, dense-matrix codes. In particular, techniques have been developed for parallelizing loops with array accesses whose indices are affine (linear) functions of enclosing loop induction variables [6]. This work is particularly interesting as most scientific and multi-media codes are affine and the maximum run time is spent in these loops. Hence parallelizing these loops can result in significant speedups. The advantages of automatic parallelization over explicit parallel programming are:

- No need of human intervention; hence large amounts of programs can be parallelized
- Less prone to error since manual intervention is not there
- Can be integrated within an compiler framework since it can be constructed to work on compiler IR

In this thesis we present methods to implement automatic parallelization inside a binary rewriter, instead of a compiler. A binary rewriter is a software tool that takes a binary executable program as input, and outputs an improved executable as output. In our case, the input code will be serial, and the output will be parallel code. *As far as we know, there are no existing methods for automatic parallelization in a static binary rewriter. Further, there are no existing binary automatic parallelization tools (static or dynamic) that perform affine-based parallelization.* Parallelization in a binary rewriter has several advantages over parallelization in a compiler:

- **Works for all compilers and languages** A parallelizer in a binary rewriter works for all binaries produced using any compiler from any source language. This compatibility is a huge practical advantage versus a compiler implementation.

- **No need to change software tool chains** A binary rewriter is a simple add-on to any existing software development tool chain. Developers and their companies, typically very resistant to changing the tool chains they are most familiar with, will not have to. This is important since many existing compilers do not perform automatic parallelization.

- **High economic feasibility** A parallelizer in a binary rewriter needs to be implemented only once for an instruction set, rather than repeatedly for each compiler.

- **Applies to legacy code** Legacy binaries for which no source is available, either because the developer is out of business, or the code is lost can be parallelized using a binary rewriter. No compiler can do this.

- **Works for assembly-language programs** A binary rewriter, unlike a compiler can parallelize assembly code, regardless of whether it is part of the program with inlined assembly or all of it. Assembly code is used sometimes to write device drivers, code for multi-media extensions, memory-mapped I/O, and time-critical code portions.

- **Can perform platform-specific tuning** Since a binary rewriter can tune the output program for the particular platform it is executing on, it is possible to tune the same input executable differently for different platforms which share the same ISA, but may have widely different runtime costs. For example, we already choose the best barrier and broadcast mechanism for the end-user platform, and are investigating specific optimizations for the instruction set enhancements and instruction latencies of that platform.

- **Can be used by end user of software** Unlike compiler-provided parallelization which can only be done by the software developer, parallelization in a binary rewriter can be done by the end user of the system depending upon his or her specific needs, constraints and environment.

The above advantages argue that *it is useful to provide automatic parallelization in a binary rewriter*, despite compiler implementation being possible. By allowing automatic parallelization to be done on *arbitrary binaries from any source*, we hope to make this technology *universal, accessible, portable, customizable to the end-user's platform, and usable by any computer user, not just developers.* In this vision, our hope is that the parallelizing rewriter will be a transparent utility that is automatically triggered for all programs at install-time.

Our approach to automatic parallelization is **not** to invent entirely new parallelization methods, but to investigate how best to adopt ideas from existing compiler methods to a binary rewriter. This adoption is not trivial, since binary rewriters pose challenges not present in a compiler, including primarily, the lack of high-level information in binaries. Parallelizing compilers rely on symbolic information, for identifying arrays, affine function indices, and induction variables; for renaming to eliminate anti and output dependencies; and for pointer analysis to prove that memory references cannot alias, allowing their parallel execution. Binaries lack symbolic information, making all these tasks more difficult. *A central contribution of this proposal is parallelization methods in a binary rewriter that can work effectively without using any symbolic or array index information.*

On the flip side, binary rewriters also enjoy an advantage compared to a compiler: they have access to the entire program including library code. The need for separate compilation – an important practical requirement – has meant that commercial compilers typically have access to only one file at a time. For large programs with many files, this represents a tiny sliver of the total code. Whole-program parallelization is important since parallelization in one procedure may inhibit parallelization at containing or contained procedures, and moreover, parallel code blocks may be in different files. Currently we look at loop-level parallelism as most execution time is spent in loops.

Of course, we recognize that parallelizing affine programs is only one step towards the goal of parallelizing all programs, albeit an important one. Many programs have non-affine parallelism, and others have little or no parallelism. Our techniques can be effectively combined with non-affine techniques to parallelize a larger set of programs. This work should be seen as what it is: a first successful attempt to parallelize binaries using affine analysis, rather than the last word. We hope to open up a new field of research with this significant step.

1.2 Contributions

The main contribution of this thesis is to present an affine automatic parallelizing tool inside a static binary rewriter. The main contribution of this work can be divided into three major parts:

- The first contribution of this work is to present how dependence analysis can be performed directly on binary code without the presence of any symbolic information. Traditionally all automatic parallelizing compilers from source rely on symbolic information to know the size and dimensions of arrays, the array indices etc. However, no such information is present in a compiled binary. We present methods to still recover dependences using only binary code. This is done by recovering array address variables in the form of linearized multi-dimensional expressions from binary code and then solving them using dependence tests within the loop bound regions.

- The second contribution of this thesis is to parallelize affine loops from binaries even when loop bounds are run-time dependent. Using the techniques developed in the first contribution, the parallelization of such loops is highly limited since solving linearized multi-dimensional expressions in the infinite region generates more dependencies limiting parallelization. Hence, we have developed techniques to guess bounds for the loop within which it is parallel. We then insert run-time checks to see that the actual loop bound is lower than or equal to the one we guessed before executing the parallel version of the loop. In this way we increase the scope of affine loops parallelized. In our results we show that we can effectively parallelize the affine benchmarks from SPEC 2006 and OMP 2001 using this techniques.

- The third contribution of this thesis is a technique by which the cache reuse of affine loops can be calculated directly from binaries using the linearized multi-dimensional expressions for address variables obtained directly from binary code. Using the cache reuse metric we can reason about the loop ordering in a loop nest and transform it to maximize cache reuse and increase coarse-grain parallelization.

1.3 Organization of Dissertation

This thesis is further organized as follows:

In **chapter 2** we introduce automatic parallelization by presenting the past literature in both the traditional and polyhedral schools of study. **Chapter 3** presents a method to parallelize affine loops directly from binary code by recovering address expressions for the affine access followed by a method to parallelize loops when their bounds are run-time dependent in **chapter 4**. In **chapter 5** we present a method to calculate the *cache reuse metric* directly from binary and use it to transform the loop. The implementation details of our automatic parallelizer are presented in **chapter 6** followed by results in **chapter 7** and related work in **chapter 8**. Conclusions of the work and suggested future directions are presented in **chapter 9**.

Chapter 2

Automatic Parallelization

In this chapter we briefly describe the background of automatic parallelization as has been described in the vast literature in this area.

An automatic parallelizer is a tool that takes as input serial code and produces as output parallel code. Traditionally such tools have been designed for source code with affine loops in them. Affine loops are loops that contain array indices that are a linear combination of the induction variables of the loop nest *i.e.*loops containing $\texttt{A}[\texttt{i}][\texttt{j}], \texttt{A}[\texttt{i} + \texttt{3j} + \texttt{5}][\texttt{4i} + \texttt{2j} + \texttt{10}], \texttt{A}[\texttt{i}][\texttt{i}]$ are all affine whereas loops containing $\texttt{A}[\texttt{i}/\texttt{2}]$ or $\texttt{A}[\texttt{i}^2]$ are not.

Existing methods in affine analysis and parallelization can be categorized into two broad classes: traditional methods and those based on the polyhedral model.

2.1 Traditional methods

Traditional methods [7] [8] [9] [10] [11] [6] [12] are those which are based on modeling loops as the units of consideration, where matrices are used to model most concepts including affine indices, iteration vectors, dependence vectors and loop transformations. Methods have been proposed for deciding what order of transformations should be applied and in what order.

The traditional methods are divided into three steps: representation of the dependencies; decision algorithms to transform loop nests and then methods to generate target code. Further we present these three steps briefly describing the methods that have been proposed for each of them in the past. These have been presented to describe the techniques used in parallelizing compilers that are built on the traditional model.

2.1.1 Representation of dependence information

In this section we describe how dependence information is represented in the traditional methods.

Traditional models predominantly represent the dependence information either as distance or direction vectors [13] [7] [8]. Distance vectors for every loop contain an entry for each loop nest where each entry is the step of the loop dependence in that induction variable.

The formal definition of the *distance vector* from [6] is:

Suppose that there is a dependence from statement $\mathtt{S_1}$ on iteration $\mathtt{i}$ of a loop nest of $\mathtt{n}$ loops to statement $\mathtt{S_2}$ on iteration $\mathtt{j}$, then the *dependence distance vector* $\mathtt{d(i,j)}$ is defined as a vector of length n such that $\mathtt{d(i,j)_k} = \mathtt{j_k} - \mathtt{i_k}$ where:

– $\mathtt{i}$ and $\mathtt{j}$ are both iteration vectors of $\mathtt{n}$ elements

– Distance vector measures the difference between $\mathtt{i}$ and $\mathtt{j}$ in the iteration space

Direction vectors represent the same information as the distance vector when less precision is required or available. The formal definition of *dependence direction vectors* from [6] is:

Suppose there is a dependence from statement $\mathtt{S}_1$ on iteration $\mathtt{i}$ of a loop nest of $\mathtt{n}$ loops to statement $\mathtt{S}_2$ on iteration $\mathtt{j}$, with a distance vector $\mathtt{d}(\mathtt{i},\mathtt{j})$, then the *dependence direction vector* $\mathtt{D}(\mathtt{i},\mathtt{j})$ is defined as a vector of length $\mathtt{n}$ such that

$$\begin{aligned} \mathtt{D}(\mathtt{i},\mathtt{j})_{\mathtt{k}} &= \text{“} < \text{”} \ \mathtt{if}\ \mathtt{d}(\mathtt{i},\mathtt{j})_{\mathtt{k}} > \mathtt{0} \\ &= \text{“} = \text{”} \ \mathtt{if}\ \mathtt{d}(\mathtt{i},\mathtt{j})_{\mathtt{k}} = \mathtt{0} \\ &= \text{“} > \text{”} \ \mathtt{if}\ \mathtt{d}(\mathtt{i},\mathtt{j})_{\mathtt{k}} < \mathtt{0} \end{aligned} \tag{2.1}$$

Distance/Direction vectors are calculated using standard linear algebraic methods described in literature. These include the Greatest Common Divisor (GCD) test [14, 15], Banerjee's inequalities [15], Single Index and Multiple Index Tests [13, 7], multidimensional GCD [15], the delta test [16] and the omega test [17].

Every loop nest has a set of distance or direction vectors associated with it that represent the dependencies present in the loop nest.

2.1.2 Decision algorithms to transform loops

Most affine transformations, their legality conditions, and code generation methods have been known for many years. However the problem of finding an optimal order of transformations is very challenging. In this section we present some of the decision algorithms that have been presented in traditional literature in the past.

Most work in traditional methods, for example by M.J Wolfe [7] describes the affine transformations independently, but does not provide a decision algorithm for deciding when to apply which transformations, and in what order.

Studies from the University of Illinois from the Polaris group include [9] [18] [19] [20]; The Polaris group extends Kap, an automatic parallelizer, and then use it to parallelize the Perfect Benchmarks. Their target architecture is Cedar, a shared-memory parallel machine with cluster memory and vector processors. Their work focuses on detecting parallelism via array and scalar analysis. Their inter-procedural analysis results de facto from inlining or is performed by hand.

Wolf & Lam's research in Unimodular transformations [21] [8] and later work by Kathyrn McKinley [12] describe how transformations can be traded in a traditional model. Wolf & Lam present a method to correctly model and apply multiple unimodular transformations to affine code to discover coarse grain parallelism minimizing communication. Unimodular transformations unify loop interchange, loop skewing and loop reversal. A unimodular matrix has three important properties. First, it is a square, meaning that it maps an n-dimensional space to an n-dimensional space. Second, it has all integral components, so it maps integer vectors to integer vectors. Third, the absolute value of its determinant is one. Due to these properties, the product of two unimodular transforms is unimodular and the inverse of a unimodular matrix is unimodular. The work by Wolf and Lam looks at transformations in isolation, and does not present a decision algorithm by which multiple transformations may be applied. Later work by SUIF [22] [23] includes interprocedural transformations. McKinley's work [12] builds on some ideas from the unimodular transformation algorithms. This work discovers coarsest grain parallelism and preserves data locality. McKinley's algorithm can take advantage of known loop bounds to more precisely compute locality and granularity of parallelism, and is very efficient. McKinleys work uses a simple cache model to model the reuse in a loop, whereas most earlier work do not have this. The reuse is calculated once and then the algorithm tries to achieve it. This work integrates loop fusion, loop fission and loop tiling.

2.1.3 Code Generation

In traditional model code generation is an immediate effect of the loop transformations applied to every loop. Using any of the decision algorithm described above, the transformation order is decided; then the loop transformations are applied to the loop in the same order generating the final code.

2.2 Polyhedral methods

Polyhedral methods [24] [25] [26] [27] [28] [29] are the second class used for affine analysis and automatic parallelization. They represent each statement in an affine loop separately as a point in an iteration domain. After this is done decisions to transform the loop are taken and parallel code is generated for the same.

Polyhedral methods have the following three advantages over traditional methods:

- First, polyhedral models handle imperfectly nested loops seamlessly in their model.
- Second, they are able to model dependence between every dynamic instance in the loop.

- Third, complex affine transformations can be modeled as scheduling functions, which in a few instances, can discover multiple traditional transformations in one step;

Traditional methods have the following advantages over the polyhedral methods:

- Their worst-case complexity is in the order of polynomial complexity against the exponential complexity of polyhedral methods.
- Their implementation complexity is also lower than the polyhedral methods.
- These methods are scalable to large programs running into millions of lines of code.

Similar to the traditional model, the polyhedral model has three steps: representation of the program in the polyhedral model; decision algorithms to transform the loop; and then generate target code.

2.2.1 Representation of Dependence information

Dependence information is represented differently in the polyhedral and traditional models. In this section we present the techniques used for representing dependence information in the polyhedral methods.

In the polyhedral model, each statement in an affine loop is separately represented as a point in an iteration domain [30]. This is different from traditional affine analysis where the loop iterations are generally considered indivisible in most scenarios. The points in the iteration domain are defined by constraints arising from loop bounds, and represented mathematically. Polyhedral model represents this information for every dynamic instance of every statement in the loop. They represent dependencies over the iteration space in a dependence polyhedron [29] [28] in a d-dimensional space, where d is the nesting depth of the loop in question.

2.2.2 Decision algorithm to transform loops

In this section we present decision algorithms of polyhedral methods.

In this step, the program represented mathematically in the polyhedral model is transformed. In the polyhedral framework, a transformation is a set of affine scheduling functions each of which maps each run-time statement instance to a logical execution date [28] [27] [26] [25] [24]. Transformations are applied to satisfy some optimization need such as improved cache locality or coarser granularity of threads, which can be added on via heuristic or other methods.

The decision algorithms in polyhedral methods are very complex in terms of implementation and computational complexity. The computational complexity of the decision algorithms present in polyhedral models is exponential in the order of the number of statements present in the loop body. Further, this algorithm are memory intensive and combined with their worst-case run-time, suffer from scalability problems.

2.2.3 Code Generation

Finally code transformations map the program representation along with the scheduling functions to output code for the target machine. Code generation relies upon syntax tree construction schemes that consist of a recursive application of domain projections and separations [31] [28]. The final code is deduced from the set of constraints describing the polyhedra attached to each node in the tree.

2.3 Conclusions

In balance we decided to implement our technologies in the traditional model. However, they can be adapted to the polyhedral model, as we will present in section 6.4 of chapter 6.

Chapter 3

Dependence Analysis

3.1 Calculating dependence vectors

The first challenge in parallelizing binaries is in calculating distance/direction vectors. In chapter 2 we have defined distance/direction vectors. In this chapter we first show how these are calculated from source code, and then present our method to calculate the same directly from binary code without any source level information.

3.1.1 From source

In this we section overview the strategy to calculate dependence vectors in the form of distance or direction vectors from affine loops (source-code loops containing array accesses whose indices are affine (linear) functions of enclosing loop induction variables). For example, if `i` and `j` are loop induction variables, then array accesses $\mathtt{A}[\mathtt{i}]$ and $\mathtt{A}[2\mathtt{i}+\mathtt{j}+3][\mathtt{i}-3\mathtt{j}+7]$ are affine, whereas $\mathtt{A}[\mathtt{i}/2]$ is not. We will present the techniques first from the source and then we will adapt it from a binary in section 3.1.2. The source-level techniques reviewed in this section are well documented in the literature of affine loop parallelism. These are presented in chapter 8.

To understand how parallelization can be done for affine-based loops; consider that dependencies between the instructions limit the parallelism that we can extract in code. For loops, loop-carried dependencies are the major inhibitors of parallelism,

```
for i from $lb_i$ to $ub_i$
  for j from $lb_j$ to $ub_j$
    A[i+1,j] += A[i,j] + 1

  end for
end for
```

(a) True-loop carried dependence

```
for i from $lb_i$ to $ub_i$
  for j from $lb_j$ to $ub_j$
    A[i,j] += A[i,j+2] + 1

  end for
end for
```

(b) Anti-loop carried dependence

```
for i from $lb_i$ to $ub_i$
  for j from $lb_j$ to $ub_j$
    A[i,j] = .........
    A[i-2,j] = .........
  end for
end for
```

(c) Output-loop carried dependence

Figure 3.1: Loop-Carried Dependencies in Source Code

and occur when the next loop iteration cannot be initiated before some previous set of loop iterations has completed. Just like scalar dependencies, loop-carried dependencies can be classified into three types: true, anti, and output loop-carried dependencies. Figure 3.1 shows an example of each type.

As in existing work, based on the formulation in [15], a dependence vector($\vec{D}$) for loops is defined as an n-dimensional vector, where n is the nesting depth of the loop. The most common formulation of a dependence vector is a *distance vector*, where each entry is the step of the loop dependence in that induction variable. For example, for the code in figure 3.1(a) the distance vector is calculated by looking for iterations $(\mathtt{i}_1, \mathtt{j}_1)$ and $(\mathtt{i}_2, \mathtt{j}_2)$ such that the two array accesses in the loop $\mathtt{A}[\mathtt{i}+1, \mathtt{j}]$ and $\mathtt{A}[\mathtt{i}, \mathtt{j}]$ access the same memory location, then their difference $(\mathtt{i}_2 - \mathtt{i}_1, \mathtt{j}_2 - \mathtt{j}_1)$ is the distance vector. The memory address that $\mathtt{A}[\mathtt{i}+1, \mathtt{j}]$ accesses in iteration $(\mathtt{i}_1, \mathtt{j}_1)$ is $\mathtt{A}[\mathtt{i}_1+1, \mathtt{j}_1]$ and the memory address that $\mathtt{A}[\mathtt{i}, \mathtt{j}]$ accesses in iteration $(\mathtt{i}_2, \mathtt{j}_2)$ is $\mathtt{A}[\mathtt{i}_1, \mathtt{j}_1]$. If both the array expressions access the same memory location, then $\mathtt{i}_1 + \mathtt{1} = \mathtt{i}_2$ and $\mathtt{j}_1 = \mathtt{j}_2$. Rearranging the terms:

$$\begin{aligned} \mathtt{i}_2 - \mathtt{i}_1 &= 1 \\ \mathtt{j}_2 - \mathtt{j}_1 &= 0 \end{aligned} \tag{3.1}$$

Hence, the distance vector for the loop in figure 3.1(a) is (1,0) indicating that there is a dependence in steps of 1 along `i`, whereas there is no dependence along `j`. Similarly, for the loop in figure 3.1(c) the distance vector is $\vec{D}$ = (2,0), indicating that there is a dependence in steps of 2 along `i`, whereas there is no dependence along induction variable `j`. Further, for the loop in figure 3.1(b) the distance vector is $\vec{D}$ = (0,2), indicating that there is a dependence in steps of 2 along induction variable `j`, and no dependence along induction variable `i`.

Linear systems of equation solvers are used to calculate distance vectors in literature. These include the GCD and Single Index Variable (SIV) test [15]. As shown above, the linear system of equations is obtained from the symbolic array index expressions and array declarations present in source code. Further when the distance cannot be found or is not deterministic, we can represent the dependence in loops by direction vectors [7] [6], a less precise formulation of distance vectors. Dependence vectors are vectors of loop nesting depth each entry of which are one of { =, <, >, * }.

3.1.2 From Binary

In this section we present our method for calculating dependence analysis from low-level code obtained from binary code, which does not contain any symbolic information or affine expressions. The analysis will be successful when the underlying access patterns are affine, even when the array indices needed for traditional dependence analysis for parallelization are absent, such as in binary code.

A source-code fragment and one of its possible binaries is shown in figure 3.2. The binary is shown in pseudo-code for comprehensibility, but actually represents machine code. Other binaries are also possible, but we will be able to illustrate the general principles of our method with this example. The binary code assumes that the array is laid out in row-major form, with the address of $\mathtt{A}[\mathtt{i},\mathtt{j}]$ being computed as:

$$\&\mathtt{A}[\mathtt{i},\mathtt{j}] = \mathtt{A} + \mathtt{i} \times \mathtt{size\_j} + \mathtt{j} \times \mathtt{elem\_size} \tag{3.2}$$

where elem_size is the size of an individual array element, and size_j is the size of the second array dimension, both in bytes. We assume row-major accesses to understand our techniques, but in no way are these techniques going to be effected if the code was arranged in a column-major format. If the code is arranged in a column-major format then the address computed in binary code will be of the form:

$$\&\mathtt{A}[\mathtt{i},\mathtt{j}] = \mathtt{A} + \mathtt{i} \times \mathtt{elem\_size} + \mathtt{j} \times \mathtt{size\_i} \tag{3.3}$$

Though our theory presently assumes row-major, it is correct for column-major codes as well.

To see how we parallelize the binary code, the following intuition is helpful: *it is a simple proof to show that for any affine array access, its address variable is*

Source Code

```
for i from $lb_i$ to $ub_i$
    for j from $lb_j$ to $ub_j$
        A[i,j] = A[i,j] + 1
    end for
end for
```

Binary Code

```
1       $reg\_lb_i \leftarrow lb_i$
2       $reg\_ub_i \leftarrow ub_j$
3       $i' \leftarrow lb_i * size\_j$          --(E)
4       $reg\_ub'_i \leftarrow ub_i * size\_j$
5 loopi:$reg\_lb_j \leftarrow lb_j$
6       $reg\_ub_j \leftarrow lb_j$
7       $j' \leftarrow lb_j * elem\_size$
8       $addr\_reg \leftarrow Base + i' + j'$  --(B)
9       $reg\_ub\_addr \leftarrow Base + i' + ub_j * elem\_size$
10 loopj:load reg $\leftarrow$ [addr_reg]
11      $reg \leftarrow reg + 1$
12      store [addr_reg] $\leftarrow$ reg
13      $addr\_reg \leftarrow addr\_reg + elem\_size$ --(A)
14      CMP $addr\_reg \leq reg\_ub\_addr$ --(C)
15      Branch if true to loopj
16      $i' \leftarrow i' + size\_j$            --(D)
17      CMP $i' \leq reg\_ub'_i$              --(F)
18      Branch if true to loopi
```

Figure 3.2: Example showing source code and it's binary code

provably always an induction variable in it's immediately enclosing loop. Of course, it is a *derived* induction variable [32], derived from the basic induction variables like `i` and `j` in source[1].

Analyzing the control flow graph that we obtain from a binary can help us recognize loops in a binary. Every back edge in the control flow graph defines a loop [32]. We know that the address of every affine access in the body of the loop is a derived induction variable. In the binary code in figure 3.2, `addr_reg` is the address register, which must be an induction variable since it came from an affine access in source. It would not be an induction variable if it was not an affine access in source. We must first check that all memory accesses in the loop are induction variables, for this loop to have been an affine loop in source. Starting from this address induction variable `addr_reg`, we can define the following six special statements in the binary ((A) to (F)) for every address variable in a loop that is an induction variable. These six statements will help us parallelize the binary, regardless of the exact binary code:

- **(A) – Address variable increment:** The rewriter searches for the increment of the address induction variable in the loop, and names it (A). See the example binary in figure 3.2 to find (A) to (F).
- **(B) – Address variable lower bound:** The incoming value of the address induction variable (`addr_reg` in the example) is its lower bound; it is marked (B).

[1]Basic induction variables are those, which are incremented by a constant every loop iteration. A derived induction variable `d` is of the form $\mathtt{d} = \mathtt{c}_1 \times \mathtt{i} + \mathtt{c}_2$, where i is a basic or derived induction variable with step `s`; hence `d` too is an induction variable with step $\mathtt{c}_1 \times \mathtt{s}$.

- **(C) – Address variable upper bound:** The upper bound comparison of the address variable for the loop-ending branch identifies the upper bound of the address variable. It is searched for and marked (C).

- **(D) – Outer loop induction variable increment:** We check if statement (B)'s right-hand side value contains another induction variable. If it does, it is distinguished as the induction variable of the next-outer loop. In the example it is `i'`. The increment which reveals this induction variable is marked (D).

- **(E) – Outer loop induction variable lower bound:** The incoming value of the outer loop induction variable (`i'` in the example) is its lower bound; it is marked (E).

- **(F) – Outer loop induction variable upper bound:** The upper bound comparison of the outer loop induction variable for the loop-ending branch identifies the upper bound of the address variable. It is searched for and marked (F).

Statements (A) to (C) are for the inner loop; and (D) to (F) are for the outer loop, if present. For loops nested to depth three or more, additional statements can be identified (e.g. (G) to (I) and so on). These special statements *can be identified from almost any binary compiled from affine accesses*, regardless of its exact form. Recognizing statements (A) to (F) in the binary relies primarily on effective induction variable analysis, which is easy for registers in binaries. By the definition of an induction variable, once it is recognized, its increment (or set of increments) reveal the statements (A) and (D). The incoming values ((B) and (E)) immediately follow, as well as the exit conditions ((C) and (F)).

Our recognizer will recognize not only virtually all affine accesses written as such, **but also affine accesses through pointers**, since the pointers themselves will be induction variables. The only non-recognized case is when the constant increment of the induction variable is hidden by layers of indirection, e.g. when the constant is in a memory location, or when the induction variable is not virtual-register-allocated in the binary rewriter's intermediate representation, but we have found such cases to be extraordinarily rare.

Let us now describe our procedure to derive address variable equations directly from a binary.

For a two-dimensional loop, consider an affine memory reference in a binary loop inside loop dimension 2, where loop dimensions are counted from outermost (1) to innermost (2). Let `addr_reg` be the address variable of this memory reference, which must be an induction variable for an affine access. From the binary code shown in figure 3.2 we can define the address variable's lower bound value (RHS of (B)) as `Initial_addr_reg`. Since, this is the lower bound to the induction variable in the innermost loop, it will be a loop-invariant quantity in the innermost loop. If the access was indeed affine in the original source code (which we do not have access to), then from applying row-major ordering to the access, it must be the case that:

$$\texttt{Initial\_addr\_reg} = \texttt{Base} + \texttt{i} \times \texttt{size\_j} + \texttt{lb}_\texttt{j} \times \texttt{elem\_size} \tag{3.4}$$

In the binary, however, the above formula is not necessarily evident. What we do know is that for induction variable $\texttt{addr\_reg}$, from the definition of an induction variable with loop-entry value $\texttt{Initial\_addr\_reg}$, we get:

$$\texttt{addr\_reg} = \texttt{Initial\_addr\_reg} + \texttt{num\_j} \times \texttt{step}_\texttt{j} \tag{3.5}$$

where $\texttt{num\_j}$ is the number of iteration of the $\texttt{j}$ loop executed so far, and $\texttt{step}_\texttt{j}$ is the step of the induction variable. Substituting eq.(3.4) into eq.(3.5) we get:

$$\texttt{addr\_reg} = \texttt{Base} + \texttt{i} \times \texttt{size\_j} + \texttt{lb}_\texttt{j} \times \texttt{elem\_size} + \texttt{num\_j} \times \texttt{step}_\texttt{j} \tag{3.6}$$

We know that the number of iterations $\texttt{num\_i} = \texttt{i} - \texttt{lb}_\texttt{i}$. Hence, $\texttt{i} = \texttt{num\_i} + \texttt{lb}_\texttt{i}$. Substituting this in eq.(3.6) we get:

$$\texttt{addr\_reg} = \texttt{Base} + \texttt{lb}_\texttt{i} \times \texttt{size\_j} + \texttt{lb}_\texttt{j} \times \texttt{elem\_size} + \texttt{num\_i} \times \texttt{size\_j} + \texttt{num\_j} \times \texttt{step}_\texttt{j} \tag{3.7}$$

Since the first three terms of the RHS are constants, we rename their sum as $\texttt{Base}_\texttt{outer}$ since they are the loop-entry value of outer loop address variable. Hence:

$$\texttt{addr\_reg} = \texttt{Base}_\texttt{outer} + \texttt{num\_i} \times \texttt{size\_j} + \texttt{num\_j} \times \texttt{step}_\texttt{j} \tag{3.8}$$

Comparing eq.(3.5) with eq.(3.8), we can now back out the value of `Initial_addr_reg` in the binary:

$$\texttt{Initial\_addr\_reg} = \texttt{Base}_{\texttt{outer}} + \texttt{num\_i} \times \texttt{size\_j} \tag{3.9}$$

We will now prove that `Initial_addr_reg` is an induction variable in the outer `i` loop. To do so, we prove that the value of `Initial_addr_reg` for `num_i` vs ($\texttt{num\_i} + 1$) is a constant value, which is the characteristic of an induction variable. Substituting ($\texttt{num\_i} + 1$) in eq.(3.9), we get:

$$\texttt{Initial\_addr\_reg}_{\texttt{num\_i}+1} = \texttt{Base}_{\texttt{outer}} + (\texttt{num\_i} + 1) \times \texttt{size\_j} \tag{3.10}$$

subtracting eq.(3.10) - eq.(3.9) yields:

$$\texttt{step}_{\texttt{i}} = \texttt{size\_j} \tag{3.11}$$

since by definition $\texttt{step}_{\texttt{i}} = \texttt{Initial\_addr\_reg}_{\texttt{num\_i}+1} - \texttt{Initial\_addr\_reg}_{\texttt{num\_i}}$. Substituting eq.(3.11) in eq.(3.8) yields:

$$\texttt{addr\_reg} = \texttt{Base}_{\texttt{outer}} + \texttt{num\_i} \times \texttt{step}_{\texttt{i}} + \texttt{num\_j} \times \texttt{step}_{\texttt{j}} \tag{3.12}$$

Generalizing this to an n-dimensional loop yields:

$$\texttt{addr\_reg} = \texttt{Base}_{\texttt{outer}} + \sum_{\texttt{k=1}}^{\texttt{n}} \texttt{num\_k} \times \mathrm{step}_{\mathrm{k}} \tag{3.13}$$

This is an important equation since it will help us derive dependence vectors next. An equation of the form eq.(3.13) is constructed by our binary rewriter for every memory reference for which special statements (such as (A) to (F) in the two-dimensional loop case) can be identified. In particular, the $\texttt{Base}_{\texttt{outer}}$ value is constructed from statement (E), whereas the step values of the inner and outer loop are found from statements (A) and (D), respectively. Memory references for which such special statements cannot be found are assumed to be non-affine, and are not analyzed further.

Deriving dependence vectors Next we aim to define the dependence vector between pairs of array accesses in the loop. To do so, we consider any two derived induction variable references in a loop (not necessarily the two in the code example above) with addresses `addr_reg1` and `addr_reg2`. Their expressions would be the following:

$$\texttt{addr\_reg}_1 = \texttt{Base}_{\texttt{outer1}} + \sum_{\texttt{k=1}}^{\texttt{n}} \texttt{num\_k} \times \mathrm{step}_{\mathrm{1k}} \tag{3.14}$$

$$\texttt{addr\_reg}_2 = \texttt{Base}_{\texttt{outer2}} + \sum_{\texttt{k=1}}^{\texttt{n}} \texttt{num\_k}' \times \mathrm{step}_{\mathrm{2k}} \tag{3.15}$$

After deriving these equations, the next step is to calculate the distance vector $(\mathtt{d1}, \mathtt{d2}, ..., \mathtt{dn})$ associated with these accesses. Say that $(\mathtt{num\_1}, \mathtt{num\_2}, ..., \mathtt{num\_n})$ and $(\mathtt{num\_1'}, \mathtt{num\_2'}, ..., \mathtt{num\_n'})$ are the iterations where $\mathtt{add\_reg_1}$ and $\mathtt{addr\_reg_2}$ alias to the same memory location, then by definition $(\mathtt{num\_1} - \mathtt{num\_1'}, \cdots \mathtt{num\_n} - \mathtt{num\_n'})$ is the distance vector associated with these accesses [2]. Hence, to calculate this distance vector we need to equate the R.H.S of eq.(3.14) and eq.(3.15). The unknowns in the equation are $\mathtt{num\_1}, \mathtt{num\_2}, ..., \mathtt{num\_n}, \mathtt{num\_1'}, \mathtt{num\_2'}, ...$ and $\mathtt{num\_n'}$. We now have 2n unknowns and one equation. But we also have the following bounds for these unknowns, as they are the number of iterations of loop dimensions.

$$0 \leq \mathtt{num\_k}, \mathtt{num\_k'} \leq \lfloor \frac{\mathtt{ub_k} - \mathtt{lb_k}}{\mathtt{step_{1k}}} \rfloor \quad \forall \mathtt{k} \in [\mathtt{1} : \mathtt{n}] \tag{3.16}$$

We derive these bounds from the special induction variable statements in the binary (statements (B), (C), (E), (F) in the two-dimensional loop case.). For loops nested with higher depths there will be statements (H), (I), . . . to determine the bounds. We now solve for the distance vectors using the equation and bound conditions. One of the following four conditions may happen:

- There is no solution to this equation in the given space. This means that the two addresses do not alias with one another. We add a distance vector of (0, 0, . . .) to this loop.

[2]Distance vectors need to be lexicographically positive, hence if $(\mathtt{num\_1} - \mathtt{num\_1'})$ is negative then the distance vector is $(\mathtt{num\_1'} - \mathtt{num\_1}, \mathtt{num\_2'} - \mathtt{num\_2}, ..., \mathtt{num\_n'} - \mathtt{num\_n})$

- There is a deterministic solution (d1, d2, . . ., dn). Then we add this to the distance vectors of the loop.

- There are multiple deterministic solutions to this equation. Then we add all the deterministic solutions to the distance vectors of this loop.

- In all other cases, when there are uncountable many solutions or when we are unable to determine the solution, the direction vector added has elements per loop dimension that are a combination of $<$, $>$, $=$ and * [7]depending on the dimension that is uncountable and the direction in which it is uncountable.

Traditional affine theory defines the (Greatest Common Divisor) GCD test [14, 15], Banerjee test [13], Delta Array tests [16] and the Single Index Variable (SIV) and Multiple Index Variable (MIV) tests [13] [6] to solve the linear equations that we derive from source. We use the same techniques to solve the equations from low-level code. Multiple tests have been defined as the techniques evolved to more precise solutions in increasing order of complexity.

If the bounds of the loop are unknown we can still say something about these equations in the infinite space. But of course this is not always true. Hence, we have developed techniques specific to a binary in case of unknown bounds and these are presented in chapter 4.

The techniques presented in this section are different from source in the following way:

- The equations to be solved are directly derived from the binary as against the symbolic array index expressions readily available from source but absent in binaries.

- From source we derive the distance vectors by solving each dimension separately, where as from binary we derive the equations equivalent to linearizing the array. We do this since there is no symbolic information in the binary to determine array bounds and dimensions. We have found that these techniques are nearly as powerful as source techniques on the Polybench benchmark suite. We have been able to discover the same dependence vectors from source as well as the corresponding binaries. In extremely rare cases, the dependence vectors from binaries are less precise than from source, but still conservative and correct. This impact is measured in chapter 7.

3.2 Examples

In this section we briefly apply the theory described in section 3.1 to four code examples to show how their loops will be parallelized from a binary without any symbolic information.

```
int A[20,50]
int B[20,50]
for i = 0 → 19 step 1
 for j = 0 → 49 step 1
  B[i,j] = A[i,j] + 10
```

(a) Example 1

```
int A[20,50]
for i = 0 → 19 step 1
 for j = 0 → 47 step 1
  A[i,j] = A[i,j+2] + 10
```

(b) Example 2

Example 1: The memory address expressions that we recover from the binary above will be of the form $\texttt{Base}_\texttt{A} + \texttt{200i} + \texttt{4j}$ and $\texttt{Base}_\texttt{B} + \texttt{200i} + \texttt{4j}$ (Assuming that the size of an integer is 4.). $\texttt{Base}_\texttt{A}$ and $\texttt{Base}_\texttt{B}$ will at least differ by 4000, since the size of each array is 4000 bytes. Without loss of generality lets assume we recover the following equations from the binary $\texttt{100} + \texttt{200i} + \texttt{4j}$ and $\texttt{4100} + \texttt{200i} + \texttt{4j}$.

When the code above is compiled to a stripped binary, symbolic information is lost. Hence we no know longer the location or dimension sizes (20, 50) of array `A`. However, we can also infer from binary code that the bounds of loop `i` is [0,19] and the bounds of loop `j` is [0,49].

Next, we apply the standard dependence tests to the set of equations we recovered $\texttt{100} + \texttt{200i} + \texttt{4j}$ and $\texttt{4100} + \texttt{200i} + \texttt{4j}$ within the ranges $0 \leq \text{i} \leq 19$ and $0 \leq \text{j} \leq 49$. The dependence tests reveal that both these memory expressions will accesses different memory locations in the iteration space, *i.e.*in other words no two iterations ever access the same location. Hence, the distance vector for this loop is (0,0) directly from binary code. We can now parallelize any dimension of the loop nest. We decide to parallelize the outer most loop dimension *i.e.*loop `i`.

Example 2: The memory address expression that we recover from the binary will be of the form $\texttt{Base}_\texttt{A} + \texttt{400i} + \texttt{4j}$ and $\texttt{Base}_\texttt{A} + \texttt{8} + \texttt{400i} + \texttt{4j}$. Without loss of generality, lets assume that $\texttt{Base}_\texttt{A}$ is 200, then the expressions that we recover from the binary will be $\texttt{200} + \texttt{400i} + \texttt{4j}$ and $\texttt{208} + \texttt{400i} + \texttt{4j}$ (assuming that the element size is 4). The ranges we discover for `i` is [0,19] and `j` is [0,47].

Next, we apply the dependence tests to this set of equations within the ranges discovered from the binary. Note that these were discovered directly from the binary without using any symbolic information that a source compiler would use. The dependence tests reveal that every two iteration of `j` accesses the same memory location as two iterations before this one *i.e.*the distance vector for this loop is (0,2). The distance vector reveals that the outer loop is parallel, hence we parallelize the outer loop `i` of the loop nest.

```
int A[100]
for i = 0 → 49 step 1
 A[i] = i;
 A[i+50] = i + 50;
```

(a) Example 3

```
int A[100]
for i = 0 → 98 step 1
 A[2i] = 10*i;
 A[2i+1] = 10*(i+50);
```

(b) Example 4

Example 3: The equations we will recover from the binary will be $\texttt{Base}_{\texttt{A}} + \texttt{4i}$ and $\texttt{Base}_{\texttt{A}} + \texttt{200} + \texttt{4i}$. Without loss of generality lets assume that $\texttt{Base}_{\texttt{A}}$ is 1000, then the expressions for memory that we recover directly form the binary will be $\texttt{1000} + \texttt{4i}$ and $\texttt{1200} + \texttt{4i}$. The range for `i` that is discovered from the binary is [0,49]. Using the dependence tests we discover that both these address expressions do not accesses the same memory location in any two iterations in the given iteration space. Hence, the distance vector associated with this loop is (0). After we have calculated the distance vector we parallelize the `i` loop of the loop nest.

Example 4: The equations we will recover from the binary will be $\texttt{Base}_{\texttt{A}} + \texttt{8i}$ and $\texttt{Base}_{\texttt{A}} + \texttt{4} + \texttt{8i}$. Without loss of generality lets assume that $\texttt{Base}_{\texttt{A}}$ is 100, then the expressions that are recovered from the binary are $\texttt{100} + \texttt{8i}$ and $\texttt{104} + \texttt{8i}$ and the range for `i` is [0,49]. Using the dependence tests we discover that no two iterations every access the same memory location again. Hence, the distance vector associated with this loop is (0). After we have calculated the distance vector we infer that loop `i` does not carry a dependence and hence can be parallelized.

Chapter 4

Run time dependent loop bounds

In this chapter we first describe the best-known methods for obtaining distance vectors from source code for affine loops with run-time determined loop bounds. We then present the limitations of the binary method presented in chapter 3 for the same and then describe our algorithm.

```
int A[20,50]
for i = 0 → ub_i step 1
  for j = 0 → ub_j step 1
    A[i,j] = A[i,j] + 10
```

The code shows a normalized loop, *i.e.*a loop with a lower bound of zero and a step of one. Loops can be normalized using existing methods such as the normalization pass in LLVM.

Figure 4.1: Code Example to motivate the algorithm to guess loop bounds for loops with run-time dependent loop bounds

Distance vectors from source code for the loop in figure 4.1 are calculated as follows. From source, existing methods make the assumption that row and column accesses are within the bounds of the array's dimensions. They solve for two iterations that refer to the same memory location within bounds for an infinite range of iteration values. If no solution exists, like in this example, we can conclude that no two iterations ever access the same memory. This implies that iterations of the `j` loop can execute in parallel (*i.e.*, the component of the distance vector for this loop is zero.) Such an analysis individually proves that the loop `i` is parallel.

To obtain distance vectors from binary for this code we cannot use the above source method since it relies on known affine expressions for array indexes in terms of induction variables, which are not apparent from the binary. Instead we start with the method for binaries in chapter 3. We showed that we can recover linearized expressions for memory accesses from a binary, and solving these multidimensional expressions gives us dependence vectors. In the presence of loop bounds the solutions from binaries are very powerful, and can handle most linear algebraic kernels as presented in chapter 3. However, when loop bounds are run-time dependent, we need to solve these multi-dimensional expressions in the infinite space (since we need to assume that the loop bounds can take any value at run-time). This greatly reduces the precision of the analysis.

For *e.g.*, let us apply the binary method from chapter 3 to the loop in figure 4.1. From the binary for the code above we will recover a memory expression of the form $\mathtt{Base_A} + \mathtt{200i} + \mathtt{4j}$ which corresponds to the $\mathtt{A}[\mathtt{i},\mathtt{j}]$ access (assuming the element size is 4). The "200" in $200i$ is because the size of a row is 50 elements, each of 4 bytes. We need to reason about this access in the infinite space for `i` and `j` since the loop bounds are unknown. In the infinite space, iterations (2,0), (1, 50) and (0, 100) refer to the same memory location. All of these iterations except (2,0) are not possible because the legal range of `j` is [0,49] and if it is greater than 49 the code accesses columns out of bounds, and thus wrapping into rows. Source code methods assume that such iterations are not possible; hence proving the loop is parallel. However, unlike source methods, the binary method in chapter 3 cannot make any assumptions about iterations remaining within array bounds, since differentiated array bounds are not known from binary code. As a result, without loop bounds, the binary method from chapter 3 fails to prove the loop is parallel because of the false loop-carried dependences that appear.

In this chapter we present a method to statically guess the most likely upper bounds of loops with unknown loop bounds. We then check the loop bounds at run-time, and run the parallel version of the loop if the loop bounds were indeed with in the ranges that we guessed. For *e.g.*for the loop shown in figure 4.1, using the theory presented in chapter 3 we discover the memory expression for the A[i][j] access to be $\texttt{Base}_\texttt{A} + \texttt{200i} + \texttt{4j}$. We then look at the coefficients multiplying the induction variables in this memory expression that we recover from the binary and guess the likely limit of the induction variable with the smallest coefficient (*i.e.*`j` in this example, since its coefficient `4` is smaller than the coefficient of `i` `200`) as the immediately higher coefficient divided by the coefficient of this induction variable; *i.e.*in this example we guess the limit on `j` as (Coefficient of `i`/Coefficient of `j`) ($i.e. \frac{200}{4} = 50$). By guessing that `j` is less than `50` no two iterations will access the same memory because now `j` has been prevented to run into `i` and we can parallelize the loop. At run-time we check if `j` indeed does not exceed `50` and this run-time check will always succeed and we will always execute the parallel version of the loop.

In the following section 4.1 we present more examples and briefly describe how our algorithm would guess loop bounds for them followed by the detailed algorithm in section 4.2. Within these bounds for the induction variables the loop is parallel in most cases. We insert run-time checks to check that these guesses were indeed correct before executing the parallel version of the loop; else, we execute the serial version of the loop. The run-time check is necessary for binary code since our guesses may be incorrect in some cases. We show one such example in section 4.1. However, this does not hurt us since that loop was actually serial in the original source code and the loop bounds we guess represent the parallel region.

4.1 Examples

In this section we first briefly describe the steps of the algorithm described in section 4.2 and then apply it to four code examples to show how their loops can be parallelized from a binary even though the loop bounds are run-time dependent.

First, we state the algorithm that we use to guess the upper loop bounds for a loop directly from a binary. Only the steps are outlined here, details in section 4.2.

Step 1: Divide memory accesses (both reads and writes) in a loop into *Dependence Groups (DGs).* Intuitively, a DG is a subset of memory addresses in the loop that are sufficiently close to one another.

Step 2: Arrange all DGs in ascending order of their base addresses, from $\mathtt{DG}_1$ to $\mathtt{DG}_\mathtt{T}$.

Step 3: For all the DGs that have writes in them make best guesses for the possible range for induction variables. These guesses are called intra-group constraints, since they are obtained by working on one DG at a time.

Step 4: Initiate a worklist by all DGs with constraints remaining after step 3.

Step 5: Work on each $\mathtt{DG_i}$ in the worklist and solve for the values of induction variables such that the accesses in $\mathtt{DG_i}$ do not overlap with those in $\mathtt{DG_{(i+1)}}$. This generates further guesses on the induction variables. Merge these new constraints with existing constraints for the same induction variable by choosing the minimum. These guesses are called inter-group constraints because they are obtained by constraining $\mathtt{DG_i}$ to not overlap $\mathtt{DG_{(i+1)}}$.

```
int A[20,50]
int B[20,50]
for i = 0 → ubi step 1
 for j = 0 → ubj step 1
  B[i,j] = A[i,j] + 10
```

(a) Example 1

```
int A[20,50]
for i = 0 → ubi step 1
 for j = 0 → ubj step 1
  A[2i,j] = 10*i+j
```

(b) Example 2

Figure 4.2: Example 1 & 2 to illustrate the algorithm to guess loop bounds

Example 1: The memory address expressions that we recover from the binary above will be of the form $\mathtt{Base_A} + \mathtt{200i} + \mathtt{4j}$ and $\mathtt{Base_B} + \mathtt{200i} + \mathtt{4j}$ (Assuming that the size of an integer is 4.). $\mathtt{Base_A}$ and $\mathtt{Base_B}$ will at least differ by 4000, since the size of each array is 4000 bytes. Without loss of generality lets assume we recover the following equations from the binary $\mathtt{100} + \mathtt{200i} + \mathtt{4j}$ and $\mathtt{4100} + \mathtt{200i} + \mathtt{4j}$.

When the code above is compiled to a stripped binary, symbolic information is lost. Hence we no know longer the location or dimension sizes (20, 50) of array A. Hence we can no longer infer (as we implicitly do from source) that $\mathtt{ub_i}$ ¡ 20 and $\mathtt{ub_j}$ ¡ 50. Instead maximum values of these bounds of the loops must be inferred.

We now show briefly how our algorithm is applied to these accesses to guess the bounds on `i` and `j`. In Step 1, we check to see if the accesses belong to different DGs. The heuristic we use is that the difference of the bases is greater than a factor (5 for our experiments) of the highest coefficient; *i.e.*$\texttt{Base}_\texttt{B} - \texttt{Base}_\texttt{A} > \texttt{5} \times \texttt{200}$ *i.e.*$(\texttt{4100} - \texttt{100}) > \texttt{5} \times \texttt{200}$. As this is true both the accesses will belong to different DGs. In Step 2, we arrange the DGs in ascending order of their bases. $\texttt{100} + \texttt{200i} + \texttt{4j}$ belongs to $\texttt{DG}_\texttt{1}$ because its base is lower than the second access which belongs to $\texttt{DG}_\texttt{2}$. In Step 3, we solve for intra-group constraints in $\texttt{DG}_\texttt{2}$ since it contains a write. We guess the bound on `j` by dividing the co-efficient multiplying `i` (the just higher co-efficient in the linearized equation) by the co-efficient of `j` *i.e.*($\frac{\texttt{200}}{\texttt{4}}$ = 50). Hence, we guess that `j` must belong to [0, 49]. In step 4, we create a worklist with all DGs that have constraints remaining. In this example both the DGs have constraints remaining on `i`; hence both of them will belong to the worklist. In step 5, we guess the bound on `i` by solving that $\texttt{DG}_\texttt{1}$ *i.e.*$\texttt{100} + \texttt{200i} + \texttt{4j}$ does not overlap with $\texttt{DG}_\texttt{2}$ *i.e.*$\texttt{4100} + \texttt{200i} + \texttt{4j}$ given the highest possible value for `j` is 49; *i.e.*$\texttt{100} + \texttt{200i} + \texttt{4} * \texttt{49} < \texttt{4100}$. Hence, `i` must be less than 19.02 or in the range [0, 19]. Since $\texttt{DG}_\texttt{2}$ is the highest DG we do not solve for it overlapping with any other DG.

After we have applied our algorithm to this loop, our guess for `i` is [0,19] and `j` is [0,49]. We now solve for dependencies within this range for the loop and discover that the loop can be parallelized. We also add lightweight run-time checks before the parallel version of the loop (which will always succeed for this loop).

Example 2: The memory address expression that we recover from the binary will be of the form $\texttt{Base}_\texttt{A} + \texttt{400i} + \texttt{4j}$. Since there is only one access, step 1 and 2 will result in placing it in $\texttt{DG}_\texttt{1}$. In step 3, we guess that the bound of `j` is ($\frac{400}{4} = 100$) or the range of j is guessed to be [0, 99]. There would be no step 4 and 5 for this loop since there is only one DG.

Next we calculate dependencies assuming the range of `j` is [0,99] and `i` can take any value and discover that the loop can be parallelized. In reality however the range of `j` will not exceed [0, 49]. But our larger discovered bounds work well since even if they did exceed 49 and be below 99 this loop can still be parallelized. i.e if the programmer decided to access two rows using a column increment (which most programmers do not do) it is still a parallel loop. From the binary this means that we see a $\texttt{A}[\texttt{20}, \texttt{50}]$ array as a $\texttt{A}[\texttt{10}, \texttt{100}]$ array. However, this is fine as we reason about the dependencies in the correct way and parallelize the loop only when our run-time checks succeed.

```
int A[100]
for i = 0 → ub_i step 1
  A[i] = i;
  A[i+50] = i + 50;
```

(a) Example 3

```
int A[100]
for i = 0 → ub_i step 1
  A[2i] = 10*i;
  A[i+50] = 10*(i+50);
```

(b) Example 4

Figure 4.3: Example 3 & 4 to illustrate the algorithm to guess loop bounds

Example 3: The equations we will recover from the binary will be $\texttt{Base}_\texttt{A} + \texttt{4i}$ and $\texttt{Base}_\texttt{A} + \texttt{200} + \texttt{4i}$. After step 1, we will place them in different DGs since the difference between the bases (200) is greater than 5 times the highest co-efficient 4. After arranging the DGs in ascending order in step 2, $\texttt{Base}_\texttt{A} + \texttt{4i}$ will belong to $\texttt{DG}_1$ and $\texttt{Base}_\texttt{A} + \texttt{200} + \texttt{4i}$ will belong to $\texttt{DG}_2$. No intra-group guesses are calculated in step 3 since the recovered equations are single dimensional. After step 4, the worklist is populated with both the DGs since both contain `i` for which there is no guess as yet. In step 5, we solve for inter-group guesses such that $\texttt{DG}_1$ does not overlap with $\texttt{DG}_2$, *i.e.*$\texttt{4i} < \texttt{200}$ or $\texttt{i} < \texttt{50}$. Hence, the range we guess for `i` is [0,49] which is also the actual limit on `i` from source. The run-time check will always succeed in binary code and we will execute the parallel version of this loop. This is correct because, regardless of the value of $\texttt{ub}_\texttt{i}$, the two array references access non-intersecting portions of the array. Our method correctly treats these non-intersecting portions as different DGs.

Example 4: The equations we will recover from the binary will be $\texttt{Base}_\texttt{A} + \texttt{8i}$ and $\texttt{Base}_\texttt{A} + \texttt{200} + \texttt{4i}$. After step 1, we will place them in different DGs since the difference between the bases (200) is greater than 5 times the highest coefficient 8. After arranging the DGs in ascending order in step 2, $\texttt{Base}_\texttt{A} + \texttt{8i}$ will belong to $\texttt{DG}_1$ and $\texttt{Base}_\texttt{A} + \texttt{200} + \texttt{4i}$ will belong to $\texttt{DG}_2$. No intra-group guesses are calculated in step 3 since the recovered equations are single dimensional. After step 4, the worklist will contain both the DGs since both contain i for which there is no guess as yet. In step 5, we solve for inter-group guesses such that $\texttt{DG}_1$ does not overlap with $\texttt{DG}_2$, *i.e.*$\texttt{8i} < \texttt{200}$ or $\texttt{i} < \texttt{25}$. The range we guess for `i` is [0,24]. However, the actual range of `i` is [0,49], so our run-time check will fail and we will execute the serial version of the loop. This is fine since the actual loop is serial (it cannot be parallelized).

4.2 Algorithm to guess Loop Bounds

In this section we describe in detail the algorithm briefly presented in section 4.1. First we describe which loops from binary code we work on and then in subsequent subsections we describe the steps of the algorithm in detail.

First, we would like to present to you the kind of loops on which our algorithm is applied on and the kind of loops on which our algorithm is effective. We apply our method to every loop that has only affine accesses in them *i.e.*accesses of the form $\mathtt{A[i+3][5j]}, \mathtt{A[i+j][k+i]}, \mathtt{A[j][j]}$ etc are all processed by our method. Our method is able to effectively parallelize loop nests with array accesses of the form $\mathtt{A[i][2j]}, \mathtt{A[3j][i+100]}$, and $\mathtt{A[j][i]}$ *i.e.*normalized accesses with induction variables in any order; however affine accesses having multiple induction variables in a single array index expression (such as $\mathtt{A[i+j]}$) or having repeated induction variables (such as $\mathtt{A[j][j]}$) are not currently effectively parallelized by our method.

Since it is impossible to tell from binary code the form of the accesses in the source it was compiled from our method is applied to all loops that contain only affine accesses. However, the guesses may not be correct for the loops containing multiple induction variables in a single array expression or repeated induction variables. Hence, the run-time checks might fail for these loops and the serial version of the code may be executed. However, these kinds of accesses are rare in real code and hence our method is nearly as powerful from binary as from source.

Every affine memory address that we recover from the binary is a linearized multidimensional equation of the form 3.13 as shown in chapter 3:

$$\texttt{MemAddr}(\texttt{Base}, \texttt{d}) = \texttt{Base} + \sum_{\texttt{j}=1}^{\texttt{n}} \texttt{d}_{\texttt{j}} \times \texttt{i}_{\texttt{j}} \tag{4.1}$$

(where `Base` and d's are constants or loop invariant quantities, i's are induction variables, and $\texttt{d}_1 >= \texttt{d}_2 >= \; >= \texttt{d}_\texttt{n}$). We arrange the memory expression with d's in this order since in the algorithm we use the immediately higher coefficient while guessing the value of a particular induction variable, *i.e.*we use $\texttt{d}_{(\texttt{m}+1)}$ when guessing the values of induction variable $\texttt{i}_\texttt{m}$. We will refer to memory addresses from binary using $\texttt{MemAddr}(\texttt{Base}, \texttt{d})$ throughout this chapter. Different memory addresses from binary will have different `Base` and `d`s. Since we work on loops with only affine accesses in them, if we discover that a loop contains an access that is not affine *i.e.*we cannot discover an linearized expression for it then we do not work on that loop.

In the following subsections we first describe our algorithm and then present an intuition for it. No proof can be presented that the guessed loop bounds are always correct, since they are not always correct. However our overall method is always correct, since we include a run-time check for the bound which executes fallback serial code when the actual bound found at run-time is outside the guessed range of bounds. In the common case, the check succeeds and parallel code is executed.

4.2.1 Step 1: Divide the accesses into DGs

A DG is a subset of memory references in the loop that are sufficiently close to one another and these set of references most likely do not overlap with other

DGs. Intuitively, while dividing memory references into DGs we try to guess all the references which access the same array, or a region of an array not overlapping with other regions. This is not immediately apparent since binaries lack symbolic information containing the locations and sizes of arrays.

We create DGs using the following method. We look at the address of each memory reference and place it in an already present DG if it is sufficiently close to the addresses already in that DG; else we create a new DG with this memory address. We define that two accesses are sufficiently close to one another if the difference between the bases is within a factor of the highest coefficient in the memory expression. The formal algorithm is presented in algorithm 1.

We now describe some of the terms used in the algorithm. `DGlist` is a list of DGs. It is initialized to NULL and then populated as we consider every memory access in the loop. $\mathtt{d}_1$ is the highest coefficient in the memory expression; hence, if the difference between the base and any of the bases already in a DG is within a factor of it, we guess that it most likely belongs to the same memory array and place this reference in that DG. $\mathtt{CD}_{\mathtt{Thres}}$ is a number that guesses the maximum difference between references in the same DG. Currently we set $\mathtt{CD}_{\mathtt{Thres}}$ to 5. With $\mathtt{CD}_{\mathtt{Thres}} = \mathtt{5}$, two accesses to A[i] and A[i+4] will belong to the same DG, whereas two accesses to A[i] and A[i+10] will belong to different DGs. Having accesses to A[i] and A[i+e] where $e > 5$ in the same loop is relatively rare in affine codes; as most constants in affine codes are less than 5. If this rare case occurs we will treat A[i] and A[i+e] as accesses to two different arrays. Accesses to different arrays A and B will belong to different DGs unless the highest dimension of A has size less than 5 (which is rare) and B immediately follows A in the binary's data layout. If this rare case appears we will treat both A and B as the same array. In both the above cases, the run-time checks will fail and the serial version of the loop will be executed. Hence, the loop may run slower than from source, but correctness is always maintained.

4.2.2 Step 2: Arrange DGs in ascending order

In this step we reorganize the DGs in `DGlist` in ascending order of the bases present in them. After arranging them in ascending order the following will be true:

All bases in $\mathtt{DG_1}$ < All bases in $\mathtt{DG_2}$ $< \cdots <$ All bases in $\mathtt{DG_T}$ (This will be $<$ since if they are equal they would belong to the same DG). We call this ordering of DGs, the *FullList*.

4.2.3 Step 3: Induce intra-group dependencies

In this step we make our best guesses for all array bounds, and hence induction variables, except the array bound of the highest dimension in an array reference. We make the guesses based on the assumption that array references accesses arrays within the bounds of each dimension.

We apply step 3 to every DG that has a write in it. The reason we apply it to DGs with writes in them is that even if a read accesses across bounds it does not create dependencies that prevent parallelization and guessing bounds considering DGs with only reads is not necessary. For *e.g.*, if there is an affine loop that only reads from an array, there is no need to guess bounds for such a loop, as it is parallel

Algorithm 1 Step 1: Algorithm to divide accesses into DGs

```
  Input: $MemAddr(Base, d)$ for all accesses in loop
  Output: $DGlist$ has the accesses divided into DGs
Require: Initialize $DGlist$ to NULL
  for all $MemAddr(Base, d)$ in loop do
      Initialize $TmpDGlist$ to NULL
      for all $DG_i$ in $DGlist$ do
          if $|Base -$ Any base in $DG_i| < d_1 \times CD_{Thres}$ then
              Put $MemAddr(Base, d)$ in $DG_i$
              Put $DG_i$ in $TmpDGlist$
          end if
      end for
      if sizeof $TmpDGList > 1$ then
          Merge all the DGs in $TmpDGList$
      end if
      if sizeof $TmpDGList == 0$ then
          A new DG with $MemAddr$ in it is added to $DGlist$
      end if
  end for
```

in the infinite space as long as there is no scalar dependency in it.

Step 3 is divided into two sub steps 3.1 and 3.2. Step 3.1 is applied to every access in a DG and step 3.2 is applied to a pair of accesses in a DG. We first present the algorithms for both the sub steps before presenting intuitions for them.

Step 3.1: The formal algorithm for step 3.1 is presented in algorithm 2. We are working on loop nests with induction variables say $\mathtt{i_1}, \mathtt{i_2}, \cdots, \mathtt{i_n}$ and guesses for each $\mathtt{g_1}, \mathtt{g_2}, \cdots, \mathtt{g_n}$. First, we initialize the guesses for each of these induction variables to `TOP` representing infinity which is what we know about each of the induction variables before the start of this step. Then we look at every memory access which is of the form $\mathtt{MemAddr}(\mathtt{Base}, \mathtt{d})$ (from eq(4.1)) and make guesses for each induction variable as follows.

Algorithm 2 Step 3.1: Guesses for induction variables using one access

```
  Input: All DGs that have a write in them
  Output: Initial guesses for the induction variables
Require: Initialize each of $g_1, g_2, \cdots, g_n$ to $TOP$
  for all $DG_i$ in FullList that has a write in it do
    for all $MemAddr(Base, d)$ in $DG_i$ do
      for $k = 2 \rightarrow n$ do
        $g_{1k} = \lfloor \frac{d_{(k-1)}}{d_k} \rfloor$
        $g_k = min(g_k, g_{1k})$
      end for
    end for
  end for
```

$$\text{The guess on } \mathtt{i_k},\ \mathtt{g_{1k}} = \lfloor \tfrac{\mathtt{d_{(k-1)}}}{\mathtt{d_k}} \rfloor\ \forall \mathtt{k} \in [2, \mathtt{n}] \tag{4.2}$$

We then update the guess already in $\mathtt{g_k}$ for $\mathtt{i_k}$ using

$$\mathtt{g_k} = \mathtt{min}(\mathtt{g_k}, \mathtt{g_{1k}}) \tag{4.3}$$

Note: $\mathtt{min}(\mathtt{TOP}, \mathtt{g_{1k}}) = \mathtt{g_{1k}}$ since $\mathtt{TOP}$ represents infinity.

We apply this to every memory access in every DG that has a write and guess for every induction variable other than the highest dimension $\mathtt{i_1}$. Note that we cannot make a guess for $\mathtt{i_1}$ since there is no $\mathtt{d_0}$ in the equation. Hence, we do not have a guess for $\mathtt{i_1}$ in this step. The guess for $\mathtt{i_1}$ is made in step 5 and will be described later.

Step 3.2: After we have applied step 3.1 to all DGs that have a write in them, we work on the same DGs considering pairs of accesses in them and apply step 3.2 on them. This algorithm is presented in algorithm 3.

We now describe the algorithm briefly. We first initialize $\mathtt{x_1}, \mathtt{x_2}, \cdots, \mathtt{x_n}$ to zeroes. These represent the adjustment we need to make to each of the induction variable bound guesses at the end of this step. Then we consider pairs of accesses in this DG, if the bases are different then we store the absolute difference in $\mathtt{Base_{diff}}$. We then run a loop that checks to see which factor of this difference came from which co-efficient and keep track of that in $\mathtt{d_k}$s. Later these are subtracted from the

Algorithm 3 Step 3.2: Guesses for induction variables using pair of accesses

```
  Input: All $DG_i$s that have a write in them and $g_1, g_2, \cdots, g_n$ from step 3.1
  Output: Refined guesses for induction variables
Require: Initialize $x_1, x_2, \cdots, x_n$ to zeroes
  for $MemAddr_1(Base_1, d)$,$MemAddr_2(Base_2, e)$ in $DG_i$ do
      $Base_{diff} = |\ Base_1$-$Base_2\ |$
      for $k = 1 \rightarrow n$ do
          if $\frac{Base_{diff}}{gcd(d_k,e_k)} \geq 1$ then
              $x_{11} = \lfloor \frac{Base_{diff}}{gcd(d_k,e_k)} \rfloor$
              $x_k = max(x_k, x_{11})$
              $Base_{diff} = Base_{diff} - \lfloor \frac{Base_{diff}}{gcd(d_k,e_k)} \rfloor \times gcd(d_k, e_k)$
          end if
      end for
  end for
  for $k = 1 \rightarrow n$ do
      $g_k = g_k - x_k$
  end for
```

guesses for induction variables $\mathtt{g_k}$ from step 3.1.

It is important to make this adjustment to the guesses on loop bounds from step 3.1 since by doing so we are making sure that each of the accesses do not run into the higher dimension of the other. After this adjustment we will not have spurious dependencies from binary that prevent parallelization. We will present further intuition to this step below.

Intuition for Step 3.1: Let us assume that the binary code we are accessing came from source code where the loop nest had induction variables (say $\mathtt{i_1}, \mathtt{i_2}, \cdots, \mathtt{i_n}$) and an array accesses $\mathtt{A}[\mathtt{C_1} \times \mathtt{i_1} + \mathtt{B_1}][\mathtt{C_2} \times \mathtt{i_2} + \mathtt{B_2}] \cdots [\mathtt{C_n} \times \mathtt{i_n} + \mathtt{B_n}]$ in the loop and the size of array A is $[\mathtt{n1}][\mathtt{n2}] \cdots [\mathtt{nn}]$. Assume that none of the induction variables is repeated, however any ordering of the induction variables is allowed. This access when recovered from the binary will be of the form.

$$(\mathtt{Base_A} + \sum_{\mathtt{j=1}}^{\mathtt{n}} \mathtt{B_j} \times \prod_{\mathtt{m=j+1}}^{\mathtt{n}} \mathtt{n_m}) + \sum_{\mathtt{j=1}}^{\mathtt{n}} \mathtt{C_j} \times \prod_{\mathtt{m=j+1}}^{\mathtt{n}} \mathtt{n_m} \times \mathtt{i_j} \tag{4.4}$$

(This assumes an element size of 1; else each one of the terms will be multiplied by the element size.) The algorithm is correct even if the compiler uses the column-major layout; we assume the row-major layout only for presenting the intuition. Our results also include Fortran benchmarks for which the GCC compiler uses the column-major layout. $\mathtt{Base_A}$ and the terms containing $\mathtt{B}$'s (shown in parenthesis above) are rolled into the constant term when recovered from the binary. We know that the memory address that we recover from binary is of the form $\mathtt{MemAddr}(\mathtt{Base}, \mathtt{d})$

(from eq.(4.1)).

Equating (4.4) and (4.1) we get :

$$\mathtt{Base} = \mathtt{Base_A} + \sum_{\mathtt{j=1}}^{\mathtt{n}} \mathtt{B_j} \times \prod_{\mathtt{m=j+1}}^{\mathtt{n}} \mathtt{n_m} \tag{4.5}$$

$$\text{and, } \mathtt{d_j} = \mathtt{C_j} \times \prod_{\mathtt{m=j+1}}^{\mathtt{n}} \mathtt{n_m} \tag{4.6}$$

First, let us calculate the actual upper bounds of the induction variables from source. From source we know that the array indices do not access arrays out of their bounds. Hence, each dimension index must be less than the actual size of that dimension.

$$i.e., \mathtt{C_k} \times \mathtt{i_k} + \mathtt{B_k} < \mathtt{n_k} \tag{4.7}$$

$$\text{Rearranging the terms, } \mathtt{i_k} < \frac{(\mathtt{n_k} - \mathtt{B_k})}{\mathtt{C_k}} \tag{4.8}$$

Hence, the upper bound of $\mathtt{i_k}$ from source is $\frac{(\mathtt{n_k}-\mathtt{B_k})}{\mathtt{C_k}}$.

Second, let us see what our guess for induction variable $\mathtt{i_k}$ is by applying step 3.1 to this access. Our guess for induction variable $\mathtt{i_k}$ is obtained by substituting eq(4.6) in eq(4.2)

$$i.e., \ \mathtt{g_{1k}} = \lfloor \frac{\mathtt{C_{(k-1)}} \times \mathtt{n_k}}{\mathtt{C_k}} \rfloor \ \forall \mathtt{k} \in [2, \mathtt{n}] \tag{4.9}$$

Next taking the minimum of $\mathtt{g_{1k}}$ and TOP (the initialized value) we get,

$$\mathtt{g_k} = \mathtt{min}(\mathtt{TOP}, \mathtt{g_{1k}}) = \mathtt{g_{1k}} = \lfloor \frac{\mathtt{C_{(k-1)}} \times \mathtt{n_k}}{\mathtt{C_k}} \rfloor \ \forall \mathtt{k} \in [2, \mathtt{n}] \tag{4.10}$$

We now show that the guesses for induction bounds are greater than or equal to the actual loop bounds. This is important because if the guesses were lower than the actual bounds our run-time checks would fail and we would always run the serial version of the loop which would not serve the purpose of parallelization. We have already seen that the guess on the induction variable $\mathtt{i_k} = \frac{\mathtt{C_{(k-1)}} \times \mathtt{n_k}}{\mathtt{C_k}}$, this is greater than the actual limit of $\mathtt{i_k}$, which is $\frac{(\mathtt{n_k} - \mathtt{B_k})}{\mathtt{C_k}}$ from eq.(4.8). We observe that if $\mathtt{C_{(k-1)}}$ is 1 and $\mathtt{B_k}$ is 0, then the value we would have guessed is the same as the actual upper bound. Further, if $\mathtt{C_k}$ is 1 as well, the guess for $\mathtt{i_k}$ is $\mathtt{n_k}$, which is the size of that array dimension.

Every guess we make for the induction variables is actually higher than or equal to its actual bound as shown above. By taking the minimum at every step we have a guess that is at least its actual bound. However, we do acknowledge that if the accesses had multiple induction variables or repeated ones our guesses may be incorrect and hence we add run-time checks to make sure we run the serial version in such cases. Further, these kind of accesses are very rare in actual code and hence our method is as powerful as the source methods on real code.

Intuition for step 3.2: Let us assume that there is a second accesses to A, $\mathtt{A[C_1 \times i_1 + B_1 + E_1] \cdots [C_n \times i_n + B_n + E_n]}$ in this loop where $\mathtt{E}$s are small numbers < 5. The memory address for this access from binary will be of the form:

$$\mathtt{Base_A} + \sum_{\mathtt{j=1}}^{\mathtt{n}} (\mathtt{B_j} + \mathtt{E_j}) \times \prod_{\mathtt{x=j+1}}^{\mathtt{n}} \mathtt{n_x} + \sum_{\mathtt{j=1}}^{\mathtt{n}} \mathtt{C_j} \times \prod_{\mathtt{x=j+1}}^{\mathtt{n}} \mathtt{n_x} \times \mathtt{i_j} \tag{4.11}$$

Recollect that this access when recovered from the binary will be of the form $\mathtt{MemAddr(Base_2, e)}$, from equation (4.1):

$$\mathtt{MemAddr(Base_2, e)} = \mathtt{Base_2} + \sum_{\mathtt{j=1}}^{\mathtt{n}} \mathtt{e_j} \times \mathtt{i_j} \tag{4.12}$$

Equating eq.(4.11) and eq.(4.12), we get:

$$\mathtt{Base_2} = \mathtt{Base_A} + \sum_{\mathtt{j=1}}^{\mathtt{n}} (\mathtt{B_j} + \mathtt{E_j}) \times \prod_{\mathtt{x=j+1}}^{\mathtt{n}} \mathtt{n_x} \tag{4.13}$$

$$\text{and, } \mathtt{e_j} = \mathtt{C_j} \times \prod_{\mathtt{x=j+1}}^{\mathtt{n}} \mathtt{n_x} \tag{4.14}$$

First, let us prove that both the references belong to the same DG using step 1 since we would apply step 3.2 to them only if both of them belong to the same DG. From step 1 we know that if the difference between the bases is $< \mathtt{d_1} \times \mathtt{CD_{Thres}}$, then they will belong to the same DG.

$$i.e., \text{ if } |\mathtt{Base_2} - \mathtt{Base}| < \mathtt{d_1} \times \mathtt{CD_{Thres}} \tag{4.15}$$

then, both the accesses will belong to this DG.

The difference between the bases from eq.(4.13) and eq.(4.5) is

$$\mathtt{Base_2} - \mathtt{Base} = \sum_{j=1}^{n} \mathtt{E_j} \times \prod_{x=j+1}^{n} \mathtt{n_j} \tag{4.16}$$

We know from eq.(4.14) and eq.(4.6) that:

$$\mathtt{d_1} = \mathtt{e_1} = \mathtt{C_1} \times \prod_{m=2}^{n} \mathtt{n_m} \tag{4.17}$$

Now, substituting eq.(4.16) and eq.(4.17) in eq.(4.15) we get:

$$\text{If } \sum_{j=1}^{n} \mathtt{E_j} \times \prod_{x=j+1}^{n} \mathtt{n_j} < \mathtt{C_1} \times \prod_{m=2}^{n} \mathtt{n_m} \times \mathtt{CD_{Thres}} \tag{4.18}$$

, then both the accesses will belong to this DG

$$\Rightarrow \frac{\mathtt{E_1}}{\mathtt{C_1}} + \frac{\mathtt{E_2}}{\mathtt{C_1} \times \mathtt{n_2}} + \cdots + \frac{\mathtt{E_n}}{\mathtt{C_1} \times \prod_{m=2}^{n} \mathtt{n_m}} < \mathtt{CD_{Thres}} \tag{4.19}$$

(which will be true in most cases since $\mathtt{C_1}$ and $\mathtt{E}$s are small positive numbers < 5 and $\mathtt{n_2 \cdots n_n}$ are relatively large, and $\mathtt{CD_{Thres}}$ is 5 in our experiments.)

Hence, both these accesses $\mathtt{MemAddr(Base, d)}$ and $\mathtt{MemAddr(Base_2, e)}$ from the binary will belong to the same DG.

First, let us see what the bounds for the induction variable from source would be in the presence of the second access as well. We know that accesses from source do not access out of bounds in correct programs. We have seen that the bounds for each induction variable ($\mathtt{i_k}$) only considering the first access is $\frac{(\mathtt{n_k - B_k})}{\mathtt{C_k}}$ as shown in eq.(4.8). Now considering that the second access does not access out of bounds we get:

$$\mathtt{C_k \times i_k + B_k + E_k < n_k} \tag{4.20}$$

$$\text{Rearranging the terms, } \mathtt{i_k} < \frac{(\mathtt{n_k - (B_k + E_k)})}{\mathtt{C_k}} \tag{4.21}$$

The difference between the bounds calculated from eq.(4.8) and eq.(4.21) is $\frac{\mathtt{E_k}}{\mathtt{C_k}}$

Now let us apply algorithm 3 to both these accesses.

We know that $\mathtt{Base_{diff}} = \sum\limits_{\mathtt{j=1}}^{\mathtt{n}} \mathtt{E_j} \times \prod\limits_{\mathtt{x=j+1}}^{\mathtt{n}} \mathtt{n_j}$. By dividing it with $\mathtt{gcd(d_k, e_k) = d_k}$ (since $\mathtt{d_k = e_k}$ in our case) repeatedly in loop and keeping the remainder of it for the next iteration we recover $\mathtt{x_k}$s of the form $\lfloor \frac{\mathtt{E_k}}{\mathtt{C_k}} \rfloor$ as long as $\mathtt{C_k}$s are factors of $\mathtt{E_k}$s. By subtracting $\mathtt{x_k}$s from the already present guesses of the induction variables we get $\mathtt{g_k} = \frac{\mathtt{C_{(k-1)} \times n_k}}{\mathtt{C_k}} - \frac{\mathtt{E_k}}{\mathtt{C_k}}$. Many of the Es will be zeroes, hence we will not make adjustment to many bounds, however we will make adjustment to the bounds that have small constant Es in their terms. Further, it is good to note that the term we subtract using algorithm 3 is equivalent to the difference of the bounds as shown above.

It is important to note at this point that by subtracting from the already guessed bounds, we are making sure that the second access which accesses a few extra elements in some dimensions does not run into the higher dimension of the first access. This is very important because if we do not make this adjustment we will have extra dependencies from binary, which will prevent parallelization, and by subtracting the extra from bounds we will not see those spurious dependencies. Also it is important to note that the new guess we have for the bounds is also higher than or equal to the actual bounds of the loop.

4.2.4 Step 4: Create the worklist

In this step we create a worklist with DGs that have accesses with remaining constraints so that we can apply step 5 on them to guess the upper bounds for the remaining induction variables. After step 3 we have upper bound constraints for all the induction variables in the memory addresses other than the ones that correspond to the highest dimension in the write accesses. We need a method to guess the upper bound on these induction variables as well. This method is step 5. Hence, we now create a worklist with all DGs in which there is an induction variable for which we do not have an upper bound guess as yet. These would be the highest dimension induction variables since we do not have guesses for those after step 3. This worklist will enable us to work on only those DGs that have remaining constraints.

4.2.5 Step 5: Work on Inter-group constraints

In this step we look at all DGs in the worklist created in step 4 (recall that these DGs have induction variables for which we have no guesses as yet) and solve for this DG not overlapping with the immediately following DG in the FullList. While creating DGs we assumed that each DG corresponds to a non-overlapping array region. Hence, it is required that different DGs do not overlap with each other; else this would generate false dependencies from binaries. Solving this generates further guesses on the remaining induction variables. These guesses are called inter-group constraints.

The formal method for solving that $\mathtt{DG_i}$ from worklist does not overlap with $\mathtt{DG_{(i+1)}}$ (the immediately following DG in the FullList of DGs) is presented in algorithm 4; we describe it briefly here. For every $\mathtt{DG_i}$ that has constraints remaining we substitute the guesses for all induction variables other than the highest one in all its memory expressions and require that this be less than the lowest base in $\mathtt{DG_{(i+1)}}$. Solving the above constraint we can obtain a higher bound for the highest induction variable. We then choose the minimum of the present guess and the already present minimum guess for that induction variable. This way we ensure that all our guesses are respected.

Algorithm 4 Step 5: Algorithm for Inter-group constraints

```
**Input**: Worklist from step 4 and guesses $g_1, g_2, \cdots, g_n$ from step 3.2
**Output**: Final guesses for bounds $g_1, g_2, \cdots, g_n$
for all $DG_i$ in worklist after step 4 do
    for all $MemAddr(Base, d)$ in $DG_i$ do
        $Base_{low}$ = Lowest Base from $DG_{(i+1)}$ in FullList
        $g_{11} = \lfloor \frac{Base_{low} - Base - (\sum_{j=2}^{n} d_2 * g_2)}{d_1} \rfloor$
        $g_1 = min(g_1, g_{11})$
    end for
end for
```

Intuition for step 5: Now that we have presented an algorithm for calculating the bounds on the highest induction variable, let us apply this to an access from source code, to show that our method guesses the value for the highest induction variable that is $\geq$ to the actual bound on that induction variable.

In step 3 we assumed we were working with loop nests whose induction variables are $(\mathtt{say}, \mathtt{i_1}, \mathtt{i_2}, \cdots \mathtt{i_n})$ and array accesses $\mathtt{A}[\mathtt{C_1} \times \mathtt{i_1} + \mathtt{B_1}] \cdots [\mathtt{C_n} \times \mathtt{i_n} + \mathtt{B_n}]$ in the loop, and the size of array A is $[\mathtt{n_1}][\mathtt{n_2}] \cdots [\mathtt{n_n}]$. Let this access belong to $\mathtt{DG_i}$.

First, let us recollect the guesses for all induction variables except the highest induction variable from step 3. One of the guesses we would have made for induction variable $\mathtt{i_k}$ (where $\mathtt{k} \in$ [2,n]) is $\frac{\mathtt{C_{(k-1)}} \times \mathtt{n_k}}{\mathtt{C_k}}$ (eq.(4.9)). Hence, the final guess after step 4 will be equal to or lower than this guess.

Next, let us assume that there is an access to array B in the same loop belonging to $\mathtt{DG}_{(\mathtt{i+1})}$, *i.e.*the immediately following DG in the FullList. If this second array $\mathtt{B}$ is laid immediately after $\mathtt{A}$ in the binary, then $\mathtt{Base}_{\mathtt{B}}$ will be at least:

$$\mathtt{Base}_{\mathtt{B}} = \mathtt{Base}_{\mathtt{A}} + \prod_{\mathtt{j=1}}^{\mathtt{n}} \mathtt{n}_{\mathtt{j}} \text{ (this term is the size of A)} \tag{4.22}$$

Let us assume that all accesses corresponding to B belong to $\mathtt{DG}_{(\mathtt{i+1})}$. The lowest address of $\mathtt{DG}_{(\mathtt{i+1})}$ will be $\mathtt{Base}_{\mathtt{B}}$.

Next, we apply the method in algorithm 4 for solving $\mathtt{DG}_{\mathtt{i}}$ not overlapping with $\mathtt{DG}_{(\mathtt{i+1})}$ from source to derive the guess for $\mathtt{i}_1$ and then verify that this guess is correct. For doing so we must substitute our guesses for all the induction variables except the highest dimension induction variable in the expression of memory address A and this must be less than $\mathtt{Base}_{\mathtt{B}}$. The expression for memory address A obtained by substituting the intra-group guesses eq.(4.9) in eq.(4.4) is:

$$\mathtt{Base}_{\mathtt{A}} + \sum_{\mathtt{j=1}}^{\mathtt{n}} \mathtt{B}_{\mathtt{j}} \times \prod_{\mathtt{x=j+1}}^{\mathtt{n}} \mathtt{n}_{\mathtt{x}} + \mathtt{C}_1 \times \prod_{\mathtt{x=2}}^{\mathtt{n}} \mathtt{n}_{\mathtt{x}} \times \mathtt{i}_1 + \sum_{\mathtt{j=2}}^{\mathtt{n}} \mathtt{C}_{\mathtt{j}} \times \prod_{\mathtt{x=j+1}}^{\mathtt{n}} \mathtt{n}_{\mathtt{x}} \times (\frac{\mathtt{C}_{\mathtt{j-1}} \times \mathtt{n}_{\mathtt{j}}}{\mathtt{C}_{\mathtt{j}}} - 1) \tag{4.23}$$

The only unknown in eq.(4.23) is $\mathtt{i}_1$. This must be less than $\mathtt{Base}_{\mathtt{B}}$ (from eq.(4.22)).

Hence,

$$\mathtt{Base_A} + \sum_{j=1}^{n} \mathtt{B_j} \times \prod_{x=j+1}^{n} \mathtt{n_x} + \mathtt{C_1} \times \prod_{x=2}^{n} \mathtt{n_x} \times \mathtt{i_1} + \sum_{j=1}^{n} \mathtt{C_j} \times \prod_{x=j+1}^{n} \mathtt{n_x} - \sum_{j=2}^{n} \mathtt{C_j} \times \prod_{x=j+1}^{n} \mathtt{n_x} \\ \leq \mathtt{Base_A} + \prod_{j=1}^{n} \mathtt{n_j} \tag{4.24}$$

Rearranging the terms we get,

$$\mathtt{i_1} \leq \frac{\prod_{j=1}^{n} \mathtt{n_j} - (\mathtt{C_1} \times \prod_{x=2}^{n} \mathtt{n_x} + \sum_{j=1}^{n} \mathtt{B_j} \times \prod_{x=j+1}^{n} \mathtt{n_x})}{\mathtt{C_1} \times \prod_{x=2}^{n} \mathtt{n_x}} \tag{4.25}$$

Further,

$$\mathtt{i_1} \leq \frac{\mathtt{n_1}}{\mathtt{C_1}} - 1 - \frac{\mathtt{B_1}}{\mathtt{C_1}} - \frac{(\sum_{j=2}^{n} \mathtt{B_j} \times \prod_{x=j+1}^{n} \mathtt{n_x})}{\mathtt{C_1} \times \prod_{x=2}^{n} \mathtt{n_x}} \tag{4.26}$$

$$i.e. \mathtt{i_1} \leq \frac{\mathtt{n_1}}{\mathtt{C_1}} - 1 - \frac{\mathtt{B_1}}{\mathtt{C_1}} - (\Delta) \tag{4.27}$$

The remaining values are small since the constant C's and B's are small and the sizes of arrays in affine code are generally large.

Hence, the guess for $\mathtt{i_1}$ will be:

$$g_1 = \lfloor \frac{(n_1 - B_1)}{C_1} - 1 \rfloor \tag{4.28}$$

As seen before from source we require that the array expression must not exceed the size of the array dimension. Hence the highest dimension array expression ($\mathtt{C_1} \times \mathtt{i_1} + \mathtt{B_1}$) must not exceed the highest dimension ($\mathtt{n_1}$).

$$i.e. \mathtt{C_1} \times \mathtt{i_1} + \mathtt{B_1} < \mathtt{n_1} \tag{4.29}$$

Rearranging the terms $\mathtt{i_1} < \frac{(\mathtt{n_1}-\mathtt{B_1})}{\mathtt{C_1}}$

Hence, the maximum value $\mathtt{i_1}$ can take is $\frac{(\mathtt{n_1}-\mathtt{B_1})}{\mathtt{C_1}} - \mathtt{1}$ and this is what we get by solving the equations from binary.

We have now seen that the algorithm 4 to calculate the bounds on the highest dimension induction variable yields a limit on it that is the true limit on it even when the method is applied to source code.

At the end of step 5, we now have made best guesses for all induction variables in the loop that appear in a memory address. If there is an induction variable that does not appear in any memory access, then we just assume that it can take any value since we have no way of determining its bounds. This does not hurt our method and is reasonable since even from source if an induction variable does not appear in any of the memory addresses present in the loop it could take any value at run-time and this would be legal.

For array accesses that came from dynamically allocated memory we apply the algorithm described above. It is important to note that all `d`s in the `MemAddr`(`Base`, `d`) expression would be loop invariant symbols rather than constants.

In many cases the memory expression we recover from binary code for these accesses will be of the form

$$\mathtt{Base} + \mathtt{x_1} \times \mathtt{x_2} \cdots \mathtt{x_n} \times \mathtt{i_1} + \mathtt{x_2} \times \mathtt{x_3} \cdots \mathtt{x_n} \times \mathtt{i_2} + \cdots + \mathtt{x_n} \times \mathtt{i_n} \tag{4.30}$$

where all the $\mathtt{x}$s and $\mathtt{Base}$ are loop invariant quantities. By applying the algorithm to such an access we guess that the bound on $\mathtt{i_k}$ is $\mathtt{x_{(n-1)}}$. We then check that the actual bounds are less than this loop invariant quantity (this check would succeed) before executing the parallel version of the loop.

Now that we have constraints on all the induction variables, we calculate the distance vectors and take parallelizing decisions for this loop assuming these as loop bounds. We then clone this loop and run the parallel version of the loop when the run-time checks for all induction variables succeed; else we run the serial version of the loop. Since we check at run-time that the loop bounds that we have guessed are actually correct we will always be conservatively correct. Please note that using the distance vector method to parallelize is our implementation method, one may use any parallelizing decision algorithm including polyhedral methods.

Chapter 5

Cache Analysis and Loop Transformation

In this chapter we present techniques to calculate cache reuse metric directly from binary code and then use it to take decisions about transforming the loop using an algorithm adopted from McKinley's [12] algorithm.

5.1 Characterizing cache reuse

To increase the performance of affine loops it is extremely important to study the cache access patterns in the loop and optimize so that the cache is accessed efficiently. Interchanging loop dimensions in a loop nest changes the access patterns of the arrays. Different interchanged orders of loop dimensions generally have very different run-times and hence it is important to choose the correct ordering of loops. Further, fusing different loop nests may increase the reuse across memory accesses and decrease the run-time of the loop. Hence, it is important to study the cache behavior in affine loops and take decisions about the loop transformations that need to be applied to a particular loop nest.

Existing source-level cache reuse estimation techniques do not work on binaries, and in this chapter we derive binary-level techniques for the same. In any search strategy for deciding loop transformations to optimize for cache, it is very useful (and often necessary) to have a *cache reuse estimation module* that can evaluate the cache access cost of any proposed transformation order. We will also evaluate the solution in a concrete implementation of cache optimization based on McKinley's algorithm presented in [12], which is meant for parallelizing compilers enhanced with additional cache optimization search methods that will be described in section 5.2. *However our cache reuse model is not restricted to [12] and can be applied to any cache optimization strategy, both in parallelizing and non-parallelizing compilers.*

For every loop dimension containing affine accesses, we define `LoopCost(l)` as the total number of cache lines that will be accessed by the affine accesses in the loop nest when the dimension `l` is interchanged to the innermost position. It is worthy noting that `LoopCost(l)` does not measure the reuse directly. Instead `reuse` = `Cache access cost assuming all misses - LoopCost(l)`. The goal is that the final loop ordering should maximize reuse, which is the same as to minimize `LoopCost(l)`. In this chapter we first describe the existing method to calculate the `LoopCost(l)` from source in affine literature, and then show our new method to adapt the source method to a binary.

In section 5.2 we describe how the `LoopCost(l)` values for each dimension `l` can be used to decide an ideal ordering of loops in a loop nest. Further, this *ideal order* may not always be achievable given dependencies that prevent interchange; hence, we use an algorithm to achieve the best possible order, which we call *optimal order*. This algorithm has been described in [12] and will be reproduced in section 5.2.

5.1.1 From source

This section overviews the strategy used to calculate `LoopCost(l)` in terms of the cache lines accessed from source. This formulation has been described by McKinley in [12], but is widely used in all types of affine analysis such as [33] in traditional affine literature and [34] in the polyhedral framework.

First, let us describe the two kinds of data reuse that are present in affine loops.

- **Temporal reuse:** is present when multiple accesses are made to the same memory location. When a single array reference in different iterations access the same location it is called *self-temporal-reuse*. When multiple array references (in same or different iterations) access the same location it is called *group-temporal-reuse*.

- **Spatial reuse:** is present when accesses to nearby memory locations that share a cache line at some level of the memory hierarchy. Spatial reuse may result from *self-spatial-reuse* – consecutive accesses by the same array reference to the same cache line; or from *group-spatial-reuse* – multiple array references accessing the same cache line.

Without loss of generality, we assume C has row-major storage when calculating the `LoopCost(l)` from source. We then show that from a binary we do not need this assumption.

The example in figure 5.1 presents a loop code from gemver.c (a part of the *polybench* benchmark suite). We will use this as a running example in this section

```
for i from $lb_i$ to $ub_i$
  for j from $lb_j$ to $ub_j$
    X[i] = X[i] + beta * A[j,i] * Y[j]
  end for
end for
```

(a) Original - Loop in gemver

```
for j from $lb_j$ to $ub_j$
  for i from $lb_i$ to $ub_i$
    X[i] = X[i] + beta * A[j,i] * Y[j]
  end for
end for
```

(b) Interchanged Loop to maximize reuse

Figure 5.1: Example Loop to illustrate Cache Reuse Algorithm

and the next to calculate `LoopCost(l)` both from source and binary.

Figure 5.1 shows how loop interchange can help cache performance. We will show that figure 5.1(a), which shows the original code, has higher `LoopCost(l)`; whereas figure 5.1(b) after interchange has lower `LoopCost(l)`. We observe that in the original loop presented in figure 5.1(a) the accesses to A stride along columns whereas the memory placement in C is row-major. Hence, we need to fetch a new element of A into cache for every iteration of j. Each element of Y is fetched into the cache for each iteration of i and this is not an optimal way to access Y. Each element of X is fetched into the cache only once for the loop nest and this is the optimal way to access X. Hence, two (i.e. A and Y) of the three arrays accessed in this loop are not accessed efficiently where as one (i.e. X) is accessed efficiently. If the two loops are interchanged as shown in figure 5.1(b), two (i.e. A and Y) of the three arrays accessed will be accessed efficiently and one of them (i.e X) is not accessed efficiently. Hence, a faster way to execute this loop is to interchange the two loops.

In this section we present a mathematical method to characterize the number of cache lines accessed by affine accesses and call this the `LoopCost(l)`. This `LoopCost(l)` can be adapted to any search strategy. We adopt it to McKinley's search strategy and this will be presented in section 5.2.

The steps for calculating `LoopCost(l)` are (a) generation of reference groups followed by (b) calculating cache lines for each of these reference groups. In the sections that follow we present each of these steps. The binary adoptions of these algorithms are presented in section 5.1.2.

5.1.1.1 Generation of reference groups

References Ref_1 and Ref_2 are said to belong to the same reference group with respect to a loop nesting level if there is either temporal or spatial reuse between them, detected as follows:

- There is a temporal reuse between Ref_1 and Ref_2, if (a) they have a non-loop-carried dependence (i.e., a dependence within the same loop iteration); or (b) they have a loop-carried dependence which at this nesting depth has a distance vector component which is a small constant d (we have found $d < 10$ to work well), and all other entries in the distance vector are zero.

- There is spatial reuse between Ref_1 and Ref_2 if they refer to the same array and differ by at most d0 in the first subscript dimension of the array indices, where d0 is less than or equal to the cache line size in terms of array elements. All other subscripts must be identical.

Few observations with respect to the reference groups are (i) each data reference belongs to only one reference group; and (ii) any one reference in the reference group can be used to calculate the cache lines, as it completely characterizes the cache behavior of that reference group.

5.1.1.2 Calculating `LoopCost(l)` in terms of cache lines

In this section we describe how the *LoopCost(l,* Ref_k*)* in terms of cache lines is calculated for each reference group Ref_k at each nesting level l. We take one data reference from each reference group (as we have already described any reference in the reference group is representative of its cache behavior) and calculate the contribution towards `LoopCost(l)`. Assume that we are working with a loop nest of the form $L = \{l_1, ..., l_n\}$ and $R = \{\text{Ref}_1, ..., \text{Ref}_m\}$ contains representatives from each of the reference groups. Each Ref_k has array indices of the form $\{f_{k1}(i_1, ..., i_n), ..., f_{kn}(i_1, ..., i_n)\}$ and its contribution to `LoopCost(l)` is decided based on the category it belongs to. The three categories it can belong to are listed below:

- Loop invariant - if the subscripts of the reference do not vary with this loop nesting level l, then it requires only one cache line for all iterations of this loop dimension. These references are loop invariant and have temporal locality. **Mathematically:** $LoopCost(l, Ref_k) = 1$, if none of the f_k values vary with i_l, where i_l is the induction variable associated with loop dimension l

- Consecutive - if only the last subscript dimension (the column) varies with this loop nest , then it requires a new cache line every cls iterations (cls is the cache line size, expressed as the number of elements of this type that fit in a cache line), resulting in $trip_l/cls$ ($trip_l$ is the trip count of this loop dimension l) number of cache lines accessed for this loop nest. These references are consecutive and have spatial locality. **Mathematically:** $LoopCost(l, Ref_k) = trip_l/cls$, if only f_{kn} varies with i_l and all other f_k values do not vary in i_l

- No Reuse - if the subscripts vary with in any other manner, then the array reference is assumed to require a different cache line in every iteration, yielding a total of $trip_l$ number of cache lines. **Mathematically:** $LoopCost(l, Ref_k) = trip_l$, otherwise

Next `LoopCost(l)` is calculated as follows:

$$LoopCost(l) = (\sum_{k=1}^{m} LoopCost(l, Ref_k)) \prod_{h \neq l} trip_h \qquad (l \in [1:n])$$

Then this would be representative of the total cache lines accessed by this loop when it is the inner most loop in this loop nest.

5.1.1.3 Running example

In this section we see how the `LoopCost(l)` algorithm performs on the example presented in Figure 5.1(a). Table 5.1 shows the `LoopCost(l)` that is calculated for this loop where n1 is the trip count of i (outer loop in program order) and n2 is the trip count of j (inner loop in program order).

The `LoopCost(l)` that is calculated is used in the literature in source-based search strategies to optimize the loop. We will present one such search strategy in

reference group	Candidate Inner Loop (i)	Candidate Inner Loop (j)
Y[j]	1/4*n2*n1	1*n2
X[i]	1*n1	1/4*n1*n2
A[j,i]	n2*n1	1/4*n1*n2
loop cost	5/4*n1*n2 + n1	1/2*n1*n2 + n2

Table 5.1: `LoopCost(l)` for the loop in gemver

section 5.2. However, before that we present the calculation of `LoopCost(l)` from a binary in the next section. We will then be able to use the same search strategy from a binary.

5.1.2 From binary

In this section we describe the calculation of `LoopCost(l)` algorithm directly from a binary. The steps for calculating `LoopCost(l)` are (a) generation of reference groups followed by (b) calculating cache lines for each of these reference groups using a representative memory reference from each of the reference groups. In the following sections we will describe the algorithms applied directly to a binary.

5.1.2.1 Generation of reference groups

In this section, we describe how the reference groups are generated from a binary. Unfortunately affine array index expressions $\mathtt{f_k}$ (such as $\mathtt{2i + 3}$ in $\mathtt{A[2i + 3]}$) that are available in source are not available in a binary. Hence, the source techniques cannot be applied directly since the array indices are needed for calculating reuse above in each of the three cases (loop invariant, consecutive and no reuse). Let us assume that there are two references $\texttt{Ref}_1$ and $\texttt{Ref}_2$ in a binary in a loop. Then we can recover the following affine expressions for these references as described in chapter 3 directly from a binary without any symbolic information, where $\mathrm{Base}_{\mathrm{outer1}}$, $\mathrm{Base}_{\mathrm{outer2}}$, $\mathrm{step}_{\mathrm{1k}}$ and $\mathrm{step}_{\mathrm{2k}}$ values are all constants.

$$\texttt{addr\_reg}_1 = \texttt{Base}_{\texttt{outer1}} + \sum_{\texttt{k=1}}^{\texttt{n}} \texttt{num\_k} * \mathrm{step}_{\mathrm{1k}} \tag{5.1}$$

$$\texttt{addr\_reg}_2 = \texttt{Base}_{\texttt{outer2}} + \sum_{\texttt{k=1}}^{\texttt{n}} \texttt{num\_k} * \mathrm{step}_{\mathrm{2k}} \tag{5.2}$$

For any two references $\texttt{Ref}_1$ and $\texttt{Ref}_2$ the following two conditions must be checked to decide if they belong to the same reference group:

- There is a temporal reuse between $\texttt{Ref}_1$ and $\texttt{Ref}_2$ and they belong to the same reference group if
  - (a) the coefficients multiplying all the induction variables is the same for both the accesses; {**Mathematically:** The following condition should be true (i) $\forall l \in [1 : n]$ $\texttt{step}_{11} = \texttt{step}_{21}$ } and
  - (b) the bases of both the accesses differ by a small multiple of the coefficient of the induction variable of the loop nesting level we are considering.

{**Mathematically:** For *e.g.*, if it is loop nest `k`, then the following two conditions should be true (i) $(\texttt{Base}_{\texttt{outer1}} - \texttt{Base}_{\texttt{outer2}})\%\texttt{step}_{\texttt{1k}} = \texttt{0}$ and (ii) $\frac{(\texttt{Base}_{\texttt{outer1}} - \texttt{Base}_{\texttt{outer2}})}{\texttt{step}_{\texttt{1k}}} \leq d$, where `d` is a small number (heuristically set at less than ten).} If the above conditions are true then $\texttt{Ref}_1$ and $\texttt{Ref}_2$ belong to the same reference group and there is temporal locality between them.

- There is a spatial reuse between $\texttt{Ref}_1$ and $\texttt{Ref}_2$ and they belong to the same reference group if

  - (a) the coefficients multiplying all the induction variables is the same for both the accesses; {**Mathematically:** The following condition should be true (i) $\forall l \in [1:n]$ $\texttt{step}_{\texttt{1l}} = \texttt{step}_{\texttt{2l}}$ } and
  - (b) the bases differ by less than the cache line size. {**Mathematically:** The following condition should be true (i) $(\texttt{Base}_{\texttt{outer1}} - \texttt{Base}_{\texttt{outer2}}) <$ `CACHE_LINE_SIZE` }

  If this is the case, both the accesses lie on the same cache line. Hence, there is spatial reuse and they should be placed in the same reference group.

  In all other cases Ref_1 and Ref_2 belong to different reference groups.

Two observations with respect to the reference groups that are formed are (i) each data reference belongs to only one reference group; and (ii) any reference in the reference group can be used to calculate the cache lines, since it completely characterizes the cache lines. These observations remain true from a binary as well.

5.1.2.2 Calculating $\texttt{LoopCost(l)}$ in terms of cache lines

In this section we describe how the $\texttt{LoopCost}(\texttt{l}, \texttt{Ref}_\texttt{k})$ in terms of cache lines is calculated for each reference group at each nesting level and then $\texttt{LoopCost(l)}$ is calculated from it directly from a binary. The intuitions are similar to that from source, but the precise conditions are different in a binary. We take one data reference from each reference group (as we have already described any reference in the reference group is representative of its cache behavior) and calculate the contribution towards $\texttt{LoopCost(l)}$. Assume that we are working with a loop nest of the form $\texttt{L} = \{\texttt{l}_1, \cdots, \texttt{l}_\texttt{n}\}$ and $\texttt{R} = \{\texttt{Ref}_1, \cdots, \texttt{Ref}_\texttt{m}\}$ contains representatives from each of the reference groups. Each $\texttt{Ref}_\texttt{k}$ is of the form $\{\texttt{Base}_\texttt{outerk}, \texttt{step}_\texttt{k1}, \cdots, \texttt{step}_\texttt{kn}\}$ and similar to the source formulation its contribution to $\texttt{LoopCost(l)}$ is decided based on the category it belongs to. The three categories it can belong to are listed below:

- Loop invariant - if the address expression that we obtain from the binary does not contain a term for this loop nest, then it requires only one cache line for all iterations of this loop dimension. These references are loop invariant references and have temporal locality.

  Mathematically: $\texttt{LoopCost}(\texttt{l}, \texttt{Ref}_\texttt{k}) = 1$, if $\texttt{step}_\texttt{kl} = 0$

- Consecutive - if the address expression obtained from a binary has a multiplicative factor $\texttt{step}_\texttt{kl}$ associated with this loop nest level and $\texttt{step}_\texttt{kl}$ is less than the cache line size, then a new cache line is required every $\texttt{ITERS\_IN\_CACHE\_LINE}$ iterations (where $\texttt{ITERS\_IN\_CACHE\_LINE}$ is $\frac{\texttt{CACHE\_LINE\_SIZE}}{\texttt{step}_\texttt{kl}}$), resulting in a total of $\frac{trip_l}{\texttt{ITERS\_IN\_CACHE\_LINE}}$ number of cache lines accessed for this loop nest. These references are consecutive and have spatial locality.

Mathematically: $LoopCost(l, Ref_k) = trip_l/\text{ITERS\_IN\_CACHE\_LINE}$, if $\text{step}_{kl} < \text{CACHE\_LINE\_SIZE}$ and $\text{ITERS\_IN\_CACHE\_LINE} = \frac{\text{CACHE\_LINE\_SIZE}}{\text{step}_{kl}}$

- No Reuse - if the affine expression is of any other form, then the array reference is assumed to require a different cache line every iteration, yielding a total of $trip_l$ number of cache lines accessed.

  Mathematically: $LoopCost(l, Ref_k) = trip_l$, otherwise

Next similar to the source formulation `LoopCost(l)` is calculated as follows:

$$LoopCost(l) = (\sum_{k=1}^{m} LoopCost(l, Ref_k)) \prod_{h \neq l} trip_h \qquad (l \in [1:n])$$

Then this would be representative of the total cache lines accessed by this loop when it is the inner most loop in this loop nest.

5.2 Using Loop Cost in McKinley's Algorithm

In this section we describe how the `LoopCost(l)` calculated in section 5.1 from either source or binary can be used in Mckinley's search strategy [12]. We choose McKinley's search strategy (from traditional affine literature) for implementing and testing the cache reuse model. We can use it in any other decision algorithm as well. We chose the decision algorithm in traditional literature over the one used in the Polyhedral model because of the following.

- Polyhedral models have exponential worst-case performance for their dependence analysis , search strategy and code generation.

- Polyhedral models are very complex to implement.

Of course, the polyhedral model is a powerful compilation framework that works well when compiling purely affine programs from source code. We do not mean to suggest otherwise. We did not choose it for implementation in a binary rewriter for the reasons above, and do not believe we lose much from that choice in this first attempt at cache-based affine transformation of binary code. Further, the contribution of this chapter is to present a cache reuse model from binaries; the decision algorithm is only to test it and is not the major contribution of this thesis.

5.2.1 McKinley's Algorithm

McKinley's algorithm uses `LoopCost(l)` to determine the ideal loop order for the different loop dimensions present in a particular loop nest. It then uses dependence information that we calculated in chapter 3 to obtain the loop ordering (called Optimal order) closest to the ideal order. These steps are further described below. First, we present the main steps of the algorithm and then we describe these in detail in the sections that follow. This algorithm is as presented in [12] and our variation is to perform multi-level blocking. We outline the parts of the algorithm common with McKinley's here to help understand our search strategy; explanations are in subsequent sections.

- Calculate the `LoopCost(l)` associated with each loop dimension in the loop nest as described in section 5.1.

- While disregarding loop-carried dependencies, calculate the *ideal order* of loop dimensions that yields the best data locality in terms of the fewest cache lines accessed.

- Analyze the loop-carried dependencies in the loop and determine the order of loops closest to the ideal order that can be achieved. This loop ordering is called the *optimal order*.

- Strip-mine each loop level in the nesting depth and use a strategy to maximize parallelism.

This algorithm is our improvement on the algorithm presented in [12] since it blocks each level of the nesting loop whereas McKinley's algorithm blocks only one level. This algorithm achieves an ordering of loops that improves data locality and also increases parallelism as we try to interchange the parallel outer strip-mined loop to the outer most position possible. The same decision strategy is used from source and binary.

5.2.1.1 Determining the Ideal Loop Order

Even though `LoopCost(l)` does not directly measure reuse across outer loops, McKinley's algorithm can use it to determine the loop permutation for the entire nest which accesses the fewest cache lines by relying on the following observation:

If loop `i` *promotes more reuse than loop* `j` *when both are considered for the innermost loop,* `i` *will promote more reuse than* `j` *at any outer loop position.* [12]

We therefore simply rank the loops dimensions using their `LoopCost(l)` ordering the loops from outermost to innermost in increasing `LoopCost(l)`. This order is the best theoretical order for this loop nesting which we call the ideal order.

5.2.1.2 Determining the Optimal Loop Order

In this section we describe the algorithm used to obtain the optimal order from the ideal order and the dependencies present in a loop nest. The optimal order of the loops is calculated using the following method. Start from the loop that is outer-most in the ideal order and check if it is legal to position this loop at the outer most position. If yes, place it and examine the next loop in the ideal order. If not examine the next outer-most loop and check if it can be placed in the outer-most position. Repeat until all loops have been placed. This algorithm is presented in figure 5.2.

5.2.1.3 Strip-mining and Interchanging for blocking

In this section we will describe the strip mining and blocking strategy we use. McKinley's algorithm does not use this strategy and we have developed it to enhance the performance. We prefer strip-mining every loop in the nest to increase locality of data accesses. Strip-mining each loop and interchanging the outer of the strip-mined loops decreases the overall footprint of the loop, hence decreasing the run-time. The algorithm that is followed in this part is that we assume the new nest of the loop is (1, 2, ,3 , 4 ...) where even numbers represent the original loops and

```
INPUT:
    $L = \{l_1, ....., l_n\}$ the original loop nest
    $M = \{m_1, ....., m_n\}$ a permutation of loop nests, in this case the ideal order
OUTPUT:
    $P = \{p_1, ....., p_n\}$, the legal permutation of loops closest to ideal order,
        called the optimal order here
ALGORITHM:
    $P = \phi;\ k = 0;\ m = n$
    while $L \neq \phi$
      for $j = 1, m$
        $l = m_j$  working with the $j^{th}$ loop in M
        if direction vectors for $\{p_1, ..., p_k, l\}$ are legal
          $P = \{p_1, ...., p_k, l\}$
          $M = M - \{l\};\ k = k + 1;\ m = m - 1$
          break for
        endif
      endfor
    endwhile
```

Figure 5.2: Algorithm to obtain Optimal Order from Ideal Order

the odd numbers represent the outer of strip-mined loops. The ideal order for this nest will then be (1, 3, ... 2, 4...) representing the fact that the outer-strip need to moved to the outer most position and the original loops need to be as inner-most as possible. We then pass this ideal-order and new direction/distance vectors through the optimal loop order calculation algorithm presented in figure 5.2 to determine the optimal order for this strip-mined nest.

The last step is to determine which of the outer strip-mined loops to parallelize. The algorithm followed here is to look for the outer-most strip that can be parallelized and to bring it to the outer-most level possible. This will help increase parallelism and decrease synchronization.

5.2.1.4 Running Example

Now let us apply the strategy described in section 5.2 to the loop in figure 5.1 using the `LoopCost(l)` calculated in section 5.1.1.3. The `LoopCost(l)` of loop (`i`) is greater than the `LoopCost(l)` of loop (`j`). Hence, the ideal order is $(\mathtt{j}, \mathtt{i})$. This order can be achievable as the interchange is legal for this loop. The original distance vector for this loop nest is $(\mathtt{0}, \mathtt{1})$. Interchanging the two loops transforms the distance vector to $(\mathtt{1}, \mathtt{0})$. One may assume this is sub-optimal since the outer loop cannot be parallelized anymore. However, this is not the complete picture. We still need to perform the strip-mine and interchange step before the algorithm terminates.

Figure 5.3 shows the code of the loop in gemver in the various steps of the

```
for j1 from $1$ to $num\_tiles_j$ step $tile\_size_j$
   for j from $j1$ to $j1 + tile\_size_j$
     for i1 from $1$ to $num\_tiles_i$ step $tile\_size_i$
       for i from $i1$ to $i1 + tile\_size_i$
         X[i] = X[i] + beta * A[j,i] * Y[j]
       end for
     end for
   end for
end for
```

(a) Each loop dimension strip-mined

```
for j1 from $1$ to $num\_tiles_j$ step $tile\_size_j$
   for i1 from $1$ to $num\_tiles_i$ step $tile\_size_i$
     for j from $j1$ to $j1 + tile\_size_j$
       for i from $i1$ to $i1 + tile\_size_i$
         X[i] = X[i] + beta * A[j,i] * Y[j]
       end for
     end for
   end for
end for
```

(b) Loops Interchanged to achieve *ideal order*

```
for i1 from $1$ to $num\_tiles_i$ step $tile\_size_i$
   for j1 from $1$ to $num\_tiles_j$ step $tile\_size_j$
     for j from $j1$ to $j1 + tile\_size_j$
       for i from $i1$ to $i1 + tile\_size_i$
         X[i] = X[i] + beta * A[j,i] * Y[j]
       end for
     end for
   end for
end for
```

(c) Loops Interchanged to maximize parallelism

```
for i1 from $1$ to $num\_tiles_i$ step $tile\_size_i$
   for j from $lb_j$ to $ub_j$
     for i from $i1$ to $i1 + tile\_size_i$
         X[i] = X[i] + beta * A[j,i] * Y[j]
     end for
   end for
end for
```

(d) Loop j rolled back

Figure 5.3: Example Loop to illustrate Strip Mining Algorithm

strip-mine and interchange algorithm. We will explain each step referencing the corresponding code. When we strip mine this loops we generate the following nest $(\texttt{j1}, \texttt{j}, \texttt{i1}, \texttt{i})$ where `j1` and `i1` are the outer stripped loops and `j` and `i` are the original loops. The code is shown in figure 5.3(a). After interchanging to bring the stripped loops to the outer positions the order obtained is $(\texttt{j1}, \texttt{i1}, \texttt{j}, \texttt{i})$ as shown in figure 5.3(b). We observe that the `i1` loop can be parallelized where as the `j1` loop cannot be. Hence, we interchange to obtain the following final nest $(\texttt{i1}, \texttt{j1}, \texttt{j}, \texttt{i})$ shown in figure 5.3(c). This maximizes parallelism since (i1) the outer loop of this new nest can be parallelized. Further, this increases cache performance as the inner loops are in the order that maximizes data reuse. Finally we observe that `j1` and `j` are next to each other in the program and hence it is wise to combine them back into the original `j` loop. This code is shown in figure 5.3(d).

5.3 Handling Imperfect Nests

In this section we describe our method to handle imperfectly nested loops, like the one present in *doitgen* (from Polybench benchmark suite) shown in figure 5.4. McKinley's decision algorithm and the calculation of the *cache reuse metric* are applicable only to perfectly nested loops. Hence, we apply them on the perfectly nested portions of the imperfect nests. For example, in *doitgen*, we apply McKinley's decision algorithm on loop nests ($\texttt{loop}_\texttt{r}$, $\texttt{loop}_\texttt{q}$) and ($\texttt{loop}_\texttt{p}$, $\texttt{loop}_\texttt{s}$) separately since they are the perfectly nested portions of the imperfect nest. We observe that $\texttt{loopcost(s)} > \texttt{loopcost(p)}$. Hence, $\texttt{loop}_\texttt{p}$ should ideally be the innermost loop if legal. We check and find that interchange is legal. Hence, $\texttt{loop}_\texttt{p}$ and $\texttt{loop}_\texttt{s}$ are interchanged. Similarly, $\texttt{loop}_\texttt{r}$ and $\texttt{loop}_\texttt{q}$ can be interchanged if profitable.

```
for r from $lb_r$ to $ub_r$
  for q from $lb_q$ to $ub_q$ {
   for p from $lb_p$ to $ub_p$ {
    for s from $lb_s$ to $ub_s$
     sum[r][q][p] += A[r][q][s] * C4[s][p];
     }
   for p1 from $lb_{p1}$ to $ub_{p1}$
    A[r][q][p1] = sum[r][q][p1];
    }
```

Figure 5.4: Imperfect nest in *doitgen*

Chapter 6

Infrastructure

This chapter provides a detailed description of the infrastructure used for this research. We built an affine automatic parallelizer as part of the thesis. This affine automatic parallelizer can be used in two ways (i) within a static binary rewriting tool, *SecondWrite* and (ii) as a source parallelizer. When used within *SecondWrite* it is able to automatically parallelize binary code without any high-level source information.

First, in section 6.1 we describe the static binary rewriting tool, *SecondWrite* used for this research, which was developed by our research lab @University of Maryland, College Park to implement various binary rewriting projects under one common infrastructure. Automatic parallelization is one of the areas of research that has been developed by me. Others include security addition to binaries, stack variable recovery and variable type recovery to name a few.

Second, in section 6.2 we describe in detail the affine automatic parallelizer developed to test the dependence analysis theory and cache analysis methods described in chapters 3, 4 and 5. We also describe how it interacts with the *SecondWrite* infrastructure.

Third, in section 6.3 we describe how the same affine automatic parallelizer module can be called independently to work on source code.

Last, we show you how the theory presented in chapters 3 and 4 can be implemented in the polyhedral model.

6.1 Implementation-SecondWrite

In this section we describe the binary rewriting infrastructure, SecondWrite [35, 36, 37] used for this research and how the automatic parallelizer interacts with it.

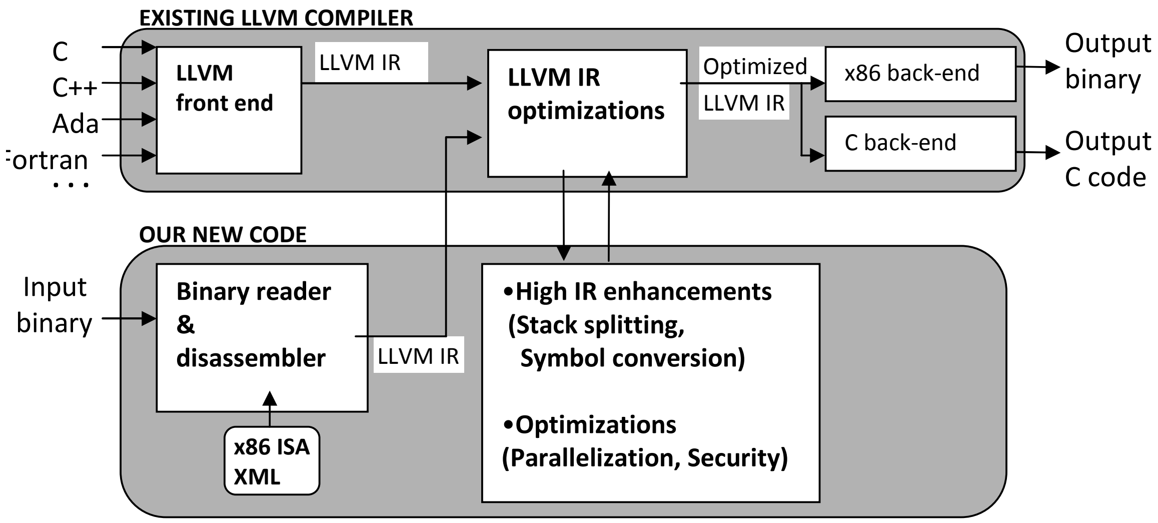


Figure 6.1: SecondWrite

Architecture of Binary Rewriter called SecondWrite is presented in figure 6.1. SecondWrite's custom binary reader and de-compiler modules translate the input x86 binary into the intermediate representation (IR) of the LLVM compiler. LLVM is a well-known open-source compiler [38] developed at the University of Illinois, and is now maintained by Apple Inc. LLVM IR is language and machine independent. Thereafter the LLVM IR produced is optimized using LLVM's pre-existing optimizations, as well as our enhancements, including automatic parallelization. Finally, the LLVM IR is code generated to output x86 code using LLVM's existing x86 code generator.

Currently SecondWrite rewrites x86 binaries. SecondWrite currently successfully rewrites binaries coming from source code totaling over 2 million lines of code, including all of the SPEC2006 benchmarks. The apache web server real-world application (230K+ LOC) is also successfully rewritten. Rewritten benchmark binaries run on average 10% faster than highly optimized input binaries, and 45% faster than unoptimized input binaries because of the existing optimizations in LLVM not including parallelization.

6.1.1 Innovations in SecondWrite

Automatic parallelization takes complex decisions and hence must be static since dynamic rewriters would not be able to pay the cost of complex decisions and transformations at run-time. Unlike existing static rewriters, SecondWrite has the following three functionalities that make it an effective platform for applying advanced program optimizations such as automatic parallelization. First, it rewrites stripped binaries (*i.e.*, those without relocation information) since most real-world binaries are stripped. Second, it rewrites the entire code, not just discoverable parts of it, thus achieving 100% code coverage. Third, it converts the code to compiler IR, since this will enable application of many existing compiler optimizations directly to a binary and some custom optimizations such as automatic parallelization can also be applied to a binary. Below we describe why existing static rewriters do not provide any of these three capabilities, but SecondWrite does. We note that SecondWrite (and any similar tool) does not work with software that is either self-modifying or performs integrity self-checks. Many recent OS do not support self-modifying code and hence we feel that this is not a limitation of SecondWrite.

Rewriting without relocation information: A key innovation in SecondWrite is that it can rewrite stripped binaries, *i.e.*, those without relocation or symbolic information, unlike existing rewriters such as ATOM [39], PLTO [40], Diablo [41], and Vulcan [42] which cannot. The compiler to help the linker in resolving addresses that can change when files are linked generates relocation information. Symbolic information is inserted for debugging. However, production binaries almost never contain such information since linkers delete relocation information by default. Corporations almost never release binaries with relocation information since they are unnecessary for execution.

The requirement for relocation information in existing rewriters arises from the need to update the target addresses of control-transfer instructions (CTIs) such as branches and calls. When rewriting binaries, code may move to new locations because instructions may be added, deleted or changed compared to the original code. Hence the targets of CTIs must be changed to their new locations. Doing so is easy for direct CTIs, since their targets are available in the CTI itself; the target can be changed to its new address in the output binary. However for indirect CTIs, the target may be computed many instructions before at an address creation point (ACP). It is impossible to find all possible ACPs for each CTI using dataflow analysis since they may be in different functions and/or propagated through memory

(memory is not tracked by dataflow analysis.) Hence existing rewriters require relocation information to identify all possible ACPs. Relocation information contains since ACPs are precisely the list of addresses that need relocation during linking.

SecondWrite has devised techniques to rewrite binaries without relocation information. Details are in [43]; here we briefly summarize the intuition of our method. Rather than trying to discover ACPs, SecondWrite relies on inserting run-time checks at indirect CTIs that translate the old target to its new address using metadata tables that store such translations for all possible old branches and call targets. Aggressive alias analysis on the indirect CTI target is used to prune the list of such possible targets to a small number. Further, compile-time optimizations are applied when possible to reduce the number of run-time checks. The result is a method than can rewrite arbitrary binaries without relocation or symbolic information with very low overhead.

Since SecondWrite rewrites binaries without relocation information, it provides a platform to rewrite and parallelize real world binaries that do not have relocation information.

Achieving 100% speculative code coverage: A key challenge in binary rewriters is discovering which parts of the code section in the input binary are definitely code, and thus should be rewritten. This is complicated since code sections often contain embedded data such as literal tables and jump tables which if rewritten by mistake will result in an incorrect program. The only way to be sure a portion of the code section is indeed code is to find a control flow path from the entry point of execution to that portion. However portions of code may be reachable only through indirect CTIs. Unfortunately the possible values of CTI targets cannot be discovered statically in all cases; hence not all code may be discovered. Existing rewriters may not discover all the code, yielding incomplete code coverage undiscovered code cannot be rewritten.

SecondWrite overcomes this problem by doing the following: (i) speculatively rewriting portions of the code segment which cannot be determined to be surely code, thus achieving 100% speculative code coverage; (ii) maintaining a copy of the original code to access the data segment. The detailed scheme is in [43]; but the intuition is that portions of the code segment, which cannot be proven to be code, are speculatively disassembled as if they are code anyway. If the speculative code turns out to indeed be code at run-time, then it is executed, achieving 100% speculative code coverage. Instead, if the speculative code arose from disassembling data bytes, that incorrect speculative code will never be executed since control will never transfer to it at run-time; preserving correctness. Instead the data is accessed from a copy of the original binary maintained in the rewritten binary. Maintaining this code copy increases code-size, but not the I-cache footprint since only the data portions of it are actually accessed, thus run-time is not affected. Since machines today have vastly more resources than even a few years ago, an increase in code size without increasing run-time is tolerable, especially given the payoff of being able to rewrite any binary.

The affine automatic parallelizer within *SecondWrite* parallelizes every loop in the speculatively disassembled code. If this was indeed code, the parallel versions of these loops will be executed. Further, the control flow graph (CFG) obtained from binary code will be conservative, since any edges from speculative code to real code will be represented. CFG is important since they are analyzed to recognize loops for affine parallelization.

Rewriting to compiler IR: Unlike existing rewriters, which represent code in low-level IR, *Secondwrite* employs high IR, making the program easier to analyze and modify. The high-level IR in *SecondWrite* contains features like function arguments, return values, symbols, types, high-level control flow, and an abstract stack frame. In contrast low-level IR has registers and memory locations instead of symbols, no type information, no argument and return value information, and a physical stack frame with an explicit stack pointer.

Low IR has several downsides: (i) low-level code has registers and memory locations instead of symbols, making dataflow analysis much less effective. Hence low-level code requires custom transformations for binaries even for relatively simple dataflow tasks such as the load-store forwarding implemented in PLTO [40] which is equivalent to constant propagation on compiler IR; (ii) low-level code has a physical stack with a precise, fixed layout that is not easy to change. Having a physical stack is problematic since it forces the layout of memory to be retained exactly in the rewritten binary, preventing modifications and optimizations of the stack and global segments, and additions to the stack segment. This is inconvenient for automatic parallelization since some transformations may allocate stack variables.

SecondWrite overcomes these problems by representing the binary code in compiler IR. This makes it easy to run mature compiler transformations already present in LLVM. Many of these compiler transformations make implementing an automatic parallelizer and cache analyzer much easier in *SecondWrite* than an earlier rewriter. Further, *SecondWrite* decouples the physical stack to an abstract stack, which makes modifications to the stack possible. This method is described in [37] and it primarily relies on three technologies. First, high-level program features such as functions, and their arguments and return values are discovered from the binary using deep static analysis. Second, registers and memory locations are replaced by symbols as in high-level programs, allowing easy compiler modification of the memory allocation. Third, type information is recovered when possible by analyzing the uses of each symbol.

The advantages of compiler IR for automatic parallelization are the following: (i) Induction variables of affine loops can be reliably recognized even when they are allocated in memory since high-IR promotion converts them to symbols. Dataflow analysis only works for symbols, not memory locations. Induction variables are allocated to memory either due to register pressure or since basic induction variables have been eliminated by the compiler and the derived induction variables are present in memory; (ii) Scalar variables in loops are now symbols in the IR; hence dataflow analysis on symbols reveals inter-iteration dependencies accurately, rather than the overly conservative dependence analysis on memory locations; (iii) Types recovered from binaries facilitate using induction variable analysis directly from LLVM, instead of writing custom induction variable recognizers; (iv) Layout restrictions are not present; hence new stack variables can be added as required by some loop transformations such as reduction, array privatization etc.; (v) Rigorous inter-procedural data flow analysis can be performed that help take parallelization decisions.

6.2 Affine Automatic Parallelizers

The block diagram of the affine automatic parallelizer developed by us is shown in figure 6.2. We describe these blocks and flow briefly first followed by their detailed description in the following subsections.

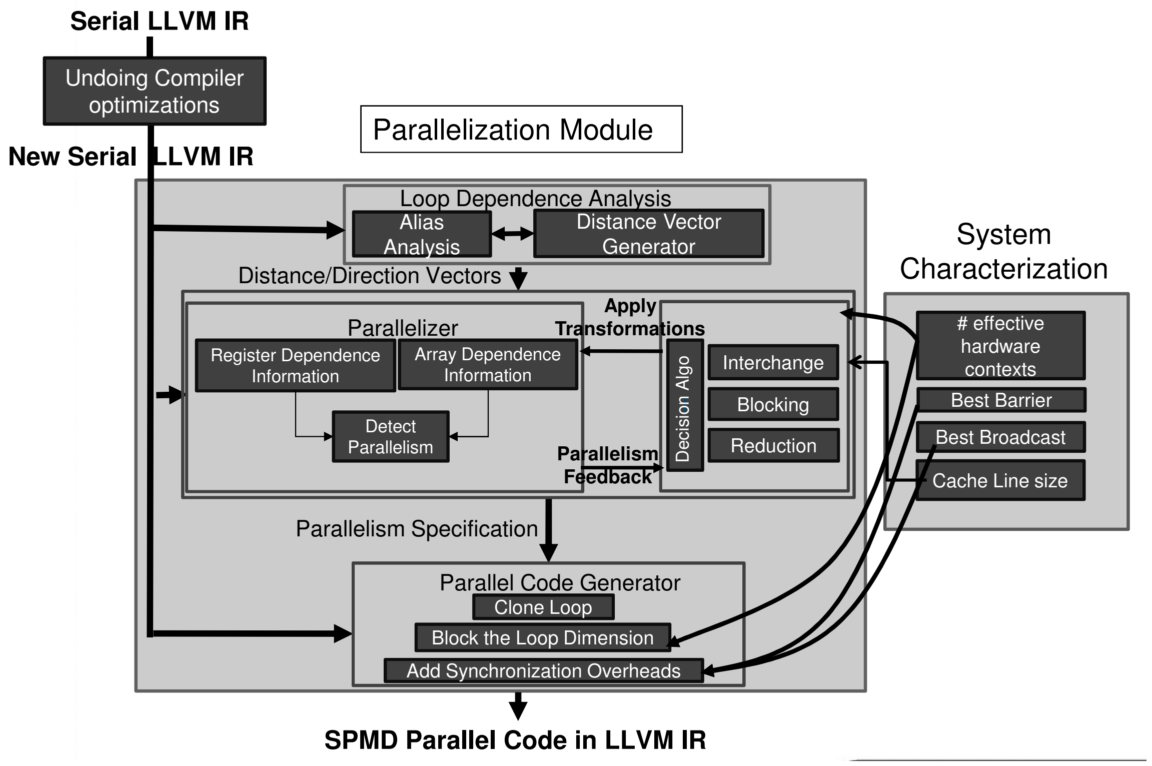


Figure 6.2: Detailed diagram of the Affine Parallelizer

First, the serial llvm IR is fed into the *undoing compiler optimizations* module. The details of the module are presented in subsection 6.2.1. The LLVM IR obtained after this module is still serial, however many of the compiler optimizations are undone and perfectly nested loops are recovered from non-perfectly nested loops. Many of the affine loop transformation algorithms such as McKinley's [12] are effective only on perfectly nested loops. However, when perfectly nested loops from source are compiled using the highest optimizations such as "-O3", then the recovered loop IR from binary code is not perfectly nested any more. Hence, McKinley's algorithm that was effective on the loop from source code is no more effective on the recovered loop from binary code. By using the *undoing compiler optimizations* module these compiler optimizations are undone and perfectly nested loops are recovered from these non-perfectly nested loops making decision algorithms such as McKinley's effective again. The details of the methods used to undo compiler optimizations along with some examples are presented in subsection 6.2.1.

Second, this new serial LLVM IR is then passed into the *loop dependence analysis* block, which consists of the *alias analysis* module and the *distance vector generator*. Every pair of memory accesses in a loop are passed into the *alias analysis* module and the *distance vector generator*. The alias analysis passes that are called in our parallelizer are the standard ones present in LLVM. We did not write any new alias analysis passes. If using the standard alias analysis passes from LLVM we discover that the two references do not alias, we can say that there is no dependence between them and that the distance vector associated with this pair of references is $(0, 0, \cdots, 0)$ consisting of as many zeroes as the loop nesting depth. If alias analysis is unable to prove that the two references do not alias with one another we pass them onto the distance vector module. This module consists of all the theory developed by us in this thesis and presented in chapters 3 and 4. It helps us discover distance/direction vectors for this pair of memory references. After we have analyzed every pair of accesses in the loop we would have generated all the distance/direction vectors associated with this loop. Hence, the output of the *loop dependence analysis* block is all distance/direction vectors associated with each loop.

Third, the distance/direction vectors and the new serial LLVM IR are passed into the *parallelizer* block that talks to the *decision algorithm* block. The decision algorithm used in our parallelizer is modified McKinley's algorithm that has been described in detail in section 5.2 of chapter 5. This decision algorithm talks to the interchange and blocking loop transformations. The details of which have been presented in section 5.2. Once, the ordering of loops in a loop nest has been decided using modified McKinley's algorithm, the parallelizer makes its parallelization decision based on the *register dependence information* and *array dependence information.* Subsection 6.2.2 presents the method used to collect register dependence information. Registers in binary code are scalars in source code terminology. It is important to analyze the register or scalar dependences in a loop since if there is a loop carried scalar dependence in a loop it may prevent parallelization of that dimension. However, some loop carried dependences such as the ones that perform reduction operations do not prevent parallelization. These will also be discussed in detail in section 6.2.2. The paralleizer considers the register dependences present in loops along with the array dependence information (*i.e.*the distance/direction vectors) to make the parallelization decision. The details on how we make the parallelization decision is presented in section 6.2.3. The output of the parallelizer block called the parallelization decision is a list of loops that we have decided to parallelize.

Finally, after the parallelization decision is taken and we have decided which loops to parallelize we pass this information along with the new serial LLVM IR to the *parallel code generator* block which generates SPMD parallel code for each of the parallel loops. The details of the *parallel code generator* are described in subsection 6.2.4.

LLVM plays a central role in our work as our entire infrastructure is built using LLVM. In subsection 6.2.5 we summarize and present the features of LLVM that helped us build the infrastructure.

6.2.1 Undoing Compiler Optimizations in a Binary Rewriter

A binary rewriter runs on code that most likely has been optimized by a compiler. This can be problematic for affine analysis since certain compiler optimizations convert perfectly nested loops to imperfectly nested loops, thereby making many affine decision algorithms such as McKinley's not generally applicable and more restrictive, as we saw in section 5.3. We show in the following three sections certain compiler optimizations that convert perfectly nested loops in source to non-perfectly nested loops when reconstructed from a binary and hence preventing standard affine technologies to be applied to them and how we handle these cases.

6.2.1.1 Undoing LICM

Loop invariant code motion(LICM) can convert a perfectly nested loop in source to a non-perfectly nested loop when recovered from a binary. In figure 6.3(a)

```
for i from $lb_i$ to $ub_i$
  for j from $lb_j$ to $ub_j$
    for k from $lb_k$ to $ub_k$
      C[i,j] += A[i,k] * B[k,j]
    end for
  end for
end for
```

(a) *Original loop in matrix multiply*

```
Loopi :  .....
  Loopj :  .....
    tmp = load &C[i,j]
    LoopK : .....
      tmp += &A[i,k] * &B[k,j]
    end LoopK
    store tmp to &C[i,j]
  end Loopj
end Loopi
```

(b) *Equivalent O3 Binary*

```
Loopi :  .....
  Loopj :  .....
    LoopK : .....
      tmp = load &C[i,j]
      tmp += &A[i,k] * &B[k,j]
      store tmp to &C[i,j]
    end LoopK
  end Loopj
end Loopi
```

(c) *Compiler LICM undone*

Figure 6.3: Example to illustrate LICM Compiler Optimization undone

we show the C code of a matrix multiply kernel and in figure 6.3(b) its corresponding pseudo binary code from "-O3" optimizations of gcc. The load of `C[i, j]` is moved to the loop nest `j` thereby making it an imperfectly nest. We recognize these cases and perform what we call "undoing compiler optimizations", where we transform the code in figure 6.3(b) to that in figure 6.3(c). The transformation undoes the effects of LICM by using the well-known *forward propagation* compiler pass on affine accesses that are not in the inner most loops, to forward propagate them to the inner-most loop. Thereafter the loop becomes perfectly nested and cache analysis and McKinley's decision algorithm can be applied on it. After we take decisions about interchanging loops and perform the transformations, we apply standard compiler optimizations including LICM to regain its performance advantages.

An added bonus is that since we use "undoing compiler optimizations" on IR, we convert non-perfectly nested loops even in source code to perfectly nested loops, which increases the scope of loop interchange. Such ideas are not novel [33], but the need for conversion of imperfectly to perfectly nested loops is greater in a binary, since compilers often do LICM which needs to be undone.

6.2.1.2 Undoing Complex Control flow

Compiler optimizations convert perfectly nested loops in source to non-perfectly nested loops in a binary to reduce the total number of actual branches executed. One such example code is presented in figure 6.4(a) and the recovered CFG from a highly optimized binary is present in figure 6.4(b). The optimized control flow ensures that only two branch instructions are executed for every iteration of the loop; however the loop is no longer a perfectly nested. We perform what LLVM calls "loop simplification" on this loop to obtain the control flow shown in figure 6.4(c) by merging the multiple basic blocks with back edges whenever legal. This loop is now perfectly nested and affine analysis can be applied on it. We apply the standard set of optimizations present in LLVM when we convert high IR back to the binary and we regain any transformation that does control flow simplification on loops. Hence, we obtain the most optimal transformed loop in the rewritten binary.

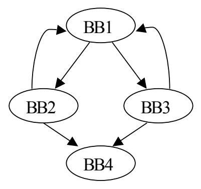
"Loop Simplification" is a standard pass in the LLVM compiler that we run on the raw IR we obtain from binary. It performs the following two transformations: (i) merges multiple exit basic blocks from a loop into one exit basic block whenever legal; (ii) merges multiple back edges of a loop into one back edge whenever legal. "Loop Simplification" or any such equivalent pass needs to be applied to compiler IR obtained from binaries to enable affine transformations on binary code.

6.2.1.3 Discovering induction variables

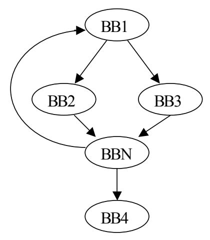
Sometimes the compiler decides to place the address calculation in registers in such a way that we cannot use the induction variable analysis of compiler theory directly on the reconstructed high compiler-IR. For example figure 6.5(a) shows the source code of an affine loop and figure 6.5(b) shows the pseudo binary code for it. The compiler has used `Reg1` as the increment to the `addr_A`; and it is equivalent to `Base_A` in the first iteration and `elem_size` thereafter. When this is reconstructed

```
for i from $lb_i$ to $ub_i$
  BB1:  if(A[i] mod 3)
   BB2:  A[i] = func1(A[i]);
  else
   BB3:  A[i] = func2(A[i]);
end for
BB4:  Outside the loop
```

(a) *Original loop in Source Code*



(b) *CFG recovered from Binary*



(b) *CFG after loop simplification*

Figure 6.4: Example loop to illustrate undoing complex control flow generated by compiler optimizations

in compiler IR (shown in figure 6.5(c)), `addr_A` is no longer an induction variable that can be recognized by the standard compiler induction variable analysis, since there is an phi instruction to represent the different increments to `addr_A`. We recognize all such cases and transform figure 6.5(c) to figure 6.5(d) by adjusting the increment of the induction variable to `elem_size` in every iteration. This can be done by initializing `addr_A` to `Base_A - elem_size` before the loop, thereby making it an induction variable that can be recognized by the standard compiler analysis.

6.2.2 Register dependencies

A register or scalar dependence is present in a loop if a location is defined in an iteration of the loop and used in another iteration of the loop. Conceptually, detecting and handling scalar dependencies is similar in source code and binaries. One minor difference is that whereas the possibly dependent locations in source

```
for i from $lb_i$ to $ub_i$
    A[i] = 8;
end for
```

(a) *Original loop in Source Code*

```
Reg1 = Base_A;
addr_A = 0;
    Loopi:  .....
    addr_A += Reg1;
    &addr_A = 8;
    Reg1 = elem_size;
```

(b) *Loop from Binary*

```
Reg1_1 = Base_A;
addr_A = 0;
    Loopi:  .....
    Reg1_3 = $\phi$(Reg1_1, Reg1_2);
    addr_A += Reg1;
    &addr_A = 8;
    Reg1_2 = elem_size;
```

(c) *Loop in compiler IR*

```
Reg1 = elem_size;
addr_A = Base_A - elem_size;
    Loopi:  .....
    addr_A += Reg1
    &addr_A = 8;
```

(d)*Loop with induction variable*

Figure 6.5: Example Loop to illustrate introducing induction variables

code are variables, in binaries they are registers and memory locations. Our scalar dependence analysis for binaries is outlined below.

We recognize register/scalar dependencies from a binary by analyzing def-use chains. All registers are checked to see if they are defined in an iteration and used in a later iteration. This is a check on def-use chains of registers, to see if the register is live at the exit block of the loop. Traditional data flow can be run on low-level code from binaries. We leverage this to check the presence of loop-carried register dependencies. We check for the presence of register dependencies at every loop depth, as certain dependence may be present at one depth and not at another loop depth. For example in the code in figure 6.6 *tmp* has a scalar dependence on $\texttt{Loop}_\texttt{j}$, however there is no scalar dependence on $\texttt{Loop}_\texttt{i}$. Hence $\texttt{Loop}_\texttt{i}$ can be parallelized in this code. Variable *tmp* may be register allocated in a binary (say to `tmp_r`) and data flow will tell us that it is live across the $\texttt{Loop}_\texttt{j}$ but not live across $\texttt{Loop}_\texttt{i}$.

Some variables in the source code may be allocated to memory; these will be analyzed as memory references by theory presented in chapter 3. They will likely appear as addresses with a constant base and no offset. The theory will handle their dependencies in a simple, degenerate case. However the dependencies get analyzed, every dependency that is present in a binary is analyzed and its effect on parallelization is accounted for.

```
for i from $lb_i$ to $ub_i$
  tmp = 0;
  for j from $lb_j$ to $ub_j$
   A[i,j] = A[i,j] + tmp;
   tmp = B[j] + 10;
  end for
end for

(a) Source code with scalar
dependence
```

```
$Loop_i$ :  ............
  tmp_r = 0;
  $Loop_j$ :  .........
   adr1_r = *addr1
   *addr1 = adr1_r + tmp_r
   adr2_r = *addr2
   tmp_r = adr2_r + 10
  end for
end for

(b) Binary Code with scalar dependence
```

Figure 6.6: Example of a loop that carries a scalar dependence

6.2.2.1 Special case of scalar dependence: Reduction

Certain scalar loop carried dependencies such as reduction do not prevent parallelization as known in affine literature [6]. One example loop that contains a reduction on the variable `sum` is presented in figure 6.7. Figure 6.7(a) shows the source code with a reduction and figure 6.7(b) shows the binary code for the same. For every scalar variable that is live across the loop, we check to see if this is due to a reduction operation. Reduction operations known and implemented from traditional affine technologies are sum, product and min/max operations. Once such scalar values are recognized using the standard rules of reduction, this scalar value is marked as reduction and no more prevents parallelization. For *e.g.*, the variable `sum` is marked as a reduction on $\texttt{Loop}_\texttt{j}$. If this loop level is chosen for parallelization then code is generated such that each parallel thread accumulates a part of `sum` and after all the parallel loops have executed they are all added up. Reduction is a standard transformation and has been explained in detail in [6].

```
for i from $lb_i$ to $ub_i$
  sum = 0;
  for j from $lb_j$ to $ub_j$
   sum = sum + A[i,j]
  end for
end for
```

(a) Source code with scalar dependence

```
$Loop_i$ :  ............
  sum_r = 0;
  $Loop_j$ :  .........
   tmp_r = *addr1
   sum_r = sum_r + tmp_r
  end for
end for
```

(b) Binary Code with scalar dependence

Figure 6.7: Example of a loop that carries a reduction dependence

6.2.2.2 Special case of scalar dependence: Values carried across loop

Sometimes when high IR is generated from binaries using *SecondWrite* we observe some spurious ϕ instructions that carry values across the loop when they are defined before the loop and used after it. One such example source code is shown in figure 6.8(a) and the binary code for it is presented in figure 6.8(b). Here the variable `tmp` has been defined before the loop and the actual use of this variable is after the loop. However, in binary the register for it $\mathtt{tmp_r}$ has been passed across the loop using a ϕ instruction without any real use of it in the loop. We have seen such code in IR generated from large benchmarks such as SPEC2006 and OMP2001. This is an artifact of our binary rewriter, *SecondWrite*. We recognize these spurious loop

```
tmp = ....
for $lb_i$ to $ub_i$, step 1
  Body of $Loop_i$
end for
.... = tmp
```

(a) Source code

```
$tmp_r$ = ....
$Loop_i$: ....
  $tmp1_r = \phi(tmp_r, tmp2_r)$
  $tmp2_r = tmp1_r$;
end $Loop_i$
.... = $tmp1_r$
```

(b) Binary code

Figure 6.8: Example loop showing a scalar value defined before a loop and used after it

carried dependences from binary and mark them as not hampering parallelization. If we parallelize these loop levels we do not do any special code generation for these since the value in the first thread is the one that can be used after the loop as well.

6.2.3 Deciding Partitions

As we have shown in chapter 3 section 3.1 for the code in figure 3.1(a) the dependence vector `vecD` = (1,0) and for the code in 3.1(c) the dependence vector is $\tilde{\mathtt{D}}$ = (2,0), indicating that there is a dependence along `i`, whereas there is no dependence along induction variable `j`. So, if we execute all iterations of `i` on one processor then we can parallelize the iterations along `j` among all the processors. Pictorially, this is represented as partition 1 in figure 6.9, which shows the iteration space as a 2-D matrix of `i` and `j` values. Conversely, in figure 3.1(b) the dependence vector is $\tilde{\mathtt{D}}$ = (0,2), indicating that there is a dependence in steps of two along induction variable `j`, and no dependence along induction variable `i`. So, if we execute

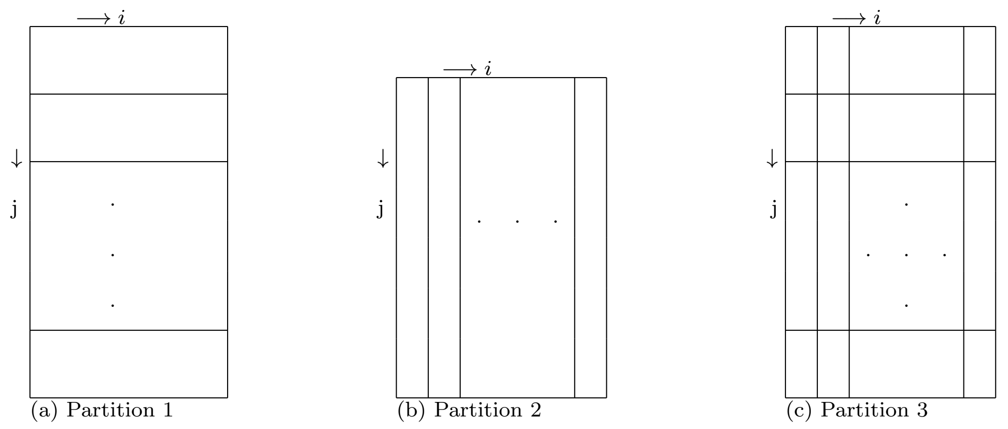


Figure 6.9: Different partitions of the iteration space

all iterations of `j` on one processor then we can parallelize the iterations along `i` among all the processors. Pictorially, this is represented as partition 2 in figure 6.9. Partition 3 in that figure can be used when there is no loop-carried dependence on either loop dimension (i.e. $\tilde{D}$ = (0,0)).

Algorithm 5 Algorithm to decide which loop dimensions to parallelize

```
Input: All loops in the program
Input: Register dependence information, Array dependence information
Output: Loop dimensions to parallelize
for all $Loop_i$ in the program do
    if $Loop_i$ is parallel based on register & array dependence information then
        if None of the parent loops of $Loop_i$ are parallel then
            $Loop_i$ is added to list of loops to be parallelized
        end if
    end if
end for
```

However, it is not always this simple when dealing with arbitrarily nested loop dimensions. It is essential to have a algorithm to effectively decide which loops to parallelize to maximize speedup from parallelization. Since we have already applied a decision algorithm to transform loops to maximize cache reuse, we would gain maximum from parallelizing the outer most dimensions that is parallel in loop nests. The algorithm used to determine the outer most dimensions to parallelize in arbitrarily nested affine loops is presented in algorithm 5. Essentially, using the algorithm we choose all loops that can be parallelized for which none of the parent loops is parallel. These loops are added for parallelization. A loop level is considered parallel if all distance/direction vectors associated with this loop have a 0 component for it and there is no parallelization preventing loop carried scalar dependence on it.

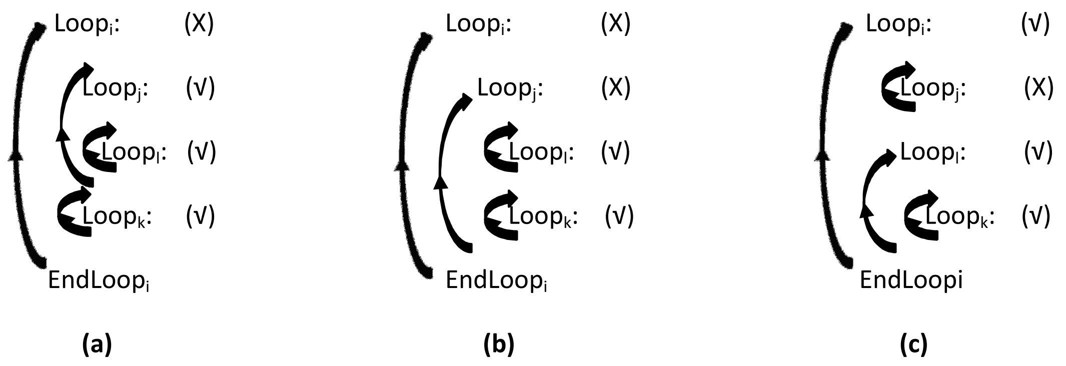


Figure 6.10: Arbitrarily nested affine loops to illustrate which loops will be parallelized by our decision algorithm

We now illustrate which loops our algorithm will parallelize for the three arbitrarily nested loops shown in figure 6.10. In each of the example loops, `Loop`$_i$ is the outermost loop and for each loop level we have marked (X) to indicate that the loop is not parallel and ($\surd$) to indicate that the loop level is parallel. For the loop structure shown in figure 6.10(a) we parallelize loop dimensions $\texttt{Loop}_\texttt{j}$ and $\texttt{Loop}_\texttt{k}$. We do not recommend $\texttt{Loop}_\texttt{l}$ for parallelization since one of its parent loop $\texttt{Loop}_\texttt{j}$ can be parallelized and has been recommended for parallelization. Next, for the loop structure in figure 6.10(b) we recommend loops $\texttt{Loop}_\texttt{l}$ and $\texttt{Loop}_\texttt{k}$ for parallelization since none of the parent loops of these two loops $\texttt{Loop}_\texttt{i}$ and $\texttt{Loop}_\texttt{j}$ are not parallel. Finally, for the loop structure in figure 6.10(c) only $\texttt{Loop}_\texttt{i}$ is recommended for parallelization since it is the outermost loop and is parallel. $\texttt{Loop}_\texttt{l}$ and $\texttt{Loop}_\texttt{k}$ are both parallel, but are not recommended for parallelization since one of their parent loops $\texttt{Loop}_\texttt{i}$ is parallel.

6.2.4 Code Generation

After the distance/direction vectors are calculated, transformations done, parallelization decision taken, and the loop dimensions to be parallelized are decided, code needs to be generated for each parallel loop dimension. Since the body of the loop is executed on all parallel threads, the most convenient and efficient code generation model is the Single Program Multiple Data (SPMD) model. The underlying idea is that the iterations of the loop are divided among threads; hence to keep the code-size increase to a minimum, the same code is executed on all threads using different loop bounds.

From source code, simply replacing the symbolic values of the lower and upper bounds of loop induction variables by new values can generate SPMD code. These methods are fairly straight forward as the symbolic information is readily available in source.

From binary code, code generation is conceptually similar to that from source. For each loop dimension to parallelize we calculate the new lower and upper bound using the formula below.

$$\begin{aligned}\texttt{new\_lb}_{\texttt{addr\_reg}} = {} & \texttt{Base}+\texttt{lb}_{\texttt{i}} * \texttt{size\_j} + \texttt{lb}_{\texttt{j}} * \texttt{elem\_size} \\ & + \frac{\text{PROC\_ID} * (\texttt{ub}_{\texttt{j}} - \texttt{lb}_{\texttt{j}}) * \texttt{elem\_size}}{\text{NPROC}}\end{aligned} \tag{6.1}$$

$$\texttt{new\_ub}_{\texttt{addr\_reg}} = \texttt{min}(\texttt{ub\_j}, \texttt{new\_lb}_{\texttt{addr\_reg}} + \frac{(\texttt{ub}_{\texttt{j}} - \texttt{lb}_{\texttt{j}}) * \texttt{elem\_size}}{\text{NPROC}}) \tag{6.2}$$

Replacing the bounds in (B) and (C) generates the parallel code to be executed on all NPROC processors. If the outer loop is partitioned, then statements (E) and (F) are similarly modified. Unlike loop partitioning, data partitioning is not necessary since we target shared memory platforms common in multi-cores.

Generating parallel code requires the use of some parallel thread library. We implement POSIX-compliant *pthreads* calls, given that POSIX is a widely used portable industry standard, although any library can be used. POSIX-complaint parallel threads are created once at the start of *main()* in the binary, rather than at each loop to avoid paying the steep thread-creation cost multiple times. Only the main thread executes serial code between parallel loops. Parallel threads only execute loop code. When a parallel thread finishes one loop it waits for the main thread to inform it which loop to execute next in a broadcast. The broadcast also contains the values of registers calculated by the main thread that are needed by the parallel loop threads. A barrier is inserted into the binary at the end of every loop.

We are also using the barrier most profitable for the machine we are working on. We have implemented a central, tree and butterfly barrier implemented. We also compare these to the platform specific barrier present on any systems (such as pthreads_barrier). Our collaborator in the past has provided us with micro benchmarks that tells us, which is the best broadcast or barrier to use. It also tells us the maximum number of threads that can run on a particular machine. We use this information to make sure we do not run more threads than the maximum available to parallelize the loop using the barrier and broadcast most profitable for this machine.

6.2.5 Using LLVM for Implementation

Our binary rewriter translates the input x86 binary to the intermediate format of the LLVM Compiler [38], and then uses the x86 back-end LLVM to write the output binary. LLVM, which stands for Low-Level Virtual Machine, is a well-known, open-source compiler developed at the University of Illinois; it is now maintained by Apple Inc. *This conversion back to compiler IR is not a necessity for the work we present in this thesis; any binary rewriter can use our theory.* However using LLVM IR enables us to use LLVM's rich infrastructure, such as control-flow analysis, dataflow analysis, and optimization passes, so that we did not have to write our own for the rewriter. Each instruction in the binary is converted to its equivalent LLVM IR instruction. The pushes and pops are analyzed to determine function arguments, caller and callee saves and stack accesses. Each stack frame present in the original binary is converted to an stack array in the intermediate IR. These techniques enable the addition of new stack variables in functions, which are required for loop transformations such as reduction. The globals are accessed from their original addresses as we retain the original segments. Register allocated variables in the binary are converted to virtual registers.

The “Loop Simplify”, “Loop Unswitch” and “Induction Variable Simplification” passes in LLVM are run to help the scalar evolution and induction variables on our code. Scalar evolution within induction variable analysis in LLVM [38] helps us in identifying the induction and derived induction variables in code. This helps us determine (A) to (F) required for distance vector calculation as described in chapter 3. We also use the control flow and data flow information present in LLVM to our leverage to identify scalar dependencies, affine loops and shared/private variables for each loop that will be broadcasted to the parallel threads.

An important side benefit of using LLVM is that it enabled us to do cross-ISA translation of code. We used LLVM’s C backend to convert an input x86 binary to equivalent functional C code. This code (which is parallel using *pthreads* in our case) was then compiled using GCC on a 64-threaded SPARC T2 machine, and speedup was measured (see results section). The reason we did this was because *SecondWrite* presently implements only an X86 front-end. Of course this cross-ISA translation will not work in the general case when the code uses machine-specific library calls. However it worked for our programs since they only used only the platform-independent C and *pthreads* libraries.

To be clear, our LLVM's output C code generated from binaries is quite low-level and lacks array declarations and index expressions. Hence source parallelism methods will not work on it, necessitating our method.

6.3 Source Parallelizer: AESOP

We also have a stand-alone version of the affine automatic parallelizer that works on the source code of programs. This is called AESOP. It is open source and can be downloaded from @aesop.ece.umd.edu. It has been developed by Timothy Creech who is a PhD student at University of Maryland, College Park and I. Two versions of it are available for download: (i) source files for it can be downloaded and built on your machine. It reliably works on 32-bit x86 Linux, however parts of AESOP may work wherever LLVM works; and (ii) binary version of it is available which can be directly used on any 32-bit Linux box.

The source affine parallelizer contains all the blocks described in section 6.2, *i.e.*the *undoing compiler optimizations* block, the *loop dependence analysis block*, the *parallelizer* and the *code generator*. The serial LLVM IR from source code is first passed through the compiler deoptimizations passes, this is not essential for source code, however a good idea to have since it may convert any non-perfectly nested loops from source code to perfectly nested ones in IR. Then the new serial LLVM IR is passed into the *loop dependence* block where distance/direction vectors are generated, then a parallelization decision is taken and parallel code generated for it.

We now explain briefly how AESOP works. We know that by feeding serial LLVM IR to the standalone version of the affine parallelizer we can obtain SPMD parallel LLVM IR. We have three scripts that help us do this as part of AESOP for source code: (i) *aesopcc* which uses *clang* [44] (a C language front-end for llvm) to generate LLVM IR for source code written in C and then feeds it to the affine parallelizer; (ii) *aesopgcc* that uses *dragonegg* [45] plugin (a plugin that integrates the LLVM optimizers and code generator with GCC) to generate LLVM IR from C/C++ source code programs and feeds it into the affine parallelizer and (iii) *aesopgfort* that uses *dragonegg* [45] plugin to generate LLVM IR from fortran programs and feeds it to the affine parallelizer. All the three scripts use the x86 back-end of LLVM to obtain parallel executable for the source code fed to them.

We have tested AESOP on benchmarks whose source code exceeds 2 million lines of code. The testing infrastructure is available for download along with AESOP and contains benchmarks from polybench, SPEC2006, OMP2001, NPB and hpcc. There is also a very easy method to add new benchmarks to AESOP. The barriers used for generating parallel code are in a library and is also available for download with AESOP.

6.4 Implementation in the Polyhedral model

In this section we consider whether we can apply the polyhedral model and its decision algorithm strengths to the theory presented in this thesis. We currently use the traditional techniques to present our implementation and results.

The polyhedral model requires array indices to be able to represent every dynamic iteration of the loop it in the polyhedron space. However, we recover linearized multi-dimensional expressions for every memory address from binary code of the form 3.13:

$$\texttt{addr\_reg} = \texttt{Base}_{\texttt{outer}} + \sum_{\texttt{k=1}}^{\texttt{n}} \texttt{num\_k} \times \mathrm{step}_{\mathrm{k}} \tag{6.3}$$

where, $\texttt{Base}_{\texttt{outer}}$ and $\texttt{step}_{\mathrm{k}}$s are loop invariant constants and $\texttt{num}_{\mathrm{k}}$ represents induction variables.

If we view the entire memory system as one unit (M), then the address expression we recover can be looked at as:

$$\texttt{addr\_reg} = \texttt{M}[\texttt{Base}_{\texttt{outer}} + \sum_{\texttt{k=1}}^{\texttt{n}} \texttt{num\_k} \times \mathrm{step}_{\mathrm{k}}] \tag{6.4}$$

"S" can replace "M" if it is the stack segment, "G" if it is the global segment and "H" if it is heap allocated since these can be viewed as three different arrays with $\texttt{Base}_{\texttt{outer}}$ and $\texttt{step}_{\mathrm{k}}$s representing the affine indices to be fed into the polyhedral system.

There has been some prior work that has tried this in the polyhedral system [46] [47]. They show that just using these recovered equations as indices to the PLUTO system does not work since it is not able to handle the large numbers present as $\texttt{Base}_{\texttt{outer}}$ and some of the $\texttt{step}_{\texttt{k}}$s. They have proposed that limited array delinearization techniques be applied to reduce the coefficients in the memory expressions. In general we think that this is only practical for small kernels from polybench, - however, will not scale well to larger benchmarks. Their results are also primarily limited to the polybench benchmark suite. Further, they show that the scalar variables created when source code is compiled to binaries limited the use of PLUTO. It is well understood that many scalar variables are created from binaries since registers are used to store temporaries, stack variables etc. They say that one must perform scalar variable merging and removal to make PLUTO applicable on the benchmark. Again, we see this as very limiting since scalar variables can easily be studied in the traditional literature and do not generally prevent parallelization unless they are loop carried which actually makes the loop non-parallel. Further, their results on the benchmarks *swim* and *mgrid* (the only benchmarks not from polybench) show that the binary speedup is half that of the source since they have not been able to completely remove the scalar dependencies in the loops whereas using the traditional techniques we have been able to scale as well from binaries as from source since we study all the scalar dependencies using data-flow algorithms.

In general we have to believe that one can implement the polyhedral decision algorithm in our system but it poses significant challenges (which have atleast so far been surpassed by small non-reliable fixes) and even then not provided significant speedups like from source for the larger benchmarks.

Chapter 7

Results

Results for this thesis were collected in three stages during the progress of the thesis. First, results were collected after the basic dependence analysis presented in chapter 3 was implemented. These results were collected on polybench-1.0 the dense matrix benchmark suite. Second, results were collected after the cache reuse metric was generated from binaries and used within the McKinley algorithm as shown in chapter 5. The results for this were collected on polybench-1.0 as well. Third, results were collected on the SPEC2006 and OMP2001 benchmark suites after the algorithm to guess loop bounds was implemented. The reason this was so is that the algorithm to guess loop bounds was implemented to test larger benchmarks coming from SPEC2006 and OMP2001. These three sets of results are presented in the following three sections.

7.1 Results for Dependence Analysis

In this section, we present the results for the dependence analysis mechanisms presented in chapter 3.

The input to our binary parallelizer is highly optimized (-O3) binaries compiled by GCC. These binaries don't contain any relocation or symbolic information. We have tested our parallelizer on benchmarks from *Polybench* (the Polyhedral Benchmark suite) and *Stream*(from the HPCC suite). We used three different machines to test our benchmarks. The machine descriptions are provided in table 7.1. The benchmarks represent heavily used kernels in scientific and multi-media workloads.

Name	*CPUs*	*Cores/CPU*	*Threads/Core*	*Total Threads*	*Manufacturer*	*Model*
DASH	1	4	2	8	Intel	Xeon E5530
BUZZ	4	6	1	24	Intel	Xeon E7450
T2	1	8	8	64	Sun	Ultra SPARC T2

Table 7.1: Test machines to test dependence analysis

Our source parallelizer is implemented by feeding the parallelizer with the symbolic information present in a source. We then apply the same dependence analysis, partition techniques and code generation methodologies we have presented and used for binaries. As the symbolic information is exploited to the fullest, we compare to state of the art affine parallelizers.

The speedups when parallelizing source and when parallelizing binaries with increasing number of threads on the different machines are presented in tables 7.1,7.2, and 7.3; one figure per machine. Table 7.1 shows the geomean of the speedup averages 5.57X when parallelizing from source code versus 4.61X when parallelizing from the x86 binary on the x86 DASH machine with 8 threads. This shows that (a) the speedups are nearly as effective from binaries as from source code, validating our theory; and (b) the speedups scale well. Table 7.2 shows the geomean of the speedups from source and binary on 24 threads on the x86 BUZZ machine are 6.27X and 5.15X respectively. The speedups on BUZZ scale less well than DASH beyond 4 cores since the communication is out of the chip beyond 6 threads.

Benchmark		1	2	4	8
2mm	Source	1	1.85	3.75	4.78
	Binary	0.98	1.76	3.59	4.85
atax	Source	1	1.42	2.01	3.54
	Binary	0.75	1.19	1.91	3.05
covariance	Source	1	1.29	2.3	3.98
	Binary	0.98	0.94	0.97	0.91
gemver	Source	1	1.73	3.14	7.53
	Binary	0.9	1.64	3.03	7.25
jacobi-2d	Source	1	1.9	3.4	7.4
	Binary	0.94	1.43	2.82	5.64
3mm	Source	1	1.8	3.64	5.01
	Binary	0.99	1.75	3.6	4.89
bicg	Source	1	1.93	3.53	6.47
	Binary	1.1	1.63	2.99	5.73
doitgen	Source	1	1.8	3.56	7.65
	Binary	0.99	1.76	3.36	7.85
gesummv	Source	1	1.67	2.82	6.86
	Binary	0.93	1.31	2.36	5.68
correlation	Source	1	1.31	2.21	4.19
	Binary	0.96	1.31	2.1	3.77
gemm	Source	1	1.84	3.67	5.6
	Binary	1.01	1.77	3.65	5.19
stream	Source	1	1.89	3.61	5.91
	Binary	1.01	2	3.77	6.77
Geo Mean	Source	1.00	1.69	3.07	5.57
	Binary	0.96	1.51	2.68	4.61

Figure 7.1: Speedup on x86 DASH for source and binary using dependence analysis techniques

Benchmark		1	2	4	8	16	24
2mm	Source	1	1.97	3.83	6.32	11.79	15.6
	Binary	1	1.96	3.75	5.96	10.52	13.53
atax	Source	1	1.57	2.42	3.27	3.5	3.09
	Binary	0.75	1.01	1.72	2.47	2.9	2.73
covariance	Source	1	1.31	2.22	4.08	7.49	10.05
	Binary	1	1.89	3.42	4.85	4.23	2.8
gemver	Source	1	1.48	2.56	3.81	4.18	4.08
	Binary	0.93	1.43	2.42	3.8	4.34	4.02
jacobi-2d	Source	1	1.89	3.29	5.02	5.01	4.56
	Binary	0.63	1.24	2.16	3.61	4	4.02
3mm	Source	1	1.97	3.82	6.34	11.85	15.82
	Binary	1	1.98	3.83	6.35	11.6	15.23
bicg	Source	1	2.1	3.55	4.59	4.31	3.97
	Binary	0.83	1.76	3.08	3.88	3.36	3.01
doitgen	Source	1	1.99	3.98	6.58	12.59	15.7
	Binary	0.99	1.98	3.95	6.6	12.53	16.12
gesummv	Source	1	1.54	2.35	3.38	2.87	2.73
	Binary	0.74	1.11	1.8	2.49	2.89	2.18
correlation	Source	1	1.32	2.24	3.96	7.17	9.86
	Binary	1	1.29	2.12	3.51	5.74	7.46
gemm	Source	1	1.97	3.89	6.59	11.29	15.67
	Binary	1	1.95	3.8	6.28	11.52	15.11
stream	Source	1	2.29	3.14	3.47	1.87	0.98
	Binary	0.95	1.99	2.97	3.12	2.02	1.16
Geo Mean	Source	1	1.76	3.03	4.62	5.91	6.27
	Binary	0.89	1.59	2.80	4.17	5.23	5.15

Figure 7.2: Speedup on x86 BUZZ for source and binary using dependence analysis techniques

Benchmark		1	2	4	8	16	32	64
2mm	Source	1	1.99	3.99	7.96	15.21	26.31	36.95
	Binary	1.03	2.05	4.09	8.14	15.65	27.44	39.62
atax	Source	1	1.4	1.74	2	1.97	1.65	0.83
	Binary	1.04	1.44	1.79	2.01	1.85	1.85	0.86
covariance	Source	1	1.34	2.29	4.26	8.12	15.3	23.25
	Binary	1	1	1	1	0.99	0.93	0.81
gemver	Source	1	1.99	3.92	7.42	11.97	14.84	9.65
	Binary	0.99	1.97	3.74	7.12	12.1	15.53	7.63
jacobi-2d	Source	1	1.83	3.46	5.82	7.81	8.63	2.65
	Binary	0.91	1.79	3.1	5.6	8.07	8.12	2.34
3mm	Source	1	2	4	7.95	15.2	26.45	40.58
	Binary	1.03	2.05	4.09	8.16	15.63	27.2	40.22
bicg	Source	1	1.95	3.88	6.7	9.44	5.01	1.62
	Binary	1.04	2.08	4.07	7.3	9.28	7.89	1.15
doitgen	Source	1	2	3.99	7.96	15.66	26.57	24.2
	Binary	1.03	2.05	4.09	8.13	16.2	28.01	21.05
gesummv	Source	1	1.97	3.91	7.56	11.69	10.06	3.26
	Binary	0.99	1.96	3.83	7.49	11.14	10.03	3.63
stream	Source	1	1.95	3.83	7.03	10.77	9.77	2.29
	Binary	0.98	1.96	3.84	7.04	10.89	9.86	1.52
correlation	Source	1	1.34	2.29	4.24	8	14.88	23.62
	Binary	0.9	1.23	2.12	3.94	7.37	13.15	20.98
gemm	Source	1	1.98	3.97	7.93	15.37	24.99	33.9
	Binary	0.99	1.99	3.96	7.9	14.98	26.41	35.58
Geo Mean	Source	1	1.79	3.32	6.01	9.81	12.21	8.64
	Binary	0.99	1.76	3.07	5.32	8.17	10.17	5.95

Figure 7.3: Speedup on SPARC T2 for source and binary using dependence analysis techniques

Table 7.3 shows the geomean of the speedups on a SPARC T2 machine average 8.64X when parallelizing from source code and 5.95X *when parallelizing an x86 binary and using our rewriter to convert it to a SPARC binary.* This cross-ISA-translation is done as described in section 6.2.5.

Further performance-related observations are as follows. On T2 beyond 8 threads the communication costs increase as we need to communicate between different cores. Also for some benchmarks (e.g. *gemver*) we observed that 64 threads run slower than 32 threads. The reasons for this could include the high communication costs between different cores or resource sharing (such as ALUs) between the 8 hardware-supported hyperthreads-like threads per core.

Some benchmarks do not scale as well as others (such as *atax*). The reason for this is that we parallelize the inner loop, and the resulting fine-grained threads for the comparatively small data set are not able to overcome barrier and broadcast latencies for the loop. The *covariance* benchmark parallelizes well from source but poorly from a binary. From source, the compiler can detect that different memory operations in the loop access different portions of the same array (upper triangle and lower triangle of a 2-D array), and hence are independent allowing parallelization. From a binary, the array's linearization prevents such discovery, so the accesses are conservatively deemed dependent. However we expect such cases to be very rare, and also with further linear algebraic techniques we will be able to correctly derive dependence vectors for most of these cases.

We present the statistical count of number of loops present in each benchmark and the number of loops parallelized successfully from source and binary in table

7.4. The total number of loops counted are the outer loops present in benchmarks. Each outer loop may contain several nesting levels. Loops parallelized refers to one nesting level of the loop being parallelized.

Benchmark	Total Number of Loops	Number of Loops parallelized from source	Number of Loops parallelized from binary	Benchmark	Total Number of Loops	Number of Loops parallelized from source	Number of Loops parallelized from binary
2mm	7	7	7	bicg	3	3	3
atax	3	3	3	doitgen	3	3	3
covariance	4	4	3	gesummv	2	2	2
gemver	5	5	5	correlation	5	5	4
jacobi-2d	2	2	2	gemm	4	4	4
3mm	10	10	10	stream	3	3	3

Figure 7.4: Number of loops parallelized from source and binary using dependence analysis techniques

When numbers were collected for this work we used timers at the beginning and end of the benchmarks to measure the time it had taken to execute and then used these numbers to calculate the speedup. This way we had accounted for initialization loops and thread creation times.

7.2 Results for cache reuse metric

In this section we present the results testing the cache reuse metric within the McKinley's decision algorithm within SecondWrite, our binary rewriter.

We use "-O3" optimized binaries from gcc-4.4.1 as input to SecondWrite, which includes our cache reuse model and McKinley's decision algorithm. The output of the decision algorithm is passed on to the parallel code generator which analyzes the dependencies in the loop using analysis presented in chapter 3 and parallelizes the outer most loop dimension. The source automatic parallelizer works on LLVM

IR. Hence, we use LLVM IR generated from "llvm-gcc -O3" as the input to our source parallelizer. During the implementation and testing of this research neither clang nor dragonegg were fully developed and standard, hence we used "llvm-gcc" the then standard.

In this section we present the results of our cache analysis techniques on binaries in a binary automatic parallelizer. First, we present the L1 cache miss rates of the original benchmarks from *Polybench* benchmark suite (a research benchmark suite with dense affine matrix codes) in figure 7.6. We use the *Cachegrind* tool, a part of the *Valgrind* tool to collect our cache numbers. We observe that five benchmarks (*3mm, gemm, doitgen, 2mm and gemver*) have significant L1 cache miss rates, with an average of 27.64% L1 cache misses in the original loops, and hence are candidates for improvement in cache reuse from our method. Our reuse model applies to all binaries, but the remaining benchmarks already have good cache reuse from their favorable loop structure. As a result our method automatically determines that they should not be transformed. In the rest of this section, although we present numbers for all benchmarks, average improvements quoted are for the five benchmarks that had significant miss rates.

We test our system on two machines, the names of which are DASH and BUZZ. We repeat the configurations of both of them in table 7.2. DASH has 4 cores with 2 threads/core, totaling to 8 threads. BUZZ has 4 CPUs with 6 cores/CPU, totalling to 24 threads. When we collected the results for this work we did not have access to T2, which was placed at Univ. of Michigan. Hence, we do not have results on T2 in this section.

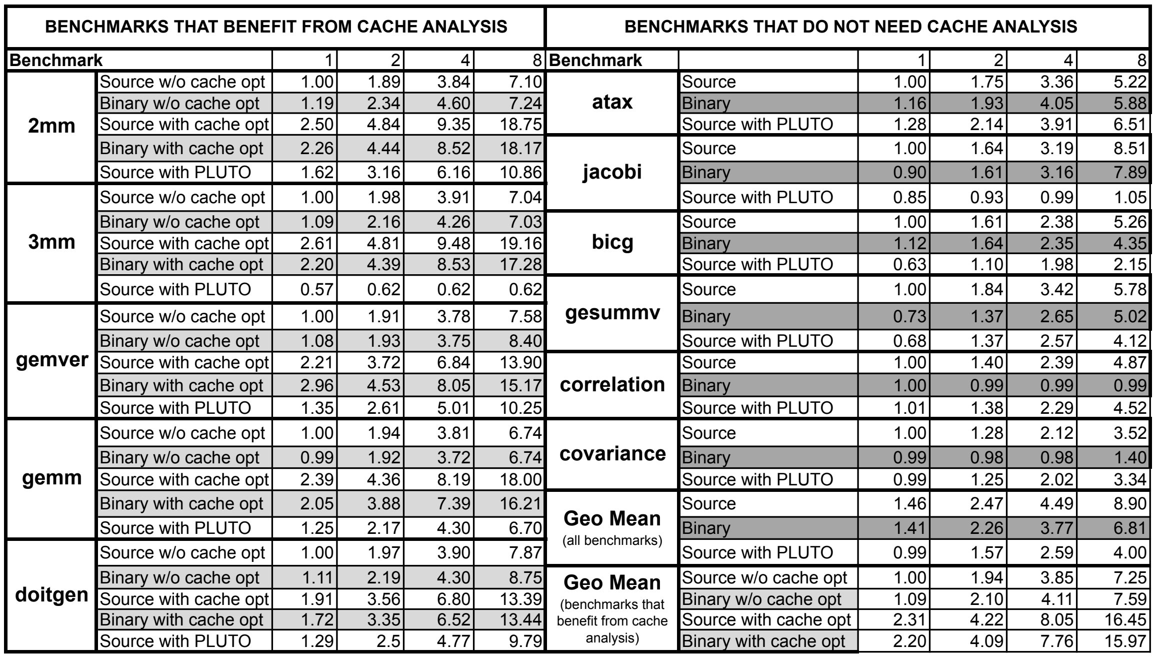
Name	*CPUs*	*Cores/ CPU*	*Threads/ Core*	*Total Threads*	*Manufacturer*	*Model*
DASH	1	4	2	8	Intel	Xeon E5530
BUZZ	4	6	1	24	Intel	Xeon E7450

Table 7.2: Test machines used for testing the cache reuse model

The six goals of our results are: (i) to show that our binary reuse methods can significantly improve performance compared to a basic binary parallelizer; (ii) to show that our binary methods can perform nearly as well as source-level cache reuse methods; (iii) to compare our results with parallelization performed by PLUTO; (iv) to show a detailed analysis of the benefit from McKinley's algorithm, compiler deoptimizations, imperfectly nested techniques and strip-mining; (v) to study the improvement in L1 cache miss rates with our cache optimizations and (vi) to present results showing the benefit of McKinley's algorithm on binaries compiled from three compilers (GCC, ICC and LLVM).

We present our speedup numbers both from source and binary with and without cache analysis through our automatic parallelizer in tables 7.5 and 7.8 for DASH and BUZZ respectively. The benchmarks that benefit from cache analysis are presented on the left side and the average for them is presented at the bottom right corner. The benchmarks that do not need cache analysis are presented on the right.

First, we show that our binary reuse methods can significantly improve performance compared to a basic binary parallelizer. Looking at the binary results (shaded in light gray) of benchmarks that benefit from cache analysis in tables 7.5



BENCHMARKS THAT BENEFIT FROM CACHE ANALYSIS					
Benchmark		1	2	4	8
2mm	Source w/o cache opt	1.00	1.89	3.84	7.10
	Binary w/o cache opt	1.19	2.34	4.60	7.24
	Source with cache opt	2.50	4.84	9.35	18.75
	Binary with cache opt	2.26	4.44	8.52	18.17
	Source with PLUTO	1.62	3.16	6.16	10.86
3mm	Source w/o cache opt	1.00	1.98	3.91	7.04
	Binary w/o cache opt	1.09	2.16	4.26	7.03
	Source with cache opt	2.61	4.81	9.48	19.16
	Binary with cache opt	2.20	4.39	8.53	17.28
	Source with PLUTO	0.57	0.62	0.62	0.62
gemver	Source w/o cache opt	1.00	1.91	3.78	7.58
	Binary w/o cache opt	1.08	1.93	3.75	8.40
	Source with cache opt	2.21	3.72	6.84	13.90
	Binary with cache opt	2.96	4.53	8.05	15.17
	Source with PLUTO	1.35	2.61	5.01	10.25
gemm	Source w/o cache opt	1.00	1.94	3.81	6.74
	Binary w/o cache opt	0.99	1.92	3.72	6.74
	Source with cache opt	2.39	4.36	8.19	18.00
	Binary with cache opt	2.05	3.88	7.39	16.21
	Source with PLUTO	1.25	2.17	4.30	6.70
doitgen	Source w/o cache opt	1.00	1.97	3.90	7.87
	Binary w/o cache opt	1.11	2.19	4.30	8.75
	Source with cache opt	1.91	3.56	6.80	13.39
	Binary with cache opt	1.72	3.35	6.52	13.44
	Source with PLUTO	1.29	2.5	4.77	9.79

BENCHMARKS THAT DO NOT NEED CACHE ANALYSIS					
Benchmark		1	2	4	8
atax	Source	1.00	1.75	3.36	5.22
	Binary	1.16	1.93	4.05	5.88
	Source with PLUTO	1.28	2.14	3.91	6.51
jacobi	Source	1.00	1.64	3.19	8.51
	Binary	0.90	1.61	3.16	7.89
	Source with PLUTO	0.85	0.93	0.99	1.05
bicg	Source	1.00	1.61	2.38	5.26
	Binary	1.12	1.64	2.35	4.35
	Source with PLUTO	0.63	1.10	1.98	2.15
gesummv	Source	1.00	1.84	3.42	5.78
	Binary	0.73	1.37	2.65	5.02
	Source with PLUTO	0.68	1.37	2.57	4.12
correlation	Source	1.00	1.40	2.39	4.87
	Binary	1.00	0.99	0.99	0.99
	Source with PLUTO	1.01	1.38	2.29	4.52
covariance	Source	1.00	1.28	2.12	3.52
	Binary	0.99	0.98	0.98	1.40
	Source with PLUTO	0.99	1.25	2.02	3.34
Geo Mean (all benchmarks)	Source	1.46	2.47	4.49	8.90
	Binary	1.41	2.26	3.77	6.81
	Source with PLUTO	0.99	1.57	2.59	4.00
Geo Mean (benchmarks that benefit from cache analysis)	Source w/o cache opt	1.00	1.94	3.85	7.25
	Binary w/o cache opt	1.09	2.10	4.11	7.59
	Source with cache opt	2.31	4.22	8.05	16.45
	Binary with cache opt	2.20	4.09	7.76	15.97

Figure 7.5: Speedup on DASH using the cache resuse metric for 8 threads

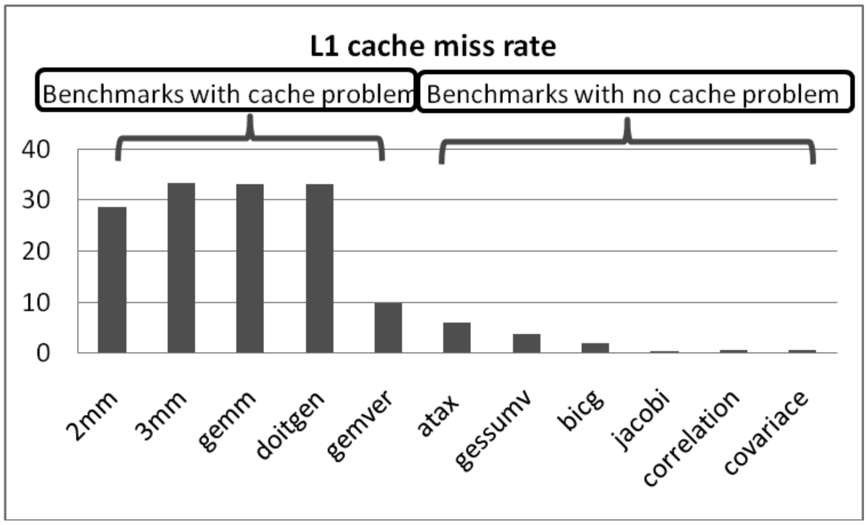


Figure 7.6: Cache miss rate for the original benchmarks

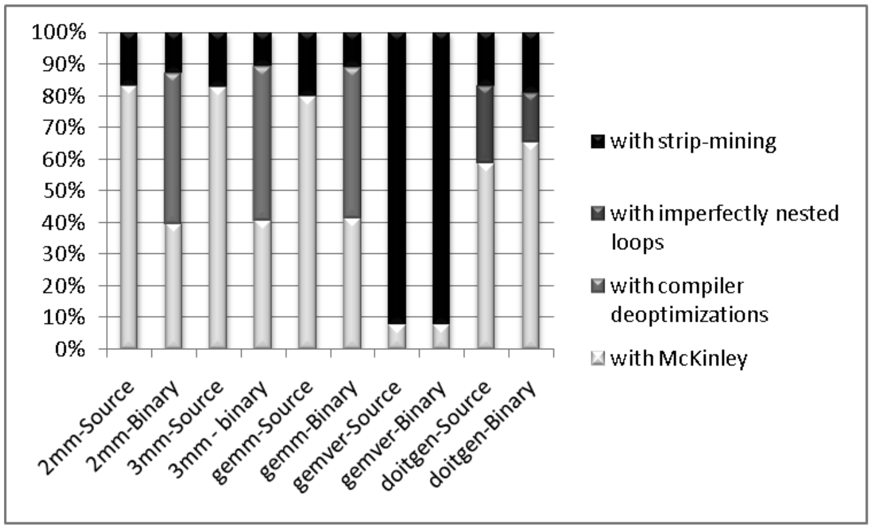


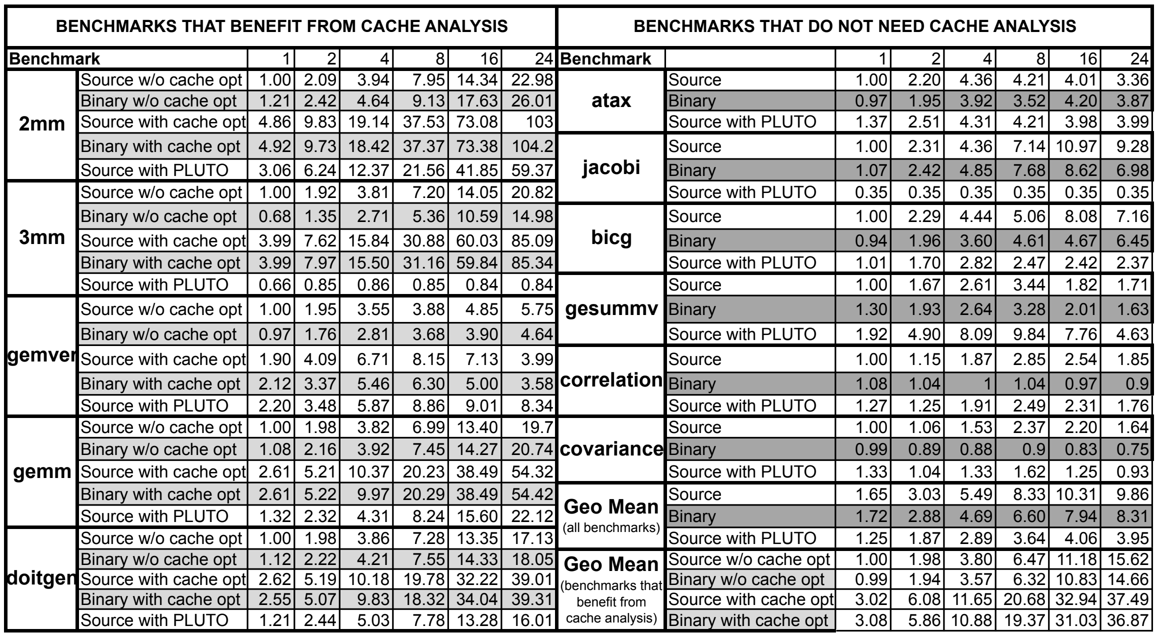
Figure 7.7: Breakup of the speedup on x86 DASH from different steps of the decision algorithm

and 7.8, we observe that: (i) the geomean speedup with cache analysis was 15.97X for 8 threads on DASH whereas the geomean speedup without cache analysis was 7.59X (2.1X better); (ii) the geomean speedup with cache analysis was 36.87X for 24 threads on BUZZ whereas the geomean speedup without cache analysis was 14.66X (2.52X better); (iii) even for one thread on DASH and BUZZ the speedups with cache analysis are 2.2X and 3.08X respectively. From these observations we conclude that cache analysis on binaries with significant cache miss rates improves their performance by factors of 2.1X-3.08X.

Second, we show that our binary methods can perform nearly as well as source-level cache reuse methods. Comparing the source and binary speedup numbers in tables 7.5 and 7.8, we observe that the speedups from binary are nearly identical to those from source code with equivalent optimization.

Third, we compare the results of our parallelizing compiler with PLUTO. The geomean speedup of our benchmarks with PLUTO is 4X on DASH and 3.95X on BUZZ whereas the speedup from our automatic parallelizer with cache analysis is 6.81X on DASH and 8.31X on BUZZ. The reason for better performance by us is that PLUTO is optimizing for maximizing parallelism and our benchmarks need to be improved for cache in addition. The power of the polyhedral model is in code transformation for parallelism. Generally speaking, cache optimization is a secondary concern in the polyhedral model, often retrofitted using heuristics based on a cache model. Hence the polyhedral model too can use our cache model from binaries. We do not suggest that PLUTO or the polyhedral model is less powerful, we merely wish to present results from PLUTO to show that our speedups are comparable to the best publicly available academic polyhedral compiler.

Fourth, figure 7.7 shows the breakup of run-time improvement between McKinley's algorithm, compiler deoptimizations, our handling of imperfectly nested loops and strip-mining. We observe that: (i) source benchmarks do not benefit from compiler deoptimizations, since our benchmarks were perfectly nested from source; (ii) binaries (such as *2mm*, *3mm* and *gemm*) benefit from compiler deoptimizations, since it acts as an enabler for McKinley's algorithm; (iii) *gemver* benefits from strip mining both from source and binary since it exposes coarse-grain parallelism against inner-loop parallelization; (iv) *doitgen* benefits from imperfectly nested techniques since McKinley's algorithm is otherwise not fully applicable to it; (v) the gain from strip-mining on our platforms for our benchmarks is limited mainly to exposing coarse-grain parallelism such as in *gemver*; all other benchmarks gain only minimally from strip-mining.



BENCHMARKS THAT BENEFIT FROM CACHE ANALYSIS							
Benchmark		1	2	4	8	16	24
2mm	Source w/o cache opt	1.00	2.09	3.94	7.95	14.34	22.98
	Binary w/o cache opt	1.21	2.42	4.64	9.13	17.63	26.01
	Source with cache opt	4.86	9.83	19.14	37.53	73.08	103
	Binary with cache opt	4.92	9.73	18.42	37.37	73.38	104.2
	Source with PLUTO	3.06	6.24	12.37	21.56	41.85	59.37
3mm	Source w/o cache opt	1.00	1.92	3.81	7.20	14.05	20.82
	Binary w/o cache opt	0.68	1.35	2.71	5.36	10.59	14.98
	Source with cache opt	3.99	7.62	15.84	30.88	60.03	85.09
	Binary with cache opt	3.99	7.97	15.50	31.16	59.84	85.34
	Source with PLUTO	0.66	0.85	0.86	0.85	0.84	0.84
gemver	Source w/o cache opt	1.00	1.95	3.55	3.88	4.85	5.75
	Binary w/o cache opt	0.97	1.76	2.81	3.68	3.90	4.64
	Source with cache opt	1.90	4.09	6.71	8.15	7.13	3.99
	Binary with cache opt	2.12	3.37	5.46	6.30	5.00	3.58
	Source with PLUTO	2.20	3.48	5.87	8.86	9.01	8.34
gemm	Source w/o cache opt	1.00	1.98	3.82	6.99	13.40	19.7
	Binary w/o cache opt	1.08	2.16	3.92	7.45	14.27	20.74
	Source with cache opt	2.61	5.21	10.37	20.23	38.49	54.32
	Binary with cache opt	2.61	5.22	9.97	20.29	38.49	54.42
	Source with PLUTO	1.32	2.32	4.31	8.24	15.60	22.12
doitgen	Source w/o cache opt	1.00	1.98	3.86	7.28	13.35	17.13
	Binary w/o cache opt	1.12	2.22	4.21	7.55	14.33	18.05
	Source with cache opt	2.62	5.19	10.18	19.78	32.22	39.01
	Binary with cache opt	2.55	5.07	9.83	18.32	34.04	39.31
	Source with PLUTO	1.21	2.44	5.03	7.78	13.28	16.01

BENCHMARKS THAT DO NOT NEED CACHE ANALYSIS							
Benchmark		1	2	4	8	16	24
atax	Source	1.00	2.20	4.36	4.21	4.01	3.36
	Binary	0.97	1.95	3.92	3.52	4.20	3.87
	Source with PLUTO	1.37	2.51	4.31	4.21	3.98	3.99
jacobi	Source	1.00	2.31	4.36	7.14	10.97	9.28
	Binary	1.07	2.42	4.85	7.68	8.62	6.98
	Source with PLUTO	0.35	0.35	0.35	0.35	0.35	0.35
bicg	Source	1.00	2.29	4.44	5.06	8.08	7.16
	Binary	0.94	1.96	3.60	4.61	4.67	6.45
	Source with PLUTO	1.01	1.70	2.82	2.47	2.42	2.37
gesummv	Source	1.00	1.67	2.61	3.44	1.82	1.71
	Binary	1.30	1.93	2.64	3.28	2.01	1.63
	Source with PLUTO	1.92	4.90	8.09	9.84	7.76	4.63
correlation	Source	1.00	1.15	1.87	2.85	2.54	1.85
	Binary	1.08	1.04	1	1.04	0.97	0.9
	Source with PLUTO	1.27	1.25	1.91	2.49	2.31	1.76
covariance	Source	1.00	1.06	1.53	2.37	2.20	1.64
	Binary	0.99	0.89	0.88	0.9	0.83	0.75
	Source with PLUTO	1.33	1.04	1.33	1.62	1.25	0.93
Geo Mean (all benchmarks)	Source	1.65	3.03	5.49	8.33	10.31	9.86
	Binary	1.72	2.88	4.69	6.60	7.94	8.31
	Source with PLUTO	1.25	1.87	2.89	3.64	4.06	3.95
Geo Mean (benchmarks that benefit from cache analysis)	Source w/o cache opt	1.00	1.98	3.80	6.47	11.18	15.62
	Binary w/o cache opt	0.99	1.94	3.57	6.32	10.83	14.66
	Source with cache opt	3.02	6.08	11.65	20.68	32.94	37.49
	Binary with cache opt	3.08	5.86	10.88	19.37	31.03	36.87

Figure 7.8: Speedup on BUZZ using cache reuse metric for 24 threads

Fifth, we present a study of the improvement in L1 cache miss rates with our cache optimizations in table 7.3. We observe that the L1 cache misses are significantly reduced after our techniques are applied to candidate loops.

We also present a detailed account of the change in loop structure for the five benchmarks in table 7.4. `i`, `j`, `k`, `s`, `p` etc represent the original loop, `i1`, `s1`, `p1` etc. represent the outer strip-mined loop and `i2`, `j2`, `s2` etc. represent the inner strip-mined loop. In gemver strip-mining is performed to achieve coarse grain parallelism. In doitgen we use non-perfectly nested techniques to achieve better cache locality.

Last, we want to show that our binary automatic parallelizer works efficiently on binaries compiled from different compilers for x86 code. For this, we compiled the benchmarks from polybench using three different compilers, GCC, ICC and LLVM all targeting the x86 ISA. The binaries were compiled using the highest level

Benchmark	*Baseline Source*	*+ cache opt*	*Baseline Binary*	*+ cache opt*
2mm	28.6%	3.2%	33.3%	2.1%
3mm	33.3%	3.2%	33.3%	2.1%
gemver	9.9%	2.2%	10.8%	2.7%
gemm	33.2%	4.6%	24.9%	3%
doitgen	33.2%	3%	24.9%	2%
Average	27.64%	3.24%	25.44%	2.38%

Table 7.3: L1 cache miss rate after applying modified McKinley algorithm

Benchmark	Input loop structure	Output loop structure
gemver	(i, j)	(i1, j, i2)
doitgen	(r, q, p, s)	(r, q, s1, p1, s2, p2)
2mm, 3mm, gemm	(i, j, k)	(i1, k1, j1, i2, k2, j2)

Table 7.4: Loop nest transformation from input to output using modified McKinley's algorithm

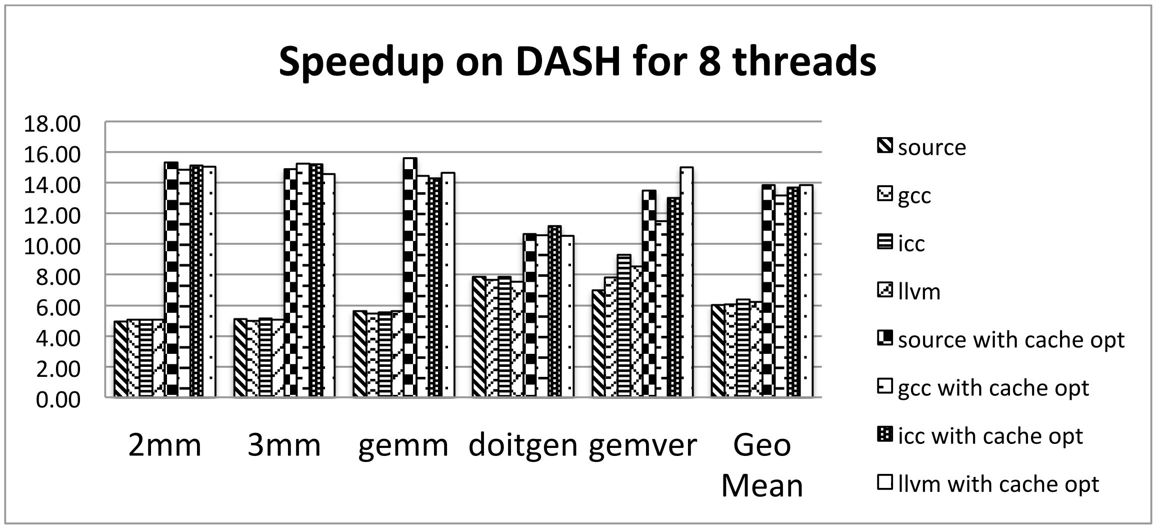


Figure 7.9: Speedup on x86 DASH for benchmarks compiled from GCC, ICC and LLVM

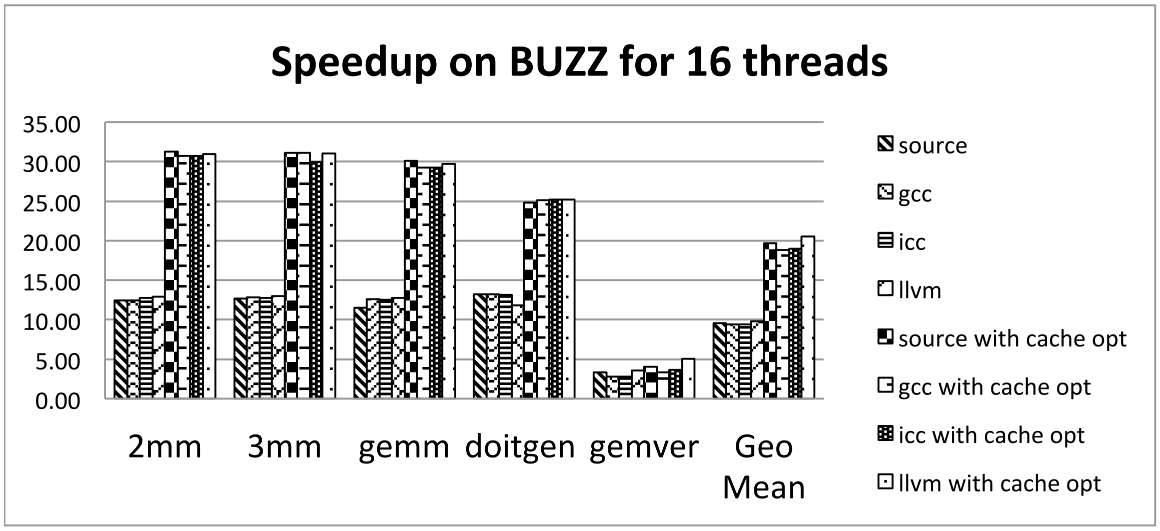


Figure 7.10: Speedup on x86 BUZZ for benchmarks compilers from GCC, ICC and LLVM

of optimization in these compilers without using vectorization. We then use these binaries as input to *SecondWrite* with the affine automatic parallelizer containing the cache reuse model within the McKinley's decision algorithm. We present results in figures 7.9 and 7.10 for the benchmarks that have high cache miss rate on both the test machines DASH and BUZZ for 8 threads and 16 threads respectively. The first four bars for each of the benchmarks in each of the graphs presents speedups from source, GCC compiled binary, ICC compiled binary and LLVM compiled binary with just the basic parallelizer. The next four bars for each of the benchmarks show the speedup from source, GCC compiled binary, ICC compiled binary and LLVM compiled binary with the parallelizer including modified McKinley's algorithm and strip-mining as well. We observe that (i) irrespective of the compiler that was used to compile the binary each one was parallelized effectively using our dependence analysis techniques. The geomean of the speedups on DASH for 8 threads from source was 6.01X, from GCC compiled binaries was 6.06X, from ICC compiled binaries was 6.38X and LLVM compiled binaries was 6.21X. Similarly, on an average the geomean of the speedups on BUZZ for 16 threads from source was 9.58X, from GCC compiled binaries was 9.4X, from ICC compiled binaries was 9.42X, from LLVM compiled binaries was 9.78X. This shows that we are able to effectively parallelize binaries compiled from any compiler. It is good to note that the LLVM IR generated from each of the compiled binary code is very different from the other and we are correctly able to use induction variable analysis to recover the linearized multi-dimension equations from each of them and then apply linear algebraic techniques to them and recover similar distance vectors from them and parallelize them in the same way. The reason I say similar and not same is that sometimes certain scalar dependences from one binary compiled using one compiler for the same benchmark manifests itself as a memory dependence from a binary from a different compiler; (ii) irrespective of the compiler each of the benchmarks gained from the McKinley algorithm. This again proves that our McKinley formulation is compiler independent and we are able to correctly take loop order decisions irrespective of the compiler used to compile the input binary. Stating this in numbers the geomean of the speedups on DASH for 8 threads using McKinley's decision algorithm from source was 13.86X (2.31X faster than basic), from GCC compiled binary was 13.18X (2.17X faster than base), from ICC binary was 13.67X (2.14X faster than basic), from LLVM binary was 13.84X (2.23X faster than basic). Similarly, the geomean of the speedup on BUZZ for 16 threads from source was 19.66X (2.05X faster than basic), from GCC compiled binary was 18.82X (2X faster than basic), from ICC compiled binary was 18.99X (2.02X faster than basic), from LLVM compiled binary was 20.54X (2.1X faster than basic).

7.3 Results for algorithm for guessing loop bounds

In this section we present results for our algorithm to guess loop bounds when the loop bounds are not statically known and then insert run-time checks to check that these guesses are actually correct before running the parallel version of the loop. The benchmarks in this section are from the SPEC2006 and OMP2001 benchmark suite as against the affine kernels from the polybench benchmark suite since for all the benchmarks in polybench the loop bounds are statically known. By testing our affine parallelizer on the larger benchmarks we show that our parallelizer is scalable. We want to note at this stage that we have not found any polyhedral papers that present results on the SPEC or OMP benchmark suites. Further, we downloaded PLUTO the research polyhedral compiler and tried running lbm (one of the smaller benchmarks from SPEC2006) through it and found that it crashes. Hence, in this section we have not been able to test our results against PLUTO as we did in the previous section.

We use "-O3" optimized binaries from gcc-4.3 and gfortran-4.3 as input to *SecondWrite*, which includes the new algorithm proposed in chapter 4 to guess loop bounds when they are statically unknown. The affine automatic parallelizer from source works on LLVM IR. Hence, we use LLVM IR generated from *clang* [44] (a C language front-end for LLVM) for the 'C' benchmarks and LLVM IR generated using the *dragonegg* [45] plugin (a plugin that integrates the LLVM optimizers and code generator with GCC) for the Fortran benchmarks as the input to our source parallelizer. We run all the binaries on the AMD Opteron(TM) processor 6212 and present results. The reason we chose to run it on this AMD processor as against the earlier DASH and BUZZ machines is that this machine was a new addition to our lab and was more powerful than either of the DASH and BUZZ machines. It was also convenient to run *SecondWrite* and the binaries on the same machine as against transferring binaries between machines, as both DASH and BUZZ are remotely located.

In this section we present our results on parallelizing binaries from SPEC2006 and OMP2001 using our new algorithm to guess loop bounds. First, we introduce our benchmarks. Second, we present the speedups we have from source and binary. For the binary numbers, we present results for speedups both with and without the new algorithm. Third, we present the actual number of affine loops that are parallelized from the binary with and without the algorithm. We measure speedups by measuring the clock time to run the programs on 1 thread and 8 threads. Fourth, we introduce the non-affine benchmarks in the SPEC2006 and OMP2001 benchmark suites. Fifth, we present the speedup of these benchmarks using our source and binary automatic parallelizers and also present the number of loops that were parallelized in these benchmarks.

Benchmark	*Language*	*Lines of code*	*Suite*
swim	Fortran	275	OMP2001
bwaves	Fortran	680	SPEC2006
mgrid	Fortran	789	OMP2001
lbm	C	908	SPEC2006
quake	C	1151	OMP2001
libquantum	C	2605	SPEC2006
milc	C	9575	SPEC2006
cactus	Fortran + C	59827	SPEC2006

Table 7.5: Description of the affine benchmarks from the SPEC 2006 and OMP 2001 benchmark suites

First, table 7.5 lists the 8 affine benchmarks that we present our results on.

The remaining benchmarks are not affine rich and the results for those are presented later in the section. We have first picked only the affine rich benchmarks from the SPEC2006 and OMP2001 benchmark suites. We manually profiled every benchmark belonging to both the benchmark suites and after examining the hot regions classified benchmarks as affine or not affine. We present our results on all the affine benchmarks first discovered from both the benchmark suites. The benchmarks *swim*, *mgrid* and *quake* belong to the OMP2001 benchmark suite and *bwaves*, *lbm*, *libquantum*, *milc* and *cactus* belong to the SPEC2006 benchmark suite. These benchmarks range from 275 to 59,827 lines of code as shown in table 7.5.

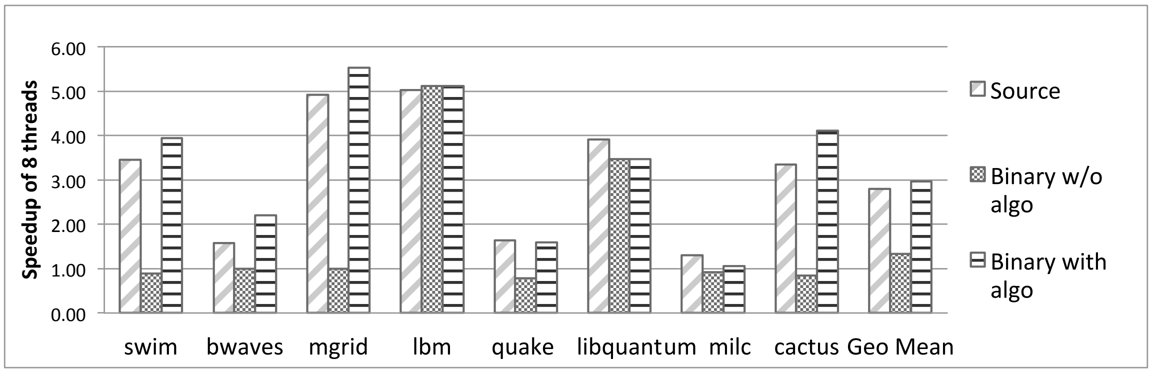


Figure 7.11: Speedup of the affine benchmarks from SPEC 2006 and OMP 2001 benchmark suites for 8 threads

Second, figure 7.11 presents the speedup for 8 threads from source and binary for each of the benchmarks w.r.t the GCC "-O3" compiled single thread version of the benchmark. There are three bars for each benchmark; (i) the first bar is the speedup of the benchmark from source code for 8 threads; (ii) the second bar is the speedup of the binary for 8 threads without the new algorithm using only the theory presented in chapter 3 and (iii) the third bar is the speedup of the binary for 8 threads using the new algorithm presented in the chapter 4. We observe that *swim*, *bwaves*, *mgrid*, *quake*, *milc* and *cactus* gain significant speedups when the new algorithm from chapter 4 is present in the static affine binary parallelizer. The significant affine loops in these benchmarks have run-time determined loop bounds and hence using our new algorithm we are able to parallelize these loops that were not parallelized using the theory developed before. The benchmarks *lbm* and *libquantum* do not have any difference in the speedups with and without the algorithm. The reason being; (i) in *lbm*, the loops bounds are statically known and hence the theory from chapter 3 is sufficient to parallelize the affine loops in it and (ii) in *libquantum* the loops are single dimensional with a write to one single dimensional memory accesses. These loops can be parallelized without the new algorithm and hence we see a speedup in *libquantum* even without the new algorithm. Overall the geomean speedup for 8 threads for the 8 benchmarks from binaries increases from 1.33X to 2.96X with the addition of the new algorithm. Our binaries run slightly faster than source since SecondWrite is able to rewrite "-O3" binaries to run 10% faster than the input binaries.

Third, table 7.6 presents the number of loops that are parallelized from the binary with and without the new algorithm. We observe that in the benchmarks *lbm* and *libquantum* the number of loops parallelized with and without the algorithm do not change. The reasons for this have been explained earlier. In *swim*, *quake*, *milc* and *cactus*, a number of loops are parallelized even when the new algorithm is not present in the static affine binary parallelizer; however, these loops are small and do not contribute to the run-time of the benchmark. Hence, these loops do not result

Benchmark	No. of loops w/o algo	No. of loops with algo
swim	6	18
bwaves	0	1
mgrid	0	6
lbm	4	4
quake	7	9
libquantum	18	18
milc	37	43
cactus	112	126

Table 7.6: Number of loops parallelized for the affine benchmarks with and without the run time loops algorithm.

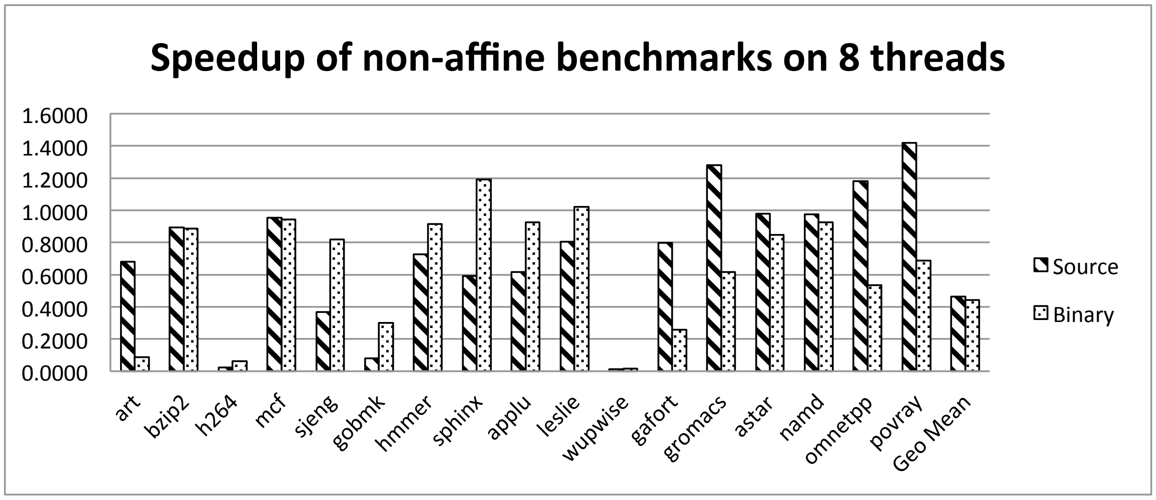


Figure 7.12: Speedup of the non-affine benchmarks from SPEC 2006 and OMP 2001 benchmark suites

in a speedup from 8 threads for these benchmarks. We make this comparison to show that it is not the number of loops that are parallelized that matter, but it is important to parallelize the run-time intensive loops that can be parallelized by our new algorithm.

Fourth, we would like to introduce the non-affine benchmarks from SPEC2006 and OMP2001 that we present results on in table 7.7. The non-affine benchmarks include *art*, *bzip2*, *h264*, *mcf*, *sjeng*, *gobmk*, *hmmer* and *sphinx* written in C. The non-affine Fortran benchmarks include *applu*, *leslie*, *wupwise* and *gafort*. *Gromacs* is a benchmark that is written in parts in C and in parts in Fortran. The non-affine C++ benchmarks include *astar*, *namd*, *omnetpp* and *povray*. All the benchmarks total to 629,908 lines of code.

Finally, we present the speedup from parallelizing these benchmarks using the source and binary automatic parallelizer containing the new algorithm to guess loop

Benchmark	*Language*	*Lines of code*	*Suite*
art	C	1914	OMP2001
bzip2	C	8293	SPEC2006
h264	C	51578	SPEC2006
mcf	C	2685	SPEC2006
sjeng	C	13847	SPEC2006
gobmk	C	197215	SPEC2006
hmmer	C	35992	SPEC2006
sphinx	C	23373	SPEC2006
applu	F	3808	OMP2001
leslie	F	3807	SPEC2006
wupwise	F	2468	OMP2001
gafort	F	1344	OMP2001
gromacs	C/F	105992	SPEC2006
astar	C++	5842	SPEC2006
namd	C++	3188	SPEC2006
omnetpp	C++	15313	SPEC2006
povray	C++	155163	SPEC2006
Total lines of code		629908	

Table 7.7: Description of the non-affine benchmarks from the SPEC 2006 and OMP 2001 benchmark suites

bounds. These results are presented in figure 7.12. The first bar presents the speedup from source and the second bar presents the speedup from binary. We observe that the average geomean speedup from source is 0.46X for 8 threads and the average speedup from binary is 0.44X. In most of these benchmarks we parallelize only small loops as all these benchmarks are not affine in the hot regions. Since we parallelize small loops there is a slowdown in the program as against a speedup. Using a better decision algorithm, which is able to take run-time decisions for running the serial version of the loop as against the parallel one for small short running loops, can avoid this slowdown. We leave this to future work since the static decision algorithm we use is good for the affine benchmarks. Next in table 7.8 we present the number of loops that were parallelized in each of the benchmarks using the binary automatic parallelizer including the algorithm to guess loop bounds for statically unknown loop bounds. The reason we present just the number of loops using the algorithm is that in these benchmarks all the loops that are parallelized are small loops mostly single dimension hence the number of loops parallelized with and without the algorithm is not different.

Benchmark	No. of loops parallelized
art	31
bzip2	18
h264	112
mcf	2
sjeng	1
gobmk	58
hmmer	74
sphinx	57
applu	5
leslie	37
wupwise	9
gafort	22
gromacs	93
astar	9
namd	11
omnetpp	9
povray	89

Table 7.8: Number of loops parallelized by our binary parallelizer for the non-affine benchmarks in SPEC 2006 and OMP 2001 benchmark suites

Chapter 8

Related Work

Supporting related work is listed throughout the thesis as appropriate. Further, in chapter 2 we have presented in detail the previous work on source affine parallelization. This chapter is divided into the following sections each presenting one potential aspect of related work: (i) Static binary rewriters and their applications; (ii) Affine based automatic parallelizers from source; (iii) Methods to calculate distance and direction vectors; (iv) Dynamic binary automatic parallelization methods; (v) Static binary automatic affine parallelizers; (vi) Automatic vectorization of binaries; (vii) Array delinearization techniques; (viii) Cache optimizations using binary rewriting and (ix) Cache analysis and optimizations in source affine parallelizers.

8.1 Static binary rewriters and their applications

In this section we briefly present the static binary rewriters that have been built and why we choose *SecondWrite* among others. Existing binary and object-code rewriters include Etch [48], squeeze and squeeze++ [49, 50], PLTO [40], DIABLO [41], ALTO [51], ATOM [39], spike [52, 53] and *SecondWrite* [54] [43] [37] [35]. Most of these binary rewriters except *SecondWrite* use low level IR to represent the program and are mostly link-time optimizers for instrumentation and code compaction.

DIABLO [41] is a link time optimizer that has access to relocation information. The advantage Diablo has over traditional compilers is that it has a whole program view because it operates at link time. It is able to optimize the program for performance by reducing code size. ATOM [39] provides a infrastructure in which an user can define different instrumentation tools. At link-time, ATOM instruments the program correctly without interfering with the program itself with low overhead. Squeeze and squeeze++ [49, 50] are link-time tools essentially for embedded software for code compaction since code size can be a huge overhead on embedded machines. They demonstrate that using just the information present at link-time, one can perform whole program analysis with an end goal of code compaction. PLTO [40] is a link-time optimization tool for the IA-32 architecture to improve performance of binaries. ALTO [51] is a link-time optimizer that is even able to optimize library calls to specific calling contexts; this way it is further able to efficiently optimize even highly optimized binaries. All the above tools are link-time optimizers and we wanted to implement our affine parallelizer on binary code without relocation information. Hence, we did not choose any of the above binary rewriters for implementing our affine parallelizer. Spike [52, 53] is a profile-guided optimizer. Its framework is able to efficiently manage the profile information of binaries and then optimize programs predominantly consisting of loops spanning multiple functions with complex control flow. We did not implement our affine parallelizer within spike since we wanted a system that is able to represent the code as IR which spike does not do. Etch [48] is a general purpose tool to give an user a platform to write their own transformation for binaries for both measurement and optimization. It can however not be used for a program transformation as complex as automatic parallelization. On the other hand *SecondWrite* [54] [43] [37] [35] is a static binary rewriter that is able to represent any binary code as intermediate compiler IR and also recover high level information such as function arguments, stack variables, type information etc. Such information can highly aid the implementation of complex code transformations such as automatic parallelization. We have described the details of *SecondWrite* and how it interacts with our affine automatic parallelizer in chapter 6.

8.2 Affine based automatic parallelizers from source

We have dedicated the entire chapter 2 to describe in detail different affine automatic parallelization techniques from source. They are presented here to complete this section. As acknowledged throughout this paper, our method builds on existing methods, but has significant differences allowing it to work on binaries for the first time.

Affine automatic parallelizers built on traditional techniques from compilers include Polaris [55], SUIF [56, 57, 23], and pHPF [58], PROMIS [59], Parafrase-2 [60] and McKinley's algorithm [12]. All these automatic parallelizing compilers parallelize code from source, unlike our method built for binary code. These methods also use distance vectors to characterize dependences in affine loops like we do. We build on these methods and adapt them for binaries.

The other school of parallelizing affine programs is using the polyhedral model like in [24] [25] [26] [27] [28] [29] [30]. These methods represent every dynamic instance of the program in a polyhedron space. Scheduling functions are used to transform loops. Details have been presented in chapter 2.

8.3 Linear algebriac methods to calculate distance and direction vectors

Affine loop parallelism has required solving systems of linear Diophantine equations [13] to calculate distance vectors. Various techniques have been proposed in literature to solve these equations. These include the Greatest Common Divisor (GCD) test [14, 15], Banerjee's inequalities [15], Single Index and Multiple Index Tests [13, 7], Multidimensional GCD [15], the delta test [16] and the omega test [17].

The GCD [14, 15] test concludes that there is a dependence between statements X[ai+b] and X[ci+d] if GCD(c,a) divides (d-b). It has been used traditionally along with Banerjee's inequalities. Both these are approximate methods. Multi-dimensional GCD [15] looks for an integer solution for multi-dimensional arrays by representing them as a system of Diophantine equations. Delta test [16] presents an efficient method of determining dependence in most common cases of affine indices by classifying pairs of array indices. They show that these array indices dominate scientific codes. Omega test [17] formulates the problem of finding dependence vectors as an integer-programming model and shows that it is fast enough to be used in production quality compilers. Knowledge before the omega test believed that integer programming was very expensive.

We adopt these tests from the source to our binary automatic parallelizer. We have presently implemented the Greatest Common Denominator (GCD), Single and Multiple Index tests and Delta test to solve the linear Diophantine equations that we recover directly from a binary. All the other tests also can be implemented in our system. We have not implemented them because our benchmarks did not need them.

8.4 Dynamic binary automatic parallelization methods

The existing dynamic binary automatic parallelization techniques are limited to non-affine techniques and do not perform sophisticated affine analysis like we do.

Yardimci and Franz [61] present a method to dynamically compile a sequential binary to a parallelized or vectorized code. Their techniques are mostly complementary to ours; in that instead of affine parallelism, their techniques include control speculation, loop distribution and automatic parallelization of recursive techniques. They parallelize loops that do not have loop carried dependencies, which limits the scope of loops parallelized drastically. As, we are able to perform sophisticated affine analysis on memory strides present in loops; the scope of loops parallelized by us is higher. Further, their techniques are dynamic preventing them from integrating sophisticated decision algorithm into their system.

Hertzberg et.al. [62] presents a method to extract speculative threads from a running executable. The system is called RASP which uses the idle cores of a multi-processor to analyze the running executable and then execute it in parallel.

Wang et.al. [63] presents a dynamic method to parallelize binaries using speculative slicing to extract both instruction level parallelism (ILP) and thread level parallelism (TLP). These techniques are primarily targeted towards irregular applications.

Yang et.al. [64] presents a system in which a running benchmark is monitored at run-time and the hot regions are parallelized and cached so that the parallel version can be executed when the hot region is repeated. Their techniques can be used on any third party binary including library codes. However, they do not present any sophisticated affine analysis like we do.

All the four methods are dynamic. Hence, they suffer from run-time overheads from analysis. Our method being static does not suffer from any run-time overhead. Most importantly, they do not optimize affine loops using sophisticated affine analysis and loop transformation whereas our method does.

8.5 Static affine automatic parallelization of binaries

We were the first ever to publish work in this area at [35]. This work presents the first main contribution in this thesis, *i.e.*how every memory access in a affine loop can be represented as a linearized multi-dimensional expression and then distance/direction vectors calculated for the same. Our second publication on guessing loop bounds for loops whose bounds are run-time dependent is under review at [65]. The third contribution on calculating the cache reuse metric directly from binaries and using it to decide loop nest transformation is under review at [66].

The only other work we are aware of that parallelizes affine binaries using static analysis of binary code is by Pradelle et.al [46] [47]. This work leverages the source to source PLUTO polyhedral compiler by transforming some parts of the affine kernels from binaries in polybench to C. Further, they have had to use many tricks on the recovered C code to feed it to the polyhedral model such as de-linearization of array accesses and scalar variable removal. In general we believe that their approach is not scalable to large benchmarks where as our methods are.

8.6 Automatic vectorization of binaries

Nakamura et.al. [67] and Dasgupta et.al. [68] present techniques to analyze binaries and vectorize them. [67] presents a technique to do a linear search on binary code at run-time and then generate SIMD instructions for the hot regions when data parallelism exists. SIMD instructions can help increase the performance of binary code since they can perform multiple regular instructions in one instruction. [68] presents the Vizer framework that analyzes binary code to identify instructions that can be replaced by SIMD instructions to increase performance. Both these methods are limited to vectorization of binaries and do not attempt to parallelize them using threads as we do.

8.7 Array delinearization techniques

Array delinearization methods [69] [70] take source code with linearized multi-dimensional accesses as input, and convert those accesses to multi-dimensional accesses when possible. Ideally if we delinearize array accesses in a binary we could take parallelizing decisions on them just as we would from source. However, source-level methods to delinearize array accesses cannot be adapted to binaries easily. This is because delinearization methods such as [69], [70] require high level intermediate C like representation equivalent to symbolic information in compilers which is not available when analyzing binary code. Such symbolic information is crucial because it contains information about the number, location, and dimension sizes of arrays, which the delinearization methods use. Finding this information in the general case from stripped binaries (*i.e.*those without symbolic information) is hard, and there are no existing methods for it. Hence delinearization methods cannot be adapted for binary code.

The method in the chapter 4 circumvents the problem of missing array information in binaries by not attempting to recover guaranteed information about array locations and dimension sizes. Instead it guesses the bounds of loops. When the guesses are correct, the code can be parallelized. Run-time checks ensure that when the guessed bounds are wrong, fallback serial code ensures correct execution. No previous method guesses loop bounds from binaries, or uses run-time checks like our method. The result is that our method is the first to parallelize binary code with unknown loop bounds.

8.8 Cache optimizations using binary rewriting

There has been some prior work in studying cache behavior directly from binaries and applying optimizations to decrease their runtime. However, these transformations are limited and do not include affine loop transformations such as interchange. Nethercote et.al [71] does dynamic binary instrumentation and uses hardware counters to measure the miss rate of caches. Using cache miss rate information, they insert prefetch instructions to fetch the memory into cache few instructions before their actual use into binaries and this improves the cache performance of binaries since the memory location is in the cache when it is used and there will be no fetch overhead. Weidendorfer et.al [72] builds on [71] and uses the dynamic cache profile information to perform optimizations such as array padding and blocking. Our method is different from these methods in two ways: (i) our method uses affine methods to analyze and transform binary whereas [71] [72] do not; They use hardware counters instead our cache reuse metric calculation method and (ii) the methods in [71] [72] are dynamic and incur run-time overhead from instrumentation, analysis and transformation, whereas our method, which is static, has no such overheads.

We do not know of any prior work that uses static techniques to estimate cache performance and use that information to improve the binaries. We claim novelty for the static techniques used to calculate the *cache reuse metric.*

8.9 Cache analysis and optimizations in source affine parallelizers

Many source affine parallelizers such as [34] [12] [33] use a *cache reuse metric* to take decisions about loop nest transformation. Wolf et.al. [33] presents a method to do an ad-hoc search of loop transformations and calculates the *cache reuse metric* for each sequence of loop transformations and selects the best transformation order. We believe that this model is lacking since an exhaustive search is expensive. Bondhugula et.al. [34] uses the *cache reuse metric* in the polyhedral framework to reason about loop transformations. We have stated our reasons for not using the polyhedral model in section 5.2. We use the decision algorithm presented by McKinley in [12] since it does not perform an exhaustive search of transformations, instead presents a method to intelligently build the most efficient loop structure using the results of *cache reuse metric* calculation. We extend McKinley's algorithm to use strip-mining (well studied in affine literature) to further improve the results presented by McKinley.

Chapter 9

Conclusions and Future Work

9.1 Conclusions and Future Work

In this thesis we have developed techniques to parallelize x86 binaries using static affine analysis techniques when binary programs do not contain any relocation information, *i.e.*binary code without any symbolic information about arrays, their sizes etc. in them. We have shown that not only can we parallelize such binaries but also take decisions about loop nest transformation using the three main contributions of this work: (i) to obtain linearized multi-dimensional expressions for each memory accesses in the loop and use linear algebraic techniques on them to obtain distance vectors; (ii) use these linearized multi-dimensional expressions to guess the loop bounds of loops whose bounds are run-time dependent and then run the parallel version of the loop if indeed the bounds were within these guesses; and (iii) use the linearized multi-dimensional expressions to calculate the cache reuse metric and then take decisions to transform the loop nest.

We present results on dense affine matrix code from the polybench benchmark suite and the affine benchmarks from the SPEC2006 and OMP2001 benchmark suites. In all the cases our results from binary show that they can scale as well source code. This shows that (i) the techniques we have developed from binaries are very powerful and equivalent to source code techniques; (ii) our methods are very scalable to large benchmarks very well since we show results on all the affine benchmarks from SPEC2006 and OMP2001.

In future I think this work will benefit from the following directions:

(i) Integrate the affine parallelizer with non-affine parallelization techniques to increase the scope of benchmarks parallelized.

(ii) Integrate a more sophisticated decision algorithm to combine many more loop transformations. Many such algorithms have been presented in the source literature and can be adapted to binaries.

(iii) A polyhedral compiler called Polly is under development within the LLVM infrastructure. It would be interesting to see if we can use our techniques to feed Polly directly from binary code using the techniques we have presented in chapter 6.

Bibliography

[1] Lyle. Cpu trends. `http://techtalk.pcpitstop.com/research-charts-cpu/`, May 2008.

[2] OpenMP Architecture Review Board. OpenMP C and C++ application program interface, version 1.0. `http://www.openmp.org`, 1998.

[3] OpenMP Review Board. OpenMP Fortran application program interface, version 2.0. `http://www.openmp.org`, 2000.

[4] The Message Passing Interface (MPI) standard. `http://www-unix.mcs.anl.gov/mpi/`, 2007.

[5] Lorin Hochstein, Jeff Carver, Forrest Shull, Sima Asgari, and Victor Basili. Parallel programmer productivity: A case study of novice parallel programmers. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 35, Washington, DC, USA, 2005. IEEE Computer Society.

[6] Ken Kennedy and John R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[7] Michael Joseph Wolfe. *Optimizing supercompilers for supercomputers*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1982.

[8] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. Parallel Distrib. Syst.*, 2(4):452–471, 1991.

[9] 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 *Proceedings of the workshop on languages and compilers for parallel computing*, pages 10–1. Springer-Verlag, Berlin/Heidelberg, 1994.

[10] Utpal K. Banerjee. Unimodular transformations of double loops. In *In proceedings of the third Workshop on Languages and Compilers for Parallel Computing*, pages 192–219, August 1990.

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

[12] Kathryn S. McKinley. A compiler optimization algorithm for shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 1998.

[13] Utpal Banerjee. *Speedup of ordinary programs*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1979.

[14] Ross Albert Towle. *Control and data dependence for program transformations.* PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1976.

[15] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Boston, 1988.

[16] Gina Goff, Ken Kennedy, and Chau-Wen Tseng. Practical dependence testing. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 15–29, New York, NY, USA, 1991. ACM Press.

[17] William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 4–13, New York, NY, USA, 1991. ACM.

[18] Rudolf Eigenmann, Jay Hoeflinger, Greg Jaxon, Zhiyuan Li, and David A. Padua. Restructuring fortran programs for cedar. *Concurrency - Practice and Experience*, 5:553–573, 1993.

[19] D. Kuck, E. Davidson, D. Lawrie, A. Sameh, C. Q. Zhu, A. Veidenbaum, J. Konicek, P. Yew, K. Gallivan, W. Jalby, H. Wijshoff, R. Bramley, U. M. Yang, P. Emrath, D. Padua, R. Eigenmann, J. Hoeflinger, G. Jaxon, Z. Li, T. Murphy, and J. Andrews. The cedar system and an initial performance study. *SIGARCH Comput. Archit. News*, 21(2):213–223, May 1993.

[20] Rudolf Eigenmann, Jay Hoeflinger, and David Padua. On the automatic parallelization of the perfect benchmarks&#174. *IEEE Trans. Parallel Distrib. Syst.*, 9(1):5–23, January 1998.

[21] Michael Edward Wolf. *Improving locality and parallelism in nested loops.* PhD thesis, Stanford University, Stanford, CA, USA, 1992.

[22] Jennifer Anderson, Saman P. Amarasinghe, and Monica S. Lam. Data and computation transformations for multiprocessors. In *In Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 166–178, 1995.

[23] Mary H. Hall, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, and Monica S. Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 49, New York, NY, USA, 1995. ACM Press.

[24] Amy W. Lim, Gerald I. Cheong, and Monica S. Lam. An affine partitioning algorithm to maximize parallelism and minimize communication. In *ICS '99: Proceedings of the 13th international conference on Supercomputing*, pages 228–237, New York, NY, USA, 1999. ACM.

[25] Martin Griebl. Automatic parallelization of loop programs for distributed memory architectures, 2004.

[26] Pierre Boulet, Alain Darte, and Georges andr Silber. Loop parallelization algorithms: From parallelism extraction to code generation. *Parallel Computing*, 24, 1997.

[27] Cédric Bastoul and Paul Feautrier. Improving data locality by chunking. In *Proceedings of the 12th international conference on Compiler construction*, CC'03, pages 320–334, Berlin, Heidelberg, 2003. Springer-Verlag.

[28] Cedric Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 7–16, Washington, DC, USA, 2004. IEEE Computer Society.

[29] 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 '08, pages 101–113, New York, NY, USA, 2008. ACM.

[30] David L. Kuck. *Structure of Computers and Computations*. John Wiley & Sons, Inc., New York, NY, USA, 1978.

[31] Fabien Quilleré, Sanjay Rajopadhye, and Doran Wilde. Generation of efficient nested loops from polyhedra. *Int. J. Parallel Program.*, 28(5):469–498, October 2000.

[32] Andrew W. Appel and Maia Ginsburg. *Modern Compiler Implementation in C*. Cambridge University Press, January 1998.

[33] Michael E. Wolf, Dror E. Maydan, and Ding-Kai Chen. Combining loop transformations considering caches and scheduling. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 29, pages 274–286, Washington, DC, USA, 1996. IEEE Computer Society.

[34] 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 '08, pages 101–113, New York, NY, USA, 2008. ACM.

[35] A Kotha K Anand M Smithson G Yellareddy R Barua. Automatic parallelization in a binary rewriter. In *MICRO 43: Proceedings of the 43rd annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society Press, 2010.

[36] P O'Sullivan, K Anand, A Kotha, M Smithson, R Barua, and A D. Keromytis. Retrofitting security in cots software with binary rewriting. In *Proceedings of the 26th International Information Security Conference*, 2011.

[37] Kapil Anand, Matthew Smithson, Khaled ElWazeer, Aparna Kotha, Jim Gruen, Nathan Giles, and Rajeev Barua. A compiler level intermediate representation based binary analysis and rewriting system. In *To Appear in European Conference on Computer Systems*, 2013.

[38] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (GCO)*, pages 75–87, 2004.

[39] Alan Eustace and Amitabh Srivastava. Atom: a flexible interface for building high performance program analysis tools. In *TCON'95: Proceedings of the USENIX 1995 Technical Conference Proceedings on USENIX 1995 Technical Conference Proceedings*, pages 25–25, Berkeley, CA, USA, 1995. USENIX Association.

[40] B. Schwarz, S. Debray, and G. Andrews. Plto: A link-time optimizer for the intel ia-32 architecture. *In Proc. 2001. Workshop on Binary Rewriting (WBT)*, 2001. `citeseer.ist.psu.edu/schwarz01plto.html`.

[41] Van Put L, Chanet D, De Sutler B De Bus B, and De Bosschere K. Diablo: a reliable , retargetable and extensible link-time rewriting framework. In *Proceedings of the Fifth IEEE International Symposium on Signal Processing and Information Technology, 2005*, pages 7–12. IEEE, 2005.

[42] Andrew Edwards, Amitabh Srivastava, and Hoi Vo. Vulcan: Binary transformation in a distributed environment. Technical Report MSR-TR-2001-50, Microsoft Research, April 2001.

[43] M Smithson, K Anand, A Kotha, K Elwazeer, N Giles, and R Barua. Binary rewriting without relocation information. Technical report, University of Maryland, College Park, November 2010.

[44] clang: a C language family frontend for LLVM. `http://clang.llvm.org/`.

[45] DragonEgg - Using LLVM as a GCC backend. `http://dragonegg.llvm.org/`.

[46] Benoit Pradelle, Alain Ketterlin, and Philippe Clauss. Polyhedral parallelization of binary code. *ACM Trans. Archit. Code Optim.*, 8(4):39:1–39:21, January 2012.

[47] Benoît Pradelle. *Static and dynamic methods of polyhedral compilation for an efficient execution in multicore environments.* PhD thesis, Intel, 2011.

[48] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolmen, Wayne Wong, Hank Levy, and Brian N Bershad. Instrumentation and optimization of win32/intel executables. *USENIX Windows NT Workshop*, August 1997.

[49] Bjorn De Sutter, Bruno De Bus, and Koen De Bosschere. Link-time binary rewriting techniques for program compaction. *ACM Trans. Program. Lang. Syst.*, 27(5):882–945, 2005.

[50] Saumya Debray, William Evans, and Robert Muth. Compiler techniques for code compression. Technical report, University of Arizona, Tucson, AZ 85721, USA, April 1999. `http://citeseer.ist.psu.edu/243038.html`.

[51] Robert Muth, Saumya K. Debray, Scott Watterson, and Koen De Bosschere. Alto: a link-time optimizer for the compaq alpha. *Softw. Pract. Exper.*, 31(1):67–101, 2001.

[52] Robert Cohn, David Goodwin, P. Geoffrey Lowney, and Norman Rubin. Spike: an optimizer for alpha/nt executables. In *NT'97: Proceedings of the USENIX Windows NT Workshop on The USENIX Windows NT Workshop 1997*, pages 3–3, Berkeley, CA, USA, 1997. USENIX Association.

[53] Robert S. Cohn, David W. Goodwin, and P. Geoffrey Lowney. Optimizing alpha executables on windows nt with spike. *Digital Tech. J.*, 9(4):3–20, 1998.

[54] Anand K and et. al. Decompilation to compiler high ir in a binary rewriter. Technical report, University of Maryland, College Park, November 2010.

[55] William Blume, Ramon Doallo, Rudolf Eigenmann, John Grout, Jay Hoeflinger, Thomas Lawrence, Jaejin Lee, David Padua, Yunheung Paek, Bill Pottenger, Lawrence Rauchwerger, and Peng Tu. Parallel programming with polaris. *Computer*, 29(12):78–82, 1996.

[56] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. Suif: an infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Not.*, 29(12):31–37, 1994.

[57] Mary W. Hall, Jennifer M. Anderson, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica S. Lam. Maximizing multiprocessor performance with the suif compiler. *Computer*, 29(12):84–89, 1996.

[58] Manish Gupta, Sam Midkiff, Edith Schonberg, Ven Seshadri, David Shields, Ko-Yang Wang, Wai-Mee Ching, and Ton Ngo. An hpf compiler for the ibm sp2. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 71, New York, NY, USA, 1995. ACM Press.

[59] Hideki Saito, Nicholas Stavrakos, Steven Carroll, Constantine D. Polychronopoulos, and Alexandru Nicolau. The design of the promis compiler. In *CC '99: Proceedings of the 8th International Conference on Compiler Construction, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99*, pages 214–228, London, UK, 1999. Springer-Verlag.

[60] Constantine D. Polychronopoulos, Milind B. Girkar, Mohammad Reza Haghighat, Chia Ling Lee, Bruce Leung, and Dale Schouten. Parafrase-2: an environment for parallelizing, partitioning, synchronizing, and scheduling programs on multiprocessors. *Int. J. High Speed Comput.*, 1(1):45–72, 1989.

[61] Efe Yardimci and Michael Franz. Dynamic parallelization and mapping of binary executables on hierarchical platforms. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 127–138, New York, NY, USA, 2006. ACM.

[62] Ben Hertzberg. *Runtime Automatic Speculative Parallelization of Sequential Programs*. PhD thesis, Stanford University, 2009.

[63] Cheng Wang, Youfeng Wu, Edson Borin, Shiliang Hu, Wei Liu, Dave Sager, Tin-fook Ngai, and Jesse Fang. Dynamic parallelization of single-threaded binary programs using speculative slicing. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 158–168, New York, NY, USA, 2009. ACM.

[64] Yang J, Skadron K, Soffa M, L, , and Whitehouse K. Feasibility of dynamic binary parallelization. In *In Proceedings of the Workshop on Hot Topics in Parallelism*, 2011.

[65] Aparna Kotha, Kapil Anand, Timothy Creech, Khaled ElWazeer, Matthew Smithson, and Rajeev Barua. Affine parallelization of loops with run-time dependent bounds from binaries. In *Under review at International Conference on Supercomputing*, 2013.

[66] Aparna Kotha, Kapil Anand, Timothy Creech, Khaled ElWazeer, Matthew Smithson, Greeshma Yellareddy, and Rajeev Barua. Affine parallelization using dependence and cache analysis in a binary rewriter. In *Under review at IEEE transactions on parallel and distributed systems*, 2013.

[67] Takashi Nakamura, Satoshi Miki, and Shuichi Oikawa. Automatic vectorization by runtime binary translation. In *Proceedings of the 2011 Second International Conference on Networking and Computing*, ICNC '11, 2011.

[68] Anshuman Dasgupta. *Vizer: A Framework to Analyze and Vectorize*. PhD thesis, Rice University, 2002.

[69] Vadim Maslov. Delinearization: an efficient way to break multiloop dependence equations. In *In Proc. the SIGPLAN'92 Conference on Programming Language Design and Implementation*, pages 152–161, 1992.

[70] Björn Franke and Michael O'boyle. Array recovery and high-level transformations for dsp applications. *ACM Trans. Embed. Comput. Syst.*, 2(2):132–162, May 2003.

[71] Nethercote and et.al. The cache behaviour of large lazy functional programs on stock hardware. In *Proceedings of the 2002 workshop on Memory system performance*, 2002.

[72] J Weidendorfer and et.al. A tool suite for simulation based analysis of memory access behavior. In *In Proceedings of International Conference on Computational Science*, 2004.