Compiler Techniques for Transformation Verification, Energy Efficiency and Cache Modeling

DISSERTATION

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Wenlei Bao, M.S.

Graduate Program in Computer Science and Engineering

The Ohio State University

2018

Dissertation Committee:

Prof. P. Sadayappan, Advisor

Prof. Gagan Agrawal

Prof. Radu Teodorescu

Prof. Louis-Noel Pouchet

Dr. Sriram Krishnamoorthy

ABSTRACT

Performance has been the focus of computer systems for decades, from past Moore law to current parallel computers. Compiler optimizations are used to improve performance by generating code to utilize hardware (e.g. cache)component efficiently. However, modern systems such as large scale system require not only performance but also resilience and energy efficiency. Increasing concern of system resilience and energy efficiency has been shown in both industry and academia.

Errors within applications, especially those escape from detection and resulting in silent data corruption, are extremely problematic. Thus, in order to improve the resilience of applications, error detection and vulnerability characterization techniques are an important step towards fault tolerant applications.

Compiler transformations, which restructure programs to improve performance by leveraging data locality and parallelism, are often complex and possibly involve bugs that leads to errors in transformed programs. Thus it is essential to guarantee the correctness, however, current approaches suffers from various problems such as transformations supported or space complexity etc. This dissertation presents a novel approach that performs dynamic verification by inserting lightweight checker codes to detect errors of transformations. The errors are exposed by the execution of checker-inserted transformed program if exist.

Energy efficiency is of increasingly importance in scenarios ranging from battery-operated devices to data centers striving for lower energy costs. Dynamic voltage and frequency

scaling (DVFS) adapts CPU power consumption by modifying processor frequency to improve energy efficiency. Typical DVFS approaches involve default strategies such as reacting to the CPU runtime load to adapt frequency, which have inherent limitations because of processor-specific and application-specific effects. This dissertation developed a novel compile-time characterization to select frequency and number of CPU cores to use, which providing significant additional benefits over the runtime approach.

Cache memory, as one of the most fundamental components of modern processors, has a significant impact on the performance of current computer systems. Compiler optimizations on efficient use of cache to reduce data movement, are often based on very approximate cost models due to the lack of precise modeling of hierarchical cache. The challenge of accurately modeling cache misses has made trace-based simulation the current method of choice. This dissertation takes a fundamentally different approach for polyhedral programs, developed a closed-form solution for modeling of misses of set-associative cache by leveraging the power of polyhedral analysis. This solution can enable program transformation choice at compile time to optimize cache misses.

In sum, the dissertation makes contributions to advance compiler technology to achieve program transformation verification, to reduce energy costs, and to effectively modeling cache behaviors.

To my parents and my girl friend, for their love and accompany.

ACKNOWLEDGMENTS

The PhD experience will be one most unforgettable part of my life. This journey is full of wonderful and contradiction moments: the joy of paper acceptances and the disappointment after paper rejects; the exciting for achievement and the exhausting for problems. I am so glad and graceful that I am able to reach this stage. Many thanks to everyone that I met during this wonderful journey, you are all part of my life and missing anyone will not make it complete.

First and foremost, I want to express my gratitude for my advisor, Prof. Saday. Prof. Saday is the best advisor I have ever met during my student life. Starting from his class to the research work and publications, he influenced and educated me in many aspects. His great passison in research motivate and encourage me. His critical insight and comments in discussison help me with my research skills in depth. He also provides me with internship opporitunies in both acamedia and industry to help with my research. My words cannot express how grateful I am for his advisement and support.

I would like to gratefully thank Dr. Sriram Krishnamoorthy and Prof. Louis-Noël Pouchet for the great research collabrations and their mentoring during my PhD. Dr. Sriram is my mentor during internship at PNNL. Most of the papers become great publications because of his brilliant ideas. His insightful advices improve the papers a lot and make them successful. Prof. Louis-Noël educated me how to do research in many aspects, especially in how to performing experimental evaluations in all my papers. I learn a lot from his

excellent suggestions in tuning the papers. I won't have those great publications without their help.

I also want to thank all the labmates at OSU for all the help! Thank you all guys and We had a good time in the lab.

Last but not least, I want to thank my parents for their unselfish love and constant support. I could not make it without their love and support. I hope I can support them well in return in the coming furture. The accompanyship of my girlfriend helps me go through the hard time, I do not know what it could be without her support. This dissertation also belongs to them.

VITA

Sep. 2006 – Jul. 2010	Bachelor of Science, EE Harbin Institute of Technology Harbin, China.
Sep. 2010 – Jul. 2012	Master of Science, EE Harbin Institute of Technology Harbin, China.
Sep. 2012 – Aug. 2014	Master of Science, ECE The Ohio State University Columbus, Ohio.
Sep. 2014 – Dec. 2016	Master of Science, CSE The Ohio State University Columbus, Ohio.
Aug. 2012 – Present	Graduate Research Assistant The Ohio State University Columbus, Ohio.
May. 2014 – Aug. 2014	Intern Pacific Northwest National Laboratory Richland, Washington.
May. 2015 – Jul. 2015	Intern Pacific Northwest National Laboratory Richland, Washington.
May. 2017 – Dec. 2017	Intern NVidia Corporation Redmond, Washington.

PUBLICATIONS

Wenlei Bao, Sriram Krishnamoorthy, Louis-Noël Pouchet, P. Sadayappan
Analytical modeling of cache behavior for affine programs.
ACM SIGPLAN Symposium on Principles of Programming Languages (POPL), Jan. 2018.

Wenlei Bao, Prashant Rawat, Martin Kong, Sriram Krishnamoorthy, Louis-Noël Pouchet, P. Sadayappan
Efficient Cache Simulation for Affine Computations.
International Workshop on Languages and Compilers for Parallel Computing (LCPC), Oct. 2017.

Wenlei Bao, Changwan Hong, Sriram Krishnamoorthy, Louis-Noël Pouchet, P. Sadayappan
Hybrid Static/Dynamic Frequency Scaling on Multicore CPUs.
ACM Transactions on Architecture and Code Optimization (TACO), Nov. 2017.

Changwan Hong, Wenlei Bao, Albert Cohen, Sriram Krishnamoorthy, Louis-Noël Pouchet, J. Ramanujam, Fabrice Rastello, P. Sadayappan
Effective Padding of Multi-Dimensional Arrays to Avoid Cache Conflict Misses.
ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI), Jun. 2016.

Wenlei Bao, Louis-Noël Pouchet, Fabrice Rastello, P. Sadayappan
Polycheck: Dynamic verification of iteration space transformations on affine programs
ACM SIGPLAN Symposium on Principles of Programming Languages (POPL), Jan. 2016.

FIELDS OF STUDY

Major Field: Computer Science and Engineering

Studies in High Performance Computing: Prof. P. Sadayappan

TABLE OF CONTENTS

Page

Abstract ii

Dedication iv

Acknowledgments v

Vita vii

List of Figures xii

List of Tables xiv

List of Algorithms xvi

Chapters:

1. Introduction 1

2. Dynamic Verification of Iteration Space Transformations on Affine Programs . 5

   2.1 Introduction 5
   2.2 Motivation and Overview 8
   2.3 Background 12
      2.3.1 Integer Sets Notation 13
      2.3.2 Polyhedral Dependences 16
   2.4 Verifying Transformations on Affine Programs 16
      2.4.1 Algorithm A: A General Algorithm for all Affine Input Programs 17
      2.4.2 Algorithm B: A Version-Number based Algorithm 23
      2.4.3 Illustrative Example: Seidel 27
      2.4.4 Time and Space Complexity 34

2.5 Scope of Applicability, Enhancements, and Limitations . . . 35
2.6 Experimental Evaluation . . . 39
2.6.1 Evaluation Using the PoCC Polyhedral Compiler . . . 40
2.6.2 Evaluation Using Cilk and the Pochoir Stencil Compiler . . . 47
2.6.3 Identifying Bugs in PolyOpt/C . . . 50
2.6.4 PolyCheck Overhead . . . 50
2.7 Conclusions . . . 53

3. Static and Dynamic Frequency Scaling on Multicore CPUs . . . 55

3.1 Introduction . . . 55
3.2 Motivation and Overview . . . 57
3.3 Adaptive Runtime to Optimize Energy Efficiency . . . 61
3.4 Static Analyses . . . 65
3.4.1 Approximating Operational Intensity . . . 65
3.4.2 Parallelism Features . . . 74
3.5 Processor Characterization . . . 76
3.5.1 One-Time Machine Profiling . . . 76
3.5.2 Decision Tree for Frequency/Core pair . . . 77
3.6 Experimental Results . . . 78
3.6.1 Experimental Protocol . . . 78
3.6.2 Summary of Results . . . 80
3.6.3 Comparison with Runtime DVFS . . . 83
3.6.4 Summary of Intel ICC and GNU GCC experiments . . . 85
3.6.5 Summary of Features . . . 87
3.7 Phase Analysis . . . 88
3.7.1 Phase Characterization . . . 89
3.7.2 Possible Improvements . . . 90
3.8 Conclusion . . . 91

4. Static Analysis of Hierarchical Set-Associative Cache Behavior for Affine Programs . . . 92

4.1 Introduction . . . 92
4.2 Overview of Modeling Approach . . . 95
4.3 Program Representation . . . 100
4.3.1 Modeling Integer Tuples . . . 101
4.3.2 Representing Programs . . . 104
4.4 Single-Level Cache Analysis . . . 106
4.4.1 Modeling Cache Accesses . . . 107
4.4.2 Miss Events in Set-Associative Caches . . . 108
4.4.3 Algorithm for Miss Calculation . . . 110

4.5 Cache Writing Policies and Hierarchical Caches . . . 110
4.6 Cache Modeling Across Program Phases . . . 114
4.6.1 Final Cache State . . . 115
4.6.2 Initial Cache State . . . 118
4.7 Experimental Evaluation . . . 119
4.7.1 Experimental Setup . . . 119
4.7.2 Evaluation of Hierarchical Set-Associative Cache . . . 119
4.7.3 Evaluation of Write Policies . . . 122
4.7.4 Evaluation of Loop Tiling . . . 124
4.7.5 Evaluation of Approximation Heuristics . . . 124
4.8 Time Complexity of PolyCache . . . 128
4.9 Discussion and Pratical Uses . . . 130
4.10 Conclusion . . . 132

5. Cache Vulnerability Analysis for Affine Programs . . . 137

5.1 Introduction . . . 137
5.2 Modeling Cache Vulnerability . . . 138
5.2.1 Cache Vulnerability . . . 139
5.2.2 Modeling Vulnerability Intervals . . . 141
5.2.3 From Vulnerability Intervals to Vulnerability Metrics . . . 144
5.3 Experimental Evaluation . . . 145
5.3.1 Implementation detail . . . 145
5.3.2 Time complexity Evaluation . . . 146
5.4 Conclusion . . . 150

6. Related Work . . . 151

6.1 Verification of Program Transformation . . . 151
6.2 Energy Optimization through DVFS . . . 152
6.3 Cache Behavior Modeling and Vulnerability Analysis . . . 154

7. Conclusion . . . 158

7.1 Conclusion . . . 158

Bibliography . . . 159

LIST OF FIGURES

Figure **Page**

2.1 Example input program (a) and transformed programs to be verified ((b)–(f)). ISA can verify (b). Our approach can dynamically verify all transformed versions against the input. 6

2.2 Iterative Seidel example based on the language specification in Figure 2.3. T and N are problem size parameters. 10

2.3 Language for affine loop programs. Indentation-based scoping (used for readability) is not shown. f_e is an expression of its arguments. 13

2.4 Runtime procedure to check every operation encountered in the transformed program for algorithm A. 24

2.5 Runtime procedure to check every operation encountered in the transformed program for algorithm B. 26

2.6 Recursive implementation of Seidel. Invoked as `seidel_rec(0,1,3,1,3,2,4,A)`. 27

2.7 The checker code inserted by Algorithm A after line 4 in the recursive Seidel implementation (Figure 2.6). Note that the compiler can optimize away the first four assert statements. 33

2.8 The checker code inserted by Algorithm B after line 4 in the recursive Seidel implementation (Figure 2.6). 34

2.9 Checker running time with fixed tiling. 52

2.10 Checker running time with parametric tiling. 52

2.11 Checker running time across problem size (LU). 53

2.12 Checker running time across problem size (Reg_detect). 53

3.1 Energy for DGEMM/MKL (top row) and Jacobi 2D (bottom row) 59

3.2 Comparison of energy savings for `on-demand` Linux governor and our proposed runtime, versus `powersave` 64

3.3 Jacobi 2D 5 point sweep 67

3.4 Details of static approach 80

3.5 Summary of experimental results 82

3.6 Energy savings versus runtime DVFS schemes 84

3.7 EDP savings versus runtime DVFS schemes 84

3.8 Different types of phases on Haswell / 4 cores 90

4.1 Illustrative example of a 2-way set-associative cache 95

4.2 Triangular Matrix-Multiply 104

LIST OF TABLES

Table **Page**

2.1 FirstWriter to array `A` in iterative Seidel 27

2.2 Statement instances executed by iterative Seidel and related relations 30

2.3 Sequence of operations for recursive Seidel; "Sdw" abbreviates shadow in the column headers 31

2.4 Sample operations for recursive Seidel with error; "Sdw" abbreviates shadow in the column headers 32

2.5 Benchmarks in evaluation using PoCC 40

2.6 Summary of tool coverage (PoCC evaluation) 44

2.7 Summary of bugs tested (PoCC evaluation) 45

2.8 Errors found in transformations by PolyOpt/C 0.2.0 48

2.9 Cilk and Pochoir benchmarks 49

2.10 Summary of evaluation using Cilk and Pochoir benchmarks 49

2.11 Overhead comparison with CodeThorn [109] 51

3.1 Processor characteristics 58

3.2 Energy efficiency summary 86

3.3 Energy-Delay Product efficiency summary 86

3.4 Summary of Features 88

3.5 Benchmark features 88

4.1 Handling of write operations by the different cache writing policies considered 112

4.2 Summary of No-Write Allocate Write Back Cache 123

4.3 Summary of Tiling (PolyCache is Poly. below) 125

4.4 Summary of Set-0 Heuristic 127

4.5 Per-operator Execution Time *s*: set, *m*: map, *x*: set or map, *p*: integer . . . 129

4.6 Benchmarks and Summary of Hierarchical Set-Associative Cache(a) 134

4.7 Benchmarks and Summary of Hierarchical Set-Associative Cache(b) 135

4.8 Summary of Per-Array Heuristic 136

5.1 Lifetime Classification of Write-Back Cache 140

5.2 Execution time experiments for problem size N=8 147

5.3 Execution time experiments for problem size N=16 148

5.4 Execution time experiments for problem size N=32 149

LIST OF ALGORITHMS

Algorithm		Page
1	Runtime DVFS algorithm	62
2	EstimateCacheMisses	72
3	Estimate Operational Intensity	73
4	Check Poor Parallel Scaling	75
5	SingleLevelMisses: Compute Misses for Single-level Cache	111
6	Compute Dirty Evictions: Writes To Next Cache Level For Write-Back Caches	113
7	FinalAccess(Refs)	116
8	Determine Dirty Cache Lines In The Final Cache State For Write-Back Caches	117
9	unACE-idle: Compute unACE idle component	142
10	Lifetime components unACE: read-to-write, read-to-evict, write-to-write ACE: read-to-read, write-to-read, write-to-evict	143
11	CVF: Compute vulnerability factor	144

CHAPTER 1

Introduction

Continuous technology scaling keeps enhancing the performance of computer systems from increasing processor frequency to parallel processing units such as multi-core and many-core processors. Higher performance has been the goal and thus developing focus for few decades. However, problems such as system errors and energy consumption become increasing concern along with the development. Thus, modern computer systems such as large scale system demand not only performance but also resilience and energy efficiency [22].

Therefore, developing automated compiler approaches to detect errors, improving the fault tolerance and energy efficiency have become an crucial and urgent requirement for both industry and academia.

Error Detection in Compiler Transformation. Optimizing programs via compiler transformations such as loop tiling to gain performance benefits by improving data locality and parallelization are critical techniques, however, often complex and possibly involve bugs that leads to errors.

This dissertation addresses the problem by designing dynamic verification approach that perform data-flow analysis on input affine program and generate lightweight checker

codes that inserted into transformed program (Chapter 2). With the execution of checker-inserted program, any iteration reordering errors will be reported. This novel dynamic approach attacked the problems of previous approaches including coverage of transformations, sensitivity to test datasets and space complexity. It could handle arbitrary iteration reordering transformations even non-affine transformations. Experimental results assess the correctness and effectiveness of proposed method and its increased coverage over previous approaches.

Improving Energy Efficiency. Energy consumption becomes an crucial concern in recent years along with the increasing demands of high performance from mobile systems to large scale systems. Dynamic voltage and frequency scaling (DVFS) as a classic mechanism in processors that enables a trade-off between performance and energy, has been widely studied to improve energy efficiency by adapting power consumption. Previous DVFS approaches dynamically adjust operated CPU frequency according to current workload. Although energy savings have been shown using these approaches, they suffer from inherent limitations, such as not accounting for processor-specific impact of frequency changes on energy for different workload types.

To tackle this problem, a lightweight runtime approach is first proposed to adapt the frequency, and then show that further improvements can be achieved for affine programs using a compile-time characterization instead of run-time monitoring to select the frequency and number of CPU cores (Chapter 3). A one-time energy characterization of CPU-specific DVFS profiles followed by a compile-time categorization of loop-based code segments in application are combined to determine a priori frequency and number of cores to execute

so as to minimize energy or energy-delay product. Experimental evaluation on multiple CPUs shows that proposed approach outperforms the dynamic approaches.

Analytical Cache Modeling and Vulnerability Analysis. Cache, as one of the most important part of modern processors, is used to bridge the speed gap between CPU computing and memory access. The overheads of data movement between memory and cache hierarchy have a significant impact on performance for current computer systems. In order to achieve high performance, programs must exhibit efficient cache memory accesses. Transformations of programs by optimizing compilers are often used to improve cache utilization to get better performance. However, cache behaviors, especially cache misses, are difficult to predict and estimate.

The current simulation approach is the practical choice for accurate modeling of cache behaviors especially cache misses because of the significant challenge of exact modeling of caches. The problem of simulation is that it requires time proportional to the dataset or problem size as well as the number of distinct cache configurations to be evaluated.

Moreover, cache is also one of the most vulnerable components. But no compiler techniques proposed to protect it from soft errors or even characterize the vulnerability, which is the measure of failure rate, other than simulation.

Therefore developing analytical cache model and compiler approaches to estimate or optimize vulnerability is of great interest in the sense of improving performance by studying cache behavior and resilience via optimizing vulnerability.

This dissertation developed a closed-form modeling of misses of set-associative cache by leveraging the polyhedral analysis and counting algorithms on polyhedral sets (Chapter 4). Experimental evaluation demonstrate exact match with trace-based cache simulation,

but with a time complexity essentially independent of the problem size. Moreover, the proposed analytical cache model can be applied to many practical cases such as padding [68], tiling and cache design analysis. Furthermore, we demonstrate in Chapter 5 how to extend the cache model to characterize or even optimize the cache vulnerability of programs.

CHAPTER 2

Dynamic Verification of Iteration Space Transformations on Affine Programs

2.1 Introduction

Optimized programs are often complex and rarely resemble the original source programs [18]. It is difficult, if not impossible, to verify the correctness of such programs through visual inspection. Automated testing is part of the classical arsenal to find bugs in compiler implementations, including production compilers [3, 6] and research compilers [98]. However, a typical challenge occurs with the selection of a relevant dataset for the program to expose a bug. If we limit testing to the bit-by-bit equivalence of outputs produced by the original and transformed programs, what guarantees that a match implies the lack of a bug? A simplistic example would entail testing a program multiplying two matrices, where the input matrices are filled with zeros. For example, a buggy-transformed program stripped of all computation would pass this test.

To address this problem, automatic equivalence checking techniques that prove two programs are equivalent have been developed. For instance, focusing on programs with static affine control- and data-flow, prior approaches employ static verification by systematically deriving one-to-one correspondence between the operations in the original and transformed

programs and ensuring they satisfy the same data dependences [127, 13, 74]. This has the significant advantage of being independent of the input dataset, yet with the *drawback of requiring both the original and transformed programs to be affine programs*. In other words, any iteration reordering transformation that generates non-affine expressions in the transformed program, such as parametric tiling or numerous abstract syntax tree (AST)-based complementary optimizations, cannot be verified with such an approach.

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

(a) Original

```
for (i=0; i<N/32; i++)
  for (j=0; j<32; j++)
    A[i*32+j] = B[i*32+
    j];
for(j=(N/32)*32; j<N; j
    ++)
  A[j] = B[j];
```

(b) Constant sectioning

```
i=0; do { //N and T are
     coprime
  A[i] = B[i];
  i = (i+T)%N;
} while (i!=0);
```

(c) Round-robin

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

(d) Parametric sectioning

```
for (i=0; i<M; i++)
  for(j=idx[i]; j<idx[i
    +1]; j++)
    A[j] = B[j];
// 0=idx[0]<= idx
    [1]<=...<= idx[M]=N
```

(e) Irregular sectioning

```
void fn(int lo, int hi)
    {
  if (lo>hi) return;
  if (lo==hi) A[lo] = B
    [lo];
  else {
    fn(lo, (lo+hi)/2);
    fn((lo+hi)/2+1, hi)
  ; } }
/*Call fn*/ fn(0,N-1);
```

(f) Recursion

Figure 2.1: Example input program (a) and transformed programs to be verified ((b)–(f)). ISA can verify (b). Our approach can dynamically verify all transformed versions against the input.

To date, all proposed solutions to the verification of iteration reordering transformations suffer from at least one major limitation in the supported transformations (e.g., Integer Set Analysis (ISA) [127]), sensitivity to the input dataset (e.g., output difference checking), or space complexity (e.g., CodeThorn [109]), which is the space to store any state (arrays,

instruction traces, etc.). In this work, we develop a novel approach to assess the correctness of ***arbitrary iteration reordering transformations*** on an affine program, including non-affine transformations. The approach is robust to arbitrary input datasets of a given size and only requires space proportional to the program's data space. To achieve this, we design an approach that employs polyhedral data-flow analysis on the affine input program to generate lightweight checker codes that are embedded in the transformed program. The check-equipped transformed code is then run, and information about any violated data dependence or other iteration reordering errors is reported to the user. This enables conclusions about the programs' equivalence for the given input problem size.

We use our PolyCheck tool to verify the correctness of numerous polyhedral compiler optimizations in the Polyhedral Compiler Collection (PoCC) research compiler [8], including non-affine transformations such as parametric tiling and some affine AST-based transformations that are not currently supported by the state-of-the-art affine checker tool ISA [4]. Using PolyCheck, we also verify code generated by Pochoir [118], a DSL-based stencil compiler, and serial execution of Cilk [31], a recursive programming system. By comparing it with the results presented for CodeThorn by Schordan et al. [109], we show that PolyCheck is much more efficient than trace-based equivalence checking schemes. Finally, we demonstrate PolyCheck's ability to identify bugs in the polyhedral compiler PolyOpt/C 0.2.0 [95].

Our approach is orthogonal and complementary to other works that assess the correctness of parallelism transformations (e.g., race detection tools [108, 50]) or out-of-bound array access checks. With this research, we make the following contributions.

- We present the first dynamic bug checker for affine programs transformed by arbitrary iteration reordering transformations using polyhedral analysis at compile-time

to create a lightweight checking code to be executed along with the transformed program.

- Our approach addresses the three main limitations of other iteration reordering checkers because it supports arbitrary loop (tiling, interchange, etc.) and non-affine transformations (parametric tiling, etc.), is not sensitive to the input dataset values, and only requires space proportional to that needed by the original program during its execution.
- We demonstrate the effectiveness of our approach in asserting the correctness of Pochoir and Cilk programs and through numerous passes of PoCC, a research polyhedral compiler that combines both affine and non-affine transformations.
- We show that PolyCheck can detect bugs in PolyOpt/C 0.2.0 [95] that were first detected by CodeThorn and is much more efficient than CodeThorn's trace-based equivalence checker.
- We present and evaluate optimizations to the checker that exploit regularities in the program dependence graph.

2.2 Motivation and Overview

To motivate the need for a verification approach that is robust to arbitrary iteration reordering transformations, we initially use a simple example. We consider the code shown in Figure 2.1(a), performing a copy from array `B` to array `A`. This code is an affine program with static control- and data-flow. With affine programs, loop bounds, conditionals, and array access expressions only involve affine expressions of the surrounding loop iterators and program parameters [47]. Figures 2.1(a) through (f) show various transformed

versions of this program. Static verification tools checking equivalence between affine programs can verify that the version in Figure 2.1(b)—tiled with a constant tile size known at code-generation time—is equivalent to the input program. However, they fail to verify the remaining versions. Figure 2.1(d) shows a code snippet typically generated by parametric tiling techniques that do not require the tile size to be known at code-generation time. Other versions may be manually or automatically generated. The version in Figure 2.1(e) is non-affine, and its correctness depends on the values of the `idx` array. Recursive versions, such as in Figure 2.1(f), can be generated for recursive parallelism or cache obliviousness.

The approach we present in this chapter can dynamically verify all versions shown in Figure 2.1. In contrast to static equivalence, we require an instrumented program to be executed. This makes our equivalence property hold for the problem size used when running the checker, a restriction on the proof of equivalence achieved by static checkers. On the other hand, this also enables us to verify arbitrary program transformations. In this work, we support arbitrary *iteration reordering* transformations, irrespective of how the code is generated to implement them. Specifically, we require the transformation to not change the total number of dynamic instances (e.g., N in the figure) of each syntactic statement (e.g., `S`) or the operations performed by the statement. This set of supported transformations includes all loop transformations (e.g., loop tiling [131], index set splitting [65], etc.). Yet, it also includes *any possible syntax to implement the reordering*, including using non-affine expressions in the generated code, AST-based transformations, etc.

We achieve this by constructing a checker to rewrite each statement in the transformed program by operations to check that statement instance. We illustrate the construction of the checker using the iterative Seidel example shown in Figure 2.2. This example better illustrates the approach than the simpler example in Figure 2.1(a).

```
1  for t = 0 to  T-1:
2    for i = 1 to N-1:
3      for j = 1 to N-1:
4       S1: A[i][j]=A[i-1][j]+A[i][j-1]
```

Figure 2.2: Iterative Seidel example based on the language specification in Figure 2.3. T and N are problem size parameters.

We first statically analyze the affine input (reference) program to determine the read-after-write (true), write-after-read (anti), and write-after-write (output) dependences. Each dependence can be represented as a function (or relation) that takes a statement instance as input and returns the other statement instance associated with the dependence. For example, in Figure 2.2, consider the statement instance `S1<t=1,i=5,j=7>` , denoting the instance of statement `S1` executed at the iteration `(t=1,i=5,j=7)` . This statement instance updates the value previously written at `A[5][7]` by `S1<t=0,i=5,j=7>` (output dependence) and reads values of `A[4][7]` and `A[5][6]` (input dependence).

Now consider the operation executed by an unknown statement in the transformed program to be verified:

```
A[8][9] = A[7][9] + A[8][8]
```

We analyze the array locations accessed by an operation in the transformed program to map it to a statement instance in the input program. The preceding operation instance can be mapped to `S1<t=0,i=8,j=9>` , `S1<t=1,i=8,j=9>` , etc., in the input program. In general, an operation encountered in the transformed program can possibly match multiple statement instances in the input program. Given this relationship, we first try to map this operation to a statement instance in the input program that has not already been mapped, and statement instance induced by the mapping satisfies the input program's dependences.

A transformed program is declared to be equivalent to an input program if (a) the statement instances in the transformed program can be mapped in a bijective, or one-to-one,

fashion to statement instances in the input program, and (b) each statement instance in such a bijection satisfies the same dependences as in the input program. For arbitrary program transformations, there are numerous ways in which the statement instances of the transformed program can be mapped to those of the input program. Enumerating each mapping to check whether or not it constitutes a dependence-preserving reordering of the statement instances in the input program is prohibitively expensive. We exploit the fact that iteration reordering transformations only change the order in which the statement instances are executed and not the variables written by a given statement instance. We also show that this enables us to check just one mapping to statement instances in the input program.

We can track dependences through the *data space* in terms of the data elements accessed by the statement instances. For each statement instance storing a value in a data element, we use a shadow variable to store the identity of the corresponding input statement instance. For example, if the preceding operation is mapped to `S1<t=1,i=8,j=9>` , the assignment to `A[8][9]` is augmented with:

```
shadow(A[8][9]) = S1<t=1,i=8,j=9>
```

This information can then be used to check the dependence as the program is executed. The checker constructed for this particular statement instance is as follows:

```
assert shadow(A[8][9]) == S1<t=0,i=8,j=9>
assert shadow(A[7][9]) == S1<t=1,i=7,j=9>
assert shadow(A[8][8]) == S1<t=1,i=8,j=8>
shadow(A[8][9]) = S1<t=1,i=8,j=9>
```

where NIL is the initial value of all array locations and variables.

We leverage the fundamental property that if the input program is affine then these true dependences and relationship between statement instances can be represented in a closed form. Together with the shadow variables, this affords compact on-the-fly tracking of the dependences that need to be satisfied. We embed the code to check the equivalence of traces

directly in the transformed program. This code can be emitted at compile time because of the input program's affine characteristics.

Note that we only perform static analysis on the input program. Beyond the constraint that the transformed program is an iteration reordered version of the input program, we do not constrain the transformed program. We do not need to perform any analysis on the transformed program. Rather, we only inspect the operations executed by the transformed program, irrespective of how they are generated. In this example, the operation `A[8][9] = A[7][9] + A[8][8]` could have been generated from any code structure with the array input expressions being complex non-affine operations that evaluate to the array index expressions `(8,9)` , `(7,9)` , and `(8,8)` , respectively.

The determination of the one-to-one correspondence requires that the input program statement instance can be computed from the information available in the operations executed by the transformed program. Additional checks are required to detect errors in the transformed program caused by omitted or duplicated statements or operations that do not have a corresponding statement instance in the input program. We will discuss these details and present optimizations to reduce the checking overhead.

2.3 Background

In this section, we present the basic notation that relates to analysis of affine programs used in the rest of the chapter.

To clarify the notation and types of programs, we consider the language for affine programs defined in Figure 2.3. While we use this language for discussion, our implementation handles C programs that conform to this specification. Note that scalars can be treated

$$
\begin{aligned}
n &\in \{M, N, , ..\} && [\textit{ProblemSizeParameters}] \\
v &\in \mathbb{Z} && [\textit{Values}] \\
l &\in \{l_1, l_2, \dots\} && [\textit{LoopIndices}] \\
A &\in \{A_1, A_2, \dots\} && [\textit{ArrayNames}]
\end{aligned}
$$

$$
\begin{aligned}
a \in \textit{AffineExpr} &::= v \mid l \mid v \times l \mid a + a \mid a - a \\
i \in \textit{ArrayIndex} &::= a(,a)* \\
lb \in \textit{LowerBound} &::= a \mid \textbf{max}(a(,a)+) \\
ub \in \textit{UpperBound} &::= a \mid \textbf{min}(a(,a)+) \\
ac \in \textit{AffineCond} &::= a > 0 \mid a < 0 \mid a == 0 \mid a! = 0 \\
&\quad \mid ac \ \textbf{and} \ ac \mid ac \ \textbf{or} \ ac \\
s \in \textit{Stmts} &::= A[i] = f_e(v \mid A[i](, v \mid A[i])*) \\
&\quad \mid s; s \mid \textbf{if} \ ac : s \\
&\quad \mid \textbf{for} \ l = lb \ \textbf{to} \ ub : s \\
p \in \textit{Programs} &::= s
\end{aligned}
$$

Figure 2.3: Language for affine loop programs. Indentation-based scoping (used for readability) is not shown. f_e is an expression of its arguments.

as one-dimensional arrays of size 1. One significant distinction is the use of a comma-separated list of expressions to represent an array index.[1]

2.3.1 Integer Sets Notation

Programs with affine data-flow and static control-flow are called static control parts (SCoP) [48, 63], roughly defined as a sequence of statements such that all loop bounds and conditional expressions are affine functions of enclosing loop iterators and variables that are constant during the SCoP execution (whose values may be unknown at compile time). Affine programs are represented in this work using (a union of) convex sets of integer tuples and (a union of) integer maps. Operations on these structures are readily available in the

[1]This representation of a tuple without surrounding parenthesis ('()' or '[]') allows us to treat function argument lists, array indices, and loop iterator values interchangeably.

ISCC calculator [125], which leverages the Integer Set Library [124] to provide operations such as union, intersection, and relation application.

Integer Set The definition of an integer set s is:

$$s = [p_1, ..., p_p] \rightarrow \{[i_1, ..., i_m] : c_1 \wedge ... \wedge c_n\}$$

Where $i_1, ..., i_m$ index the m dimensions of the set (noted $\vec{i}$); $p_1, ..., p_p$ are invariant parameters (noted $\vec{p}$); and $c_1, ..., c_n$ are n Presburger formulae, typically in the form of affine inequalities defining constraints on the values of $\vec{i}$.

Integer sets are used to precisely capture the set of runtime instances of statements in affine programs. Each statement S is associated with an iteration vector $\vec{i}_S$ with one component per surrounding loop, and the values $\vec{i}_S$ can take are captured by defining its iteration space I_S. The iteration space of the statement S_1 in Figure 2.2 is noted I_{S_1} and is:

$$I_{S_1} = [T, N] \rightarrow \{S_1[t, i, j] : 0 \leq t < T \wedge 0 \leq i < N \wedge 0 \leq j < N\}$$

Relation The definition of an integer relation, or map, $r : \vec{i} \mapsto \vec{j}$ is:

$$r = [p_1, ..., p_p] \rightarrow \{[i_1, ..., i_m] \mapsto [j_1, ..., j_n] : c_1 \wedge ... \wedge c_o\}$$

Relations are used to describe the set of memory locations accessed by statements. In polyhedral programs, array subscripts functions are affine expressions of the loop iterators and parameters. We note them $F_S^{A,i}$ for the i-th access to an array A in a statement S. For example, the statement instances of S_1 that writes array $A[i][j]$ in Figure 2.2 has $F_S^{A,1} = (i, j)$ and can be represented as:

$$R_{S_1}^{A_1} = \{S_1[t, i, j] \mapsto A[i_1, j_1] : (i_1 = i) \wedge (j_1 = j)\}$$

We note $R_S^{A_i}$ for the i-th read reference map to array A in statement S, and W_S^A as a write reference map to A in S.

Applying relations to sets The apply operation is defined as:

$$(\vec{x} \in s') \iff (\exists \vec{y} \in s \wedge (\vec{y} \mapsto \vec{x}) \in r)$$

where s' is a new set produced by *apply* of relation r to set s, which can be denoted as $s' = r(s)$.

For instance, the apply operation is used to compute the data space (set of all memory locations accessed) in a loop nest. For example, the write data footprint for array A in Figure 2.2 is $R_{S_1}^{A_1}(I_{S_1})$.

Program execution order A schedule is a relation used to specify the execution order of all statement instances. It maps points in the iteration domain to those in an integer set (the set of timestamps). As such, statement instances in the iteration domain are executed following the lexicographic ordering $\prec$ of their associated timestamp. $\prec$ is defined as $(a_1,\ldots,a_n) \prec (b_1,\ldots,b_m)$ iff there exists an integer $1 \leq i \leq min(n,m)$ s.t. $(a_1,\ldots,a_{i-1}) = (b_1,\ldots,b_{i-1})$ and $a_i < b_i$.

The original program schedule is modeled using $2d+1$ timestamps, where d is the maximal nesting depth in the program [63]. For example, the schedule of S_1 in Figure 2.2 is:

$$Sched_{S_1} = \{S_1[t,i,j] \mapsto [0,t,0,i,0,j,0]\}$$

where each odd dimension of the output space is a scalar dimension whose value denotes the lexical AST ordering of the loops surrounding the statement. For statements surrounded

by less than d loops, the even schedule components associated with the missing loops is set to 0. However, we use a special convention where the last component of the schedule corresponds to a unique identification for the statement (e.g., 1 for S_1) instead of the lexical AST order of the statement in this loop nest.

2.3.2 Polyhedral Dependences

In the polyhedral model, dependences for affine computations (such as flow dependence and output dependence) can be precisely calculated and expressed as relations from a source iteration to a target iteration in the iteration space [58, 99, 100].

Polyhedral dependences can be obtained using tools such as ISL [5]. For example, one flow dependence in the example code in Figure 2.2 is shown as follows:

$$Flow = [T,N] \rightarrow \{S_1[t,i,j] \mapsto S_1[t,i,1+j] : \\ 0 \leq t < T \wedge 1 \leq i < N \wedge 1 \leq j < N-1\}$$

The flow relation shows that a flow dependence exists between iteration (t,i,j) and $(t,i,j+1)$ for statement S_1. The value of $A[i][j]$ written by statement S_1 at iteration (t,i,j) is used later at iteration $(t,i,j+1)$ within the iteration space.

Note that we only consider ***exact dependences***, meaning we only consider the ***last write*** of source iteration for any dependence pair from source to target in write-after-write (WAW or output) and read-after-write (RAW or flow) dependences.

2.4 Verifying Transformations on Affine Programs

In this section, we present our approach for verifying that the transformed program has been obtained by a valid reordering of the iterations of the input program. This involves identifying a one-to-one correspondence between the statement instances in the input and

the transformed program, such that, for each statement instance in the input program, the corresponding statement instance in the transformed program satisfies the same dependences. In general, verifying equivalence requires comparing traces of the two programs to derive a graph isomorphism—an expensive task. We show that transformations restricted to iteration reordering can be verified at a much lower cost.

PolyCheck specification The affine specification of a transformed program, to be used by PolyCheck is provided as follows:

```
/*@ polycheck  start  spec
 <affine  input  program >
*/
<transformed program>
/*@ polycheck  end
*/
```

Note that all operations reachable from the ***segment*** enclosed in the transformed program are expected to be available at instrumentation time. In other words, any function that could be transitively called from `<transformed program>` in the above code shall be instrumented.

2.4.1 Algorithm A: A General Algorithm for all Affine Input Programs

Our approach to verification combines two stages. First, we analyze the *input* affine program to build a series of functions used to extract properties and assertion values from an executed operation in the *transformed* program. Second, we instrument the transformed program to call these functions for each executed operation, checking if the operation's runtime properties match the expected values computed by these functions. In a nutshell, the process is as follows. Assume the transformed program so far is valid, and an operation `o` attempts to write to memory location l. The last valid iteration `S<itrv>` [2] that wrote

[2]Throughout the chapter, we use multi-character identifiers that end in v, e.g. `itrv` , to denote vectors.

l is stored in `shadow(l)` . From there, we can determine what should be `S<itr2v>` , the next iteration writing to l, by evaluating a function we built via static analyis of the input program that provides the *NextWriter* iteration. We can then determine what memory location is accessed by `S<itr2v>` and using an analogous process what iteration `S<itrXv>` was the producing iteration for each memory location accessed by `S<itr2v>` . This information is then checked against the runtime information of `o`: the observed shadow value for all its operands and the memory addresses accessed must all match with the values associated with `S<itr2v>` . If any of these values do not match, then `o` is an invalid operation, and the transformed program does not match the input program. If all operations `o` are valid and every `o` was matched with exactly one iteration `S<itrv>` from the input program, the transformed progam matches the input program.

In the following, we first define the various functions extracted via static analysis to compute the *FirstWriter*, *NextWriter*, etc., instances for a memory location in Sec. 2.4.1, before detailing the runtime checking algorithm in Sec. 2.4.1.

Compile-time analysis of the input program

The first stage of analysis involves producing functions that can be evaluated at runtime in the tranformed code. These functions use various integer set/map operations, such as union ($\cup$), computing the lexicographic minimum (*lexmin*) and maximum (*lexmax*), and application of a map to a set ($M(S)$).

FirstWriter The function $\vec{t} = FirstWriter(A, \vec{w})$ computes which instance $\vec{t}$ is the first one in the input program to write to a specific memory location $\vec{w}$ for array A. To compute this function, we proceed by computing for each array A written in the program a function that returns the timestamp of the first iteration writing to an arbitrary memory location.

We model an arbitrary memory location in array A as a parametric point in a set:

$$Point_A = [\vec{W}] \rightarrow \{[\vec{w}] : \vec{w} = \vec{W}\}$$

where $\vec{w}$ and $\vec{W}$ have the same dimensionality as the array A (e.g., a 2D set for a two-dimensional array). We then express the set of instances that write to A as a map from the array index accessed to the iteration accessing it. For array A written by statement S it is:

$$M_S^A = [\vec{P}] \rightarrow \{[\vec{w}] \mapsto [\vec{i}] : W_S^A(\vec{i}) = \vec{w} \wedge \vec{i} \in I_S\}$$

where $\vec{P}$ is the vector of program parameters. Therefore, the timestamp associated with the instance writing to an arbitrary but unique memory location $\vec{w}$ is:

$$T = Sched_S(M_S^A(Point_A))$$

Finally, given $\mathcal{S}$ the set of statements in the input program, one can build the first instance across the whole program accessing a particular location $\vec{w}$: $FirstWriter(A, \vec{w}) =$

$$lexmin\left(\bigcup_{S \in \mathcal{S}}\left(Sched_S(M_S^A(Point_A)\right)\right)$$

We remark that the preceding sets and relations, including the *lexmin* operation, can be seamlessly computed using, for instance, the ISCC calculator. The *lexmin* returned is an expression (typically a tree of expressions, where leaves are possible *lexmin* values and nodes in the tree are conditions on the numerical values of $\vec{w}$ and $\vec{P}$). To build the function `FirstWriter(A,w)` , we simply translate this tree of expressions to C code. The process is repeated for all written arrays in the program, and each is embedded in the function's code so the function selects the appropriate *lexmin* expression tree to evaluate as a function of the array name considered.

NextWriter The function $\vec{t} = NextWriter(\vec{t}_{prev}, A, \vec{w})$ computes which is the instance $\vec{t}$ in the input program writing to the location $\vec{w}$ of A executing immediately after $\vec{t}_{prev}$ wrote to the same memory location $\vec{w}$. To build this function, we employ methods similar to those for *FirstWriter*. We first model an arbitrary iteration of a program as a parametric point in a set:

$$Iter1 = [\vec{T}] \rightarrow \{[\vec{t}] : \vec{t} = \vec{T}\}$$

where $\vec{t}$ and $\vec{T}$ have $2d+1$ components, the number of dimensions of a timestamp (i.e., output dimensions of scheduling functions). To capture an iteration $\vec{t}$ immediately following another iteration $\vec{t}_{prev}$, we use a slight extension of the definition of M_S^A to add input dimensions and capture $\vec{t}_{prev}$ as follows:

$$N_S^A = [\vec{P}] \quad \rightarrow \quad \{[\vec{w}, \vec{t}_{prev}] \mapsto [\vec{i}] : W_S^A(\vec{i}) = \vec{w} \wedge \vec{i} \in I_{S_1}$$
$$\wedge \vec{t}_{prev} \prec Sched_S(\vec{i})\}$$

Finally $NextWriter(\vec{t}_{prev}, A, \vec{w})$ is built using a similar expression as for $FirstWriter(A, \vec{W})$, simply substituting $M_S^{A,j}$ by $N_S^{A,j}$ and $Point_A$ by $(Iter1, Point_A)$.

WriterBeforeRead $\vec{t} = WriterBeforeRead(\vec{t}_{read}, A, \vec{w})$ computes the instance $\vec{t}$ that last wrote to a location $\vec{w}$ of array A read by iteration $\vec{t}_{read}$ in the input program. This is essential to make sure data dependences are preserved in the programs. Writes must all occur in the input order, and read values must contain the same value as in the input program when a particular instance executes. This function is computed in a manner analogous to *NextWriter*, except instead of stating $\vec{t}_{prev} \prec Sched_S(\vec{i})$ we state $Sched_S(\vec{i}) \prec \vec{t}_{read}$, and compute the *lexmax* of the problem instead of the *lexmin* to have the instance immediately preceding $\vec{t}_{read}$ writing to a location $\vec{w}$. Note that if $A(\vec{w})$ is an input location not yet overwritten by any statement instance (e.g., it is live-in data), then $\vec{t}$ is set to `Init<0>` .

LastWriter The function $\vec{t} = LastWriter(A, \vec{w})$ computes the instance $\vec{t}$ that is last to write to a memory location $\vec{w}$ in the array A. This is simply the converse of the *FirstWriter* function, which is formulated identically except the *lexmax* instead of the *lexmin* is used to find $\vec{t}$.

WriteSet and InputSet We conclude by defining two convenience sets, namely the `WriteSet` and `InputSet`, used to initialize the checking procedure.

The `WriteSet` is the set of array locations written to by the input program, we initialize their shadow to `NIL`. This corresponds to the data space built from the union of the data spaces induced by all write references in the program. Its expression, for an array A, is:

$$WriteSet(A) = \bigcup_{S \in \mathcal{S}} W_S^A(I_S)$$

The `InputSet` is the set of data being read before being written, or being read-only. It is used to initialize the last writing instance to a default starting value `Init<0>`. It is assembled by first building the set of all memory locations being read. For the array A, it is: $ReadSet(A) = \bigcup_{S \in \mathcal{S}} \left(\bigcup_i R_S^{A_i}(I_S) \right)$, where i ranges to cover each read access function to A in S.

We then prune this set from all the memory locations being written before being read, which is obtained by applying *WriterBeforeRead* on each first-read instance per memory location, keeping only locations where it returns `Init<0>`. The set of first-read instances is computed in a manner analogous to the *FirstWriter*, except considering read access functions.

Compile-time analysis of the transformed program

We present the notation used in the algorithm in a manner closer to the AST form that occurs in program analysis. Each notation that follows can be associated to the notation used in the previous section for the various functions. For example $\vec{w}$ in A is `A[wv]` .

Notations An array location is denoted by the array label and index vector (e.g., `A[idxv]`), where `idxv` is a tuple of length equal to the dimensionality of array `A`.

A statement instance is denoted by the statement label and the iteration vector (e.g., `S<itrv>`).

Arrays are indexed using array reference functions. Given an iteration vector as input, an array reference function returns the index vector to index into the array. The array reference function to compute the array index written by statement `S` is denoted by `S.w()` . The array index written by a statement instance `S<itrv>` is computed as `S.w(itrv)` .

The list of array reference functions to compute indices used in read array references in statement `S` is denoted by `S.r` . The array indices read by a statement instance `S<itrv>` is computed as `S.r[0](itrv)` , `S.r[1](itrv)` , etc.

The verification algorithm employs shadow state for every array location of interest. The shadow state associated with location `A[idxv]` is denoted by `shadow(A[idxv])` .

For every operation `o` in the transformed program, the arrays written and read by it are noted by `o.Aw` and the list `o.Ar` , respectively. The array indices for the writes and reads are denoted by `o.wv` and `o.rv()` .

Verification algorithm First, a simple static analysis of the transformed progam is performed to identify each statement that can write to any array written by the input program.

Each operation performed by these statements is marked to generate runtime check operations. We note `ts(o)` as the statement in the transformed program that generates this operation `o`. The function `CandidateInputStmts(ts)` returns the set of all statements in the input program whose syntactic structure matches that of `ts`. We then analyze and instrument all code reachable from the transformed program segment to be analyzed.

Figure 2.4 depicts the runtime checking algorithm. For simplicity, we represent function arguments as an object that contains all of the necessary parameters to evaluate the functions defined in Sec. 2.4.1. The *WriterBeforeRead* is noted `RAW`.

2.4.2 Algorithm B: A Version-Number based Algorithm

The algorithm in Figure 2.4 enables the verification of program transformations made on arbitrary input affine programs. This generality comes at a runtime overhead cost. For example, functions which need to be evaluated for each operation, especially *NextWriter* and `RAW`, have been generated at compile time as a solution to a parametric lexicographic minimum/maximum on a union of sets. In other words, possibly many if conditionals necessarily need to be evaluated for each instance. Also, although the space overhead remains linear in the input dataset size, storing multidimensional instance vectors in programs nested by d loops requires $2d+1$ integer attributes per memory location accessed in the input program. To overcome these limitations, we present a slightly different technique that requires storing only a single integer per memory location and, in most cases, does not require the evaluation of deep conditionals for each operation. We achieve this by restricting the class of affine programs handled to those where *the iteration vector of an operation can be computed by solving linear equations using the index value of the memory locations*

```
@initialize:
  //executed before starting execution of transformed program
  for w in WriteSet:
    shadow(w) = NIL
  for i in InputSet:
    shadow(i) = Init<0>

def input_statement_instance(o):
  assert shadow(o.Aw[o.wv]) exists
  if shadow(o.Aw[o.wv]) == NIL:
    S<itrv> = FirstWriter(o.Aw[o.wv])
  else:
    S<itrv> = NextWriter(shadow(o.Aw[o.xv]),o.Aw[o.wv])
  assert S<itrv> exists  //valid iteration
  assert S in CandidateInputStmts(ts(o))
  return S<itrv>

@verify(o):
  //executed for every operation encountered in the transformed
    program
  S<itrv> = input_statement_instance(o)
  for i in 0 .. |S.r|-1:
    assert S.r[i](itrv) == o.rv[i]
    assert RAW(S,i)(itrv) == shadow(o.Ar[i][o.rv[i]])
  shadow(o.Aw[o.wv]) = S<itrv>

@terminate:
  //executed after execution the transformed program
  for w in WriteSet:
    assert shadow(w) == LastWriter(w)
  report SUCCESS
```

Figure 2.4: Runtime procedure to check every operation encountered in the transformed program for algorithm A.

accessed by the operation. We also do not store the last instance that have written in this location, but instead the *number of instances which previously wrote to this location.*

Compile-time analysis of the input program The analysis extracts the `WriteSet` and `InputSet` similarly to the general approach, as well as `LB(S)` , the iteration domain of the statement S (i.e., I_S).

A VersionNumber(S) function is built such that, given a instance S<itrv> of statement S, it computes how many instances previously wrote to this location in the input program and returns this value+1. This is computed by forming the set of such an instance using similar concepts as the general algorithm and building a counting polynomial for the resulting parametric set. We use Barvinok's techique [128] available in ISCC. The function RAWv(S,i) returns the VersionNumber of RAW(S,i) .

The function NumWrites(w) returns the number of statement instances that write to array location w in the input program by building the counting polynomial on WriteSet(w) .

The function LIN(S) produces a list of linear functions that consists of:

- Affine array reference functions S.w and each function in S.r
- For all i: RAWv(S,i) if RAWv(S,i) is affine
- VersionNumber(S) if VersionNumber(S) is affine.

We ensure that the matrix associated with LIN(S) is full rank (rank equal to loop nest depth of statement S) for all S. If not, the version-number based scheme cannot be used. Instead, the algorithm shown in Figure 2.4 must be employed.

Finally, the function RHS(S) returns a vector of loop indices, expressed in terms of array indices and version numbers, that forms the right-hand side of the equation to solve for x in LIN(S).xv = RHS(S) .

Verification algorithm The compile-time analysis of the transformed program is performed in a manner identical to the general algorithm. Figure 2.5 gives the pseudocode that instruments the transformed program for this scheme.

```
1  @initialize:
2    //executed before starting execution of transformed program
3    for w in WriteSet:
4      shadow(w) = 0
5    for i in InputSet:
6      shadow(i) = 0
7
8  def input_statement_instance(o):
9    assert shadow(o.Aw[o.wv]) exists #writing to a location in
       WriteSet
10   for S in CandidateInputStmts(ts(o)):
11     Solve for x in LIN(S) . xv = RHS(S)(o.w, o.rv[1], ..) s.t. xv
       in LB(S)
12     if xv is found:
13       //check below is redundant if VersionNumber (S) is a linear
       function
14       if VersionNumber(S)(xv)==shadow(o.Aw[o.wv]):
15         return S<xv>
16   assert false //does not match any input statement instance
17
18 @verify(o):
19   //executed for every operation encountered in the transformed
       program
20   S<itr> = input_statement_instance(o)
21   for i in 0 .. |S.r|-1:
22     if RAWv(S,i) not in LIN(S): //non-linear component
23       assert shadow(o.Ar[i][o.rv[i]]) exists
24       assert RAWv(S,i)(itr) == shadow(o.Ar[i][o.rv[i]])
25   shadow(o.Aw[o.wv]) += 1
26
27 @terminate:
28   //executed after execution the transformed program
29   for w in WriteSet:
30     assert shadow(w) == NumWrites(w)
31   report SUCCESS
```

Figure 2.5: Runtime procedure to check every operation encountered in the transformed program for algorithm B.

Theorem 1. *Algorithm A (resp. B) terminates without error if and only if the operations that were verified in the transformed program execution correspond to a dependence-preserving reordering of the statement instances in the input program.*

Proof. Proof is presented in [19]. □

```
1  seidel_rec(t,ilo,ihi,jlo,jhi,T,N,A[N][N]):
2    if ilo>ihi || jlo>jhi:    return
3    if ilo==ihi && jlo==jhi:
4      A[ilo,jlo] = A[ilo-1,jlo] + A[ilo,jlo-1]
5    else:
6      seidel_rec(t,ilo,  ⌊(ilo+ihi)/2⌋,
7        jlo, ⌊(jlo+jhi)/2⌋,T,N,A)   // Top - Left
8      seidel_rec(t,ilo,  ⌊(ilo+ihi)/2⌋,
9        ⌊(jlo+jhi)/2⌋+1,jhi,T,N,A)   // Top - Right
10     seidel_rec(t,  ⌊(ilo+ihi)/2⌋+1,ihi,
11     jlo, ⌊(jlo+jhi)/2⌋,T,N,A)   // Bottom - Left
12     seidel_rec(t,  ⌊(ilo+ihi)/2⌋+1,ihi,
13     ⌊(jlo+jhi)/2⌋+1,jhi,T,N,A)   // Bottom - Right
14   if ilo==1 && ihi==N-1 && jlo==1 && jhi==N-1 && t<T:
15     seidel_rec(t+1,ilo,ihi,jlo,jhi,T,N,A)                // Next time
       step
```

Figure 2.6: Recursive implementation of Seidel. Invoked as `seidel_rec(0,1,3,1,3,2,4,A)`.

Table 2.1: FirstWriter to array `A` in iterative Seidel

	0	1	2	3
0	–	–	–	–
1	–	S[0,1,1]	S[0,1,2]	S[0,1,3]
2	–	S[0,2,1]	S[0,2,2]	S[0,2,3]
3	–	S[0,3,1]	S[0,3,2]	S[0,3,3]

2.4.3 Illustrative Example: Seidel

We provide an intuition of the algorithms' operation using the Seidel example. The key idea behind the developed approach (common to both Algorithms A and B) is to perform on-the-fly matching of each operation (executed statement instance) of the transformed program being checked with some statement instance in the execution of the input affine

program. The test program uses a shadow variable for each data variable, and as each operation of the tested program is successfully matched with a statement instance in the input program, the information is stored in the shadow variable for the data element written by the operation. The stored information either directly (Algorithm A) or indirectly (Algorithm B) enables identification of the matched statement instance in the input program's execution. If the on-the-fly matching process succeeds, it essentially identifies a valid dependence-preserving bijection between the input program's statement instances and the sequence of operations executed by the transformed program. In the dynamic matching process, if we are unable to successfully match any operation of the transformed program being checked, it means there is an error in the transformed program: its sequence of operations is provably not equivalent to any dependence-preserving reordering of the input program's statement instances.

We use an example to illustrate the verification approach. Figure 2.2 shows code for a 2D iterative stencil computation. The code sweeps through a 2D array `A`, row by row, updating element `A[i][j]` using the values of two neighboring elements, `A[i-1][j]` and `A[i][j-1]` (inner loops over `i,j`). The sweep over the 2D array is repeated for multiple time steps (outermost `t` loop). Figure 2.6 shows recursive code that performs the same computation. It splits the spatial domain of the sweep into four quadrants and recursively invokes sweeps on them, in the order top-left, top-right, bottom-left, and bottom-right. The base case performs the stencil operation on a single element.

Table 2.2 shows the sequence of 18 statement instances for the execution of the iterative Seidel code for `N=4,T=2` . For each statement instance, we see the written element of A, the two read elements of A, along with some additional information about data dependences that can be computed in a straightforward way in a polyhedral compiler framework for any

affine program: 1) the latest preceding statement instance (PrevW) that wrote to any given data element, 2) the earliest future statement instance (NextW) that will write to any given data element. Shadow variables for input and output data elements before any assignment are both shown as null ('–'). Consider, for example, the two statement instances that write to `A[2][1]` : `S<0,2,1>` and `S<1,2,1>` . In each of the two statement instances, the read data elements are `A[1][1]` and `A[2][0]` . Of these two, `A[2][0]` is a boundary element that is not written within the execution of the code. Therefore, the "previous writer" is null. However, `A[1][1]` is modified in the code by statement instances `S<0,1,1>` and `S<1,1,1>` . The table row for statement instance `S<0,2,1>` lists `S<0,1,1>` as the previous writer of the read data element `A[1][1]` , while the row for `S<1,2,1>` lists `S<1,1,1>` as the previous writer of `A[1][1]` . The next writer for the written data element `A[2][1]` is `S<1,2,1>` for statement instance `S<0,2,1>` and null for statement instance `S<1,2,1>` . Table 2.1 displays the first statement instance that writes into each element of array `A`. The top and left boundary elements are only read but not written, so they have null as their "first writer."

Table 2.3 shows the sequence of 18 operations executed by the checker for the recursive Seidel code. The table entries include the read and written elements of A for each instance, along with the values of the shadow variables associated with each data element. The shadow variables are all initialized to null. Consider the first operation executed by the recursive version: it writes into `A[1][1]` , whose shadow is currently null. The first writer of `A[1][1]` in the input program is determined (shown in Table 2.1, but actually dynamically generated, as shown later) to be `S<0,1,1>` . Thus, as long as the operands and their previous writers also match, this operation can get matched with that statement instance of the input program. The previous writers of the operands `A[0][1]` and `A[1][0]` for `S<0,1,1>` are both null, matching the null `shadow(A[0][1])` and `shadow(A[1][0])` in the checker program for

Table 2.2: Statement instances executed by iterative Seidel and related relations

Stmt	Write	NextW	Read1	PrevW	Read2	PrevW
S<0,1,1>	A[1,1]	S<1,1,1>	A[0,1]	–	A[1,0]	–
S<0,1,2>	A[1,2]	S<1,1,2>	A[0,2]	–	A[1,1]	S<0,1,1>
S<0,1,3>	A[1,3]	S<1,1,3>	A[0,3]	–	A[1,2]	S<0,1,2>
S<0,2,1>	A[2,1]	S<1,2,1>	A[1,1]	S<0,1,1>	A[2,0]	–
S<0,2,2>	A[2,2]	S<1,2,2>	A[1,2]	S<0,1,2>	A[2,1]	S<0,2,1>
S<0,2,3>	A[2,3]	S<1,2,3>	A[1,3]	S<0,1,3>	A[2,2]	S<0,2,2>
S<0,3,1>	A[3,1]	S<1,3,1>	A[2,1]	S<0,2,1>	A[3,0]	–
S<0,3,2>	A[3,2]	S<1,3,2>	A[2,2]	S<0,2,2>	A[3,1]	S<0,3,1>
S<0,3,3>	A[3,3]	S<1,3,3>	A[2,3]	S<0,2,3>	A[3,2]	S<0,3,2>
S<1,1,1>	A[1,1]	–	A[0,1]	–	A[1,0]	–
S<1,1,2>	A[1,2]	–	A[0,2]	–	A[1,1]	S<1,1,1>
S<1,1,3>	A[1,3]	–	A[0,3]	–	A[1,2]	S<1,1,2>
S<1,2,1>	A[2,1]	–	A[1,1]	S<1,1,1>	A[2,0]	–
S<1,2,2>	A[2,2]	–	A[1,2]	S<1,1,2>	A[2,1]	S<1,2,1>
S<1,2,3>	A[2,3]	–	A[1,3]	S<1,1,3>	A[2,2]	S<1,2,2>
S<1,3,1>	A[3,1]	–	A[2,1]	S<1,2,1>	A[3,0]	–
S<1,3,2>	A[3,2]	–	A[2,2]	S<1,2,2>	A[3,1]	S<1,3,1>
S<1,3,3>	A[3,3]	–	A[2,3]	S<1,2,3>	A[3,2]	S<1,3,2>

the recursive version. `Shadow(A[1][1])` is set to the successfully matched input program statement instance `S<0,1,1>` . Next, consider the second operation in the execution of the recursive version that writes into `A[1][1]` . To match this operation to a statement instance of the input program, the shadow variable value `S<0,1,1>` is used, and the next-writer in the input program is determined to be `S<1,1,1>` . The read data elements and their previous writers (null) match the shadow variables in the executing checker program, implying a match. In a similar manner, all operations in the recursive execution can be matched one-to-one with a statement instance of the iterative version.

To illustrate the verification process further, consider what happens when the tested program is not equivalent to the input program. For example, for the recursive Seidel

Table 2.3: Sequence of operations for recursive Seidel; "Sdw" abbreviates shadow in the column headers

Write	Sdw_old	Sdw_new /Match	Read_1	Shadow	Read_2	Shadow
A[1,1]	–	S<0,1,1>	A[0,1]	–	A[1,0]	–
A[1,2]	–	S<0,1,2>	A[0,2]	–	A[1,1]	S<0,1,1>
A[2,1]	–	S<0,2,1>	A[1,1]	S<0,1,1>	A[2,0]	–
A[2,2]	–	S<0,2,2>	A[1,2]	S<0,1,2>	A[2,1]	S<0,2,1>
A[1,3]	–	S<0,1,3>	A[0,3]	–	A[1,2]	S<0,1,2>
A[2,3]	–	S<0,2,3>	A[1,3]	S<0,1,3>	A[2,2]	S<0,2,2>
A[3,1]	–	S<0,3,1>	A[2,1]	S<0,2,1>	A[3,0]	–
A[3,2]	–	S<0,3,2>	A[2,2]	S<0,2,2>	A[3,1]	S<0,3,1>
A[3,3]	–	S<0,3,3>	A[2,3]	S<0,2,3>	A[3,2]	S<0,3,2>
A[1,1]	S<0,1,1>	S<1,1,1>	A[0,1]	–	A[1,0]	–
A[1,2]	S<0,1,2>	S<1,1,2>	A[0,2]	–	A[1,1]	S<1,1,1>
A[2,1]	S<0,2,1>	S<1,2,1>	A[1,1]	S<1,1,1>	A[2,0]	–
A[2,2]	S<0,2,2>	S<1,2,2>	A[1,2]	S<1,1,2>	A[2,1]	S<1,2,1>
A[1,3]	S<0,1,3>	S<1,1,3>	A[0,3]	–	A[1,2]	S<1,1,2>
A[2,3]	S<0,2,3>	S<1,2,3>	A[1,3]	S<1,1,3>	A[2,2]	S<1,2,2>
A[3,1]	S<0,3,1>	S<1,3,1>	A[2,1]	S<1,2,1>	A[3,0]	–
A[3,2]	S<0,3,2>	S<1,3,2>	A[2,2]	S<1,2,2>	A[3,1]	S<1,3,1>
A[3,3]	S<0,3,3>	S<1,3,3>	A[2,3]	S<1,2,3>	A[3,2]	S<1,3,2>

version, assume that the last two of the four recursive calls to top-left, top-right, bottom-left, and bottom-right got swapped by mistake, i.e., the sequence of calls instead becomes top-left, top-right, *bottom-right, bottom-left*. Table 2.4 shows the first few operations executed, along with shadow variable information. The on-the-fly matching is successful for the first two operations (writing into `A[1][1]` and `A[1][2]`, respectively) but will fail for the third operation, which writes into `A[2][2]`. Because the shadow variable has a null value, the matching process will use first-writer information for `A[2][2]` and attempt a match with the iterative program's statement instance `S<0,2,2>`. Of the two read data elements, we get a match for `A[1][2]` (the previous writer in the iterative program statement instance,

Table 2.4: Sample operations for recursive Seidel with error; "Sdw" abbreviates shadow in the column headers

Write	Sdw_old	Sdw_new /Match	Read_1	Shadow	Read_2	Shadow
A[1,1]	–	S<0,1,1>	A[0,1]	–	A[1,0]	–
A[1,2]	–	S<0,1,2>	A[0,2]	–	A[1,1]	S<0,1,1>
A[2,2]	–	S<0,2,2>	A[1,2]	S<0,1,2>	A[2,1]	–
A[2,1]	–	S<0,2,1>	A[1,1]	S<0,1,1>	A[2,0]	–

`S<0,1,2>` , matches the shadow variable in the test program) but not for A[2][1]. The shadow variable for `A[2][1]` is null as `A[2][1]` has not yet been written in the test program so far. However, the previous writer for `A[2][1]` in the matched statement instance is `S<0,2,1>` . This mismatch means the test program cannot be equivalent to the input program.

Figures 2.7 and 2.8 show the checker code generated for recursive Seidel code for using Algorithms A and B, respectively. Figure 2.7 first shows initialization code that sets all shadow variables to null followed by the verification code inserted after the statement in line 4 of the recursive Seidel code (Figure 2.6)—the operation performed at the base case of the recursion. The checker code encodes the previously described matching process using the example traces. Finally, an epilog code verifies that all operations were performed by the transformed program. This is done by ensuring the final shadow variables match the analytically computable last writers for the input affine program.

Figure 2.8 shows the checker code for using the optimized approach of Algorithm B, which is applicable for this example. Instead of keeping full details of matched statement instances in shadow variables, a compact version counter is stored in each shadow variable, and the matching process involves comparing the values in the version counters with analytically computable write counts for each data element in the affine input program.

```
1  @initialize: // initialize  shadow  variables
2    for (i,j)  in (0,0) to (N-1,N-1):
3      shadow(A[i][j]) =   NIL
4  @verify: // inserted  after  line  4 in  seidel_rec
5    i = ilo
6    j = jlo
7    assert i-1==ilo-1   // check  1st  index  and
8    assert j==jlo       // 2nd  index  in  A[ilo -1][ jlo ]
9    assert i==ilo       // check  1st  index  and
10   assert j-1==jlo-1   // 2nd  index  in  A[ilo ][ jlo -1]
11   if shadow(A[i][j])==  NIL: // Write
12     assert FirstWriter(A[i][j])==S<t,i,j>
13   else:
14     assert shadow(A[i][j])==S<t-1,i,j>
15   if shadow(A[i-1][j])!=  NIL: // Read
16     assert shadow(A[i-1][j])==S<t,i-1,j>
17   if shadow(A[i][j-1])!=  NIL: // Read
18     assert shadow(A[i][j-1])==S<t,i,j-1>
19   shadow(A[i][j]) = S<t,i,j>
20 @terminate: // epilog  to  ensure  completeness
21   for (i,j)  in (1,1) to (N-1,N-1):
22     assert shadow(A[i][j]) == S<T-1,i,j>
```

Figure 2.7: The checker code inserted by Algorithm A after line 4 in the recursive Seidel implementation (Figure 2.6). Note that the compiler can optimize away the first four assert statements.

```
1  @initialize: // initialize  shadow  variables
2    for (i,j)  in (0,0) to (N-1,N-1):
3      shadow(A[i][j]) = 0
4  @verify: // inserted  after  line  4 in seidel_rec
5    Determine (t,i,j) that satisfy the following inequalities        and
       equalities:
6      0<=t<T  and 0<j<N  and 0<j<N  // loop  bounds
7      i-1=ilo-1  // check  1st  index  and
8      j  =jlo    // 2nd  index  in  A[ilo-1][ jlo ]
9      i  =ilo    // check  1st  index  and
10     j-1=jlo-1  // 2nd  index  in  A[ilo ][ jlo -1]
11     // shadow  values
12     shadow(A[i][j])  =(i>0 && j>0) ? t    : 0
13     shadow(A[i-1][j])=(i>1 && j>0) ? t+1  : 0
14     shadow(A[i][j-1])=(i>0 && j>1) ? t+1  : 0
15   if no (t,i,j) found: report error
16   shadow(A[i][j]) += 1
17 @terminate: // epilog  to  ensure  completeness
18   for (i,j)  in (1,1) to (N-1,N-1):
19     assert shadow(A[i][j]) == T
```

Figure 2.8: The checker code inserted by Algorithm B after line 4 in the recursive Seidel implementation (Figure 2.6).

2.4.4 Time and Space Complexity

For each operation to be checked by the transformed program, the number of actions taken by the checker is proportional to the number of array references and loop nesting depth of each statement. Specifically, computing the corresponding statement instance requires time proportional to the loop nesting depth. Checking each array reference requires computing the array index from an iterator value with the cost proportional to that of computing the array references in the input program. Note that $\texttt{LIN(S)}$ is known when analyzing the input affine program. Rather than solve the system of equations at runtime, we compute the (pseudo-)inverse for $\texttt{LIN(S)}$ at compile time and only perform a matrix-vector product at runtime. In general, the cost of checking each operation using Algorithm A is proportional to the loop nesting depth. Given the loop nesting depth is usually small, the checker

cost is on the same order as executing the input program. Algorithm B might undertake several mappings for each operation—in the worst case, attempting a mapping with every statement in the input program. Therefore, the worst case cost of checking each operation using Algorithm B is proportional to the loop nesting depth times the number of statements in the input program. We only employ Algorithm B when the number of candidate input statements for any given statement in the transformed program is small.

The operations in the transformed program are processed in a streaming fashion as they are generated with no storage of past operations. The only significant space required by the checker is the shadow variable associated with each location read and written by the input program. Each shadow variable holds a constant-sized object (statement instance object or version number). Therefore, the algorithm's total space complexity is the same as that of the input affine program, and is independent of the number of statement instances executed by the input or the transformed program.

2.5 Scope of Applicability, Enhancements, and Limitations

The following describes a few key performance enhancements for the PolyCheck implementation.

Optimized checking of full tiles As discussed, we add several instructions for each statement in the transformed program. This overhead can be significantly improved for a common class of transformed programs. Specifically, tiled programs group statement instances into tiles to improve data locality. While the bounds for these tiles can be constants or influenced by a tile-size parameter, the code within a tile itself is often an affine program. In such scenarios, we employ index set splitting to isolate tiles that depend only on linear combinations of problem- and tile-size parameters. We outline such tiles and statically

analyze them to identify the incoming definitions—array locations last written elsewhere in the program and read within this tile. Statements with these definitions are verified in full. Other statements that only read array locations written by statement instances within the same tile (intra-tile dependences) are statically verified. Each statement is replaced with an increment of the version number of the array element written by that statement. This ensures that version counts are still maintained to enable continued verification of the remaining program.

Delayed detection and vectorization If the termination and error reporting criteria can be relaxed, the verification performance can be improved. Checking and aborting on each verification introduces several branches and leads to poor performance. Most checks compare the expected version number for the input of a dependence with the version number observed. Thus, we replace the check:

`assert a==b` with `checksum |= (a^b)`

At the end of the program, we verify the checksum:

```
assert checksum==0
```

At the end of the iteration, a non-zero checksum value indicates detection of an error in the transformed program. The bitwise or operation ensures that any error detected (using the xor operation) results in a bit being set and never subsequently unset. Therefore, the checksum is exact in that any error detected during the checker execution is eventually reported. When the checks are enclosed in loops with statically known loop bounds in the transformed code, we place the scalar checksum variable with an array to make the code amenable to vectorization.

Localizing bugs In addition to detecting errors, PolyCheck strives to isolate the offending statement instance and report the reason for failed verification. For each offending operation, we report the operation and information on the statement instance of the transformed program. When a given operation cannot be mapped to a dependence-preserving statement instance s in the input program that writes to the same location, we attempt to find alternative possible mappings for the same operation to other non-dependence-preserving statement instances. If the operation maps to a statement instance s' that lexicographically precedes s, we report the offending operation as a duplicate of an earlier operation generated by s. If the operation maps to a statement instance s' that lexicographically follows s, we report the offending operation as having executed too soon, violating a write-after-write dependence. When no valid mapping can be found for any version number, we report it as an invalid operation. When an operation is found to violate a dependence, we report the observed and anticipated statement instances in the input program. This allows the programmer to locate the offending statement instance in the transformed program, the corresponding statements in the input program, and the cause for the failed verification.

Problem-size parameters When the array access functions depend on problem-size parameters, information about them must be extracted from the transformed program as well. We support two approaches. First, the user can provide the problem size as a simple annotation to the verifier. This is expected to be one or a small number of annotations, one per parameter, and is not an onerous requirement. Second, we can treat the parameters as variables and solve for them as we solve for the loop indices. Assuming that there are sufficient linearly independent constraints involving the parameter of interest, we compute the parameters the first time they are encountered. In subsequent checks, the discovered

parameters are treated as constants. In general, problem-size parameters can be discovered by relating properties of a full polyhedron—number of integer points, last write to a given location, extremes on the array locations read/written, etc.—and relating them to the values observed when running the transformed program.

While PolyCheck can handle numerous program variants, it is still restricted to iteration reordering transformations. The scope of transformations and programs that PolyCheck can handle is discussed further in the following.

Aliasing Affine dependence analysis assumes that two locations `A1[itr1v]` and `A2[itr2v]` are identical (refer to the same location) only if `A1=A2` and `itr1v=itr2v` . This imposes a restriction on the input program's analysis. Therefore, PolyCheck cannot be employed when aliasing of arrays might occur.

Parallelization As described, PolyCheck assumes a sequential input program and verifies transformations that maintain dependences in this sequential order. However, if the transformed code is a parallel data-race-free program, the checker augmented program is also data-race-free and can be executed in parallel. Specifically, where the transformed program accesses an array location, the checker accesses the location's shadow. Verifying a parallel program then translates to verifying the program meets PolyCheck's check and is data-race free. Note that data-race freedom might be schedule-specific [50, 108] or schedule-independent [27, 101]. PolyCheck's verification is only valid for the schedules shown to be data-race free.

Data space transformations PolyCheck relies on the fact that iteration reordering transformations still maintain the operation order with respect to the data space. Data space

transformations that change the array index expressions (e.g., row/column-major to blocked or recursive storage) can lead to valid corresponding statement instances being flagged as in error. However, data space transformations that do not alter the array index expressions (e.g., padding) do not interfere with the checker.

Scalar transformations The current design of PolyCheck cannot handle scalar and basic-block-level transformations. However, some transformations are more difficult than others. Transformations such as common sub-expression elimination and register tiling potentially can be handled by tracking the operation tree representing each assignment. The operation tree represents the set of expressions on array locations in the `InputSet` and `WriteSet` that produce the current value. When a value is assigned to an element in the `WriteSet`, the entire operation tree is checked. Conversely, some transformations, such as constant propagation, change the operations sufficiently to interfere with the construction of the bijection just by tracking their data space effects.

2.6 Experimental Evaluation

Implementation details PolyCheck first analyzes an input affine program to compute the check codes. This stage was implemented using ISL-0.12 [5] (with barvinok-0.36 and pet-0.04), an integer set library for the polyhedral model, and using the LLVM/Clang-3.4 compiler infrastructure [6]. ISL [124] performs dependence analysis of the source program to generate dependency relations, and the algorithms described in previous sections are implemented to calculate the correct checker for each dependency relation. Then PolyCheck inserts the checker codes for each target statement by analyzing the transformed program to check then performs code generation and outputs the checker-inserted program to enable equivalence checking.

2.6.1 Evaluation Using the PoCC Polyhedral Compiler

The following is a detailed evaluation of PolyCheck, discussing the coverage and checking of several compiler optimization passes from the PoCC polyhedral research compiler [8].

Benchmarks PolyCheck currently requires the input program to have a static control-flow and use only affine expressions of surrounding loop iterators and program parameters in the loop bounds and array expressions. The PolyBench/C test suite gathers a collection of programs meeting these requirements [9]. Table 2.5 displays the various computation codes that were evaluated. We used the large dataset size provided.

Table 2.5: Benchmarks in evaluation using PoCC

Benchmark	Description
gemm	Matrix-multiply C=alpha.A.B+beta.C
gemver	Vector Multiplication and Matrix Addition
lu	LU decomposition
bicg	BiCG Sub Kernel of BiCGStab Linear Solver
correlation	Correlation matrix Computation
covariance	Covariance matrix Computation
jacobi-2d-imper	2D Jacobi stencil computation
seidel-2d	2D Seidel stencil computation
fdtd-2d	2D Finite Different Time Domain Kernel
reg_detect	Edge detection
doitgen	Multiresolution analysis kernel (MADNESS)

Tools and setup All benchmark transformations used in evaluation are performed by PoCC-1.2 [8]. To perform comparison experiments with state of the art, we consider the

integer set analysis framework (ISA-0.13) [4], a static equivalence checking tool for affine programs using a widening approach.

All running time experiments are performed on Intel Xeon E5-2650 processors at frequency 2.60 GHz with 32 KB L1 cache. The programs are all compiled with GCC-4.8.1 compiler with -O3 optimization. The reported running time is the average of five runs.

Compiler Passes Considered

A polyhedral compiler contains numerous passes, including extracting the polyhedral representation (scop extraction), performing array data-flow analysis (dependence computation), finding an optimized schedule for the program operations (scheduling), and implementing the new schedule as a new C program (code generation). Each of these stages involves several external libraries and possibly tens of thousands of lines of codes. In addition, for better performance numerous complementary AST-based transformations may be performed after code generation, such as unrolling/register-tiling, loop bound optimization, or full tile separation. PoCC [8] implements each of these, and we selected several critical pass combinations to demonstrate our approach.

passthru This configuration performs only SCoP extraction (using the Clan [2] pass), converts the program to its internal polyhedral representation (ScopLib), does not perform any optimization, and generates back a loop-based code implementing the original schedule (using the CLooG [25] pass) that is converted to PoCC's AST representation (PAST) and pretty-printed to a C program. The generated code is an affine program.

data locality This configuration includes all stages of passthru, adding polyhedral data dependence analysis (using the Candl [47] pass) and computation of an optimized schedule

of operations to improve temporal data locality and coarse-grain parallelization (using the Pluto algorithm [34]). The generated code is an affine program.

fixed tiling This configuration includes all stages of data locality, adding a modification of the polyhedral representation of statements to implement iteration tiling (a.k.a. loop blocking) whenever possible (using the Pluto [34] pass). Tile sizes are constants provided to the compiler, and the code generation tool attempts to optimize the code structure using the tile size information. The generated code is an affine program.

AST-based unrolling This configuration includes all stages of fixed tiling and unrolls all innermost loops by four, generating an epilog loop to cope with parametric loop bounds if they are not a perfect multiplier of the unroll factor. It stresses the PAST optimizers in PoCC—the AST-based backend optimizers. While the generated code is semantically an affine program, the generated code structure can challenge SCoP extractors in recovering the entire program as a single affine region.

AST-based bound optimization This configuration includes all stages of fixed tiling and performs aggressive loop bound hoisting and simplification, replacing the cascading min/-max expressions generated by CLooG for conjunctions of inequalities by a tree of evaluation [136]. While the generated code is still semantically an affine program, recovering the loop bounds expressions roughly requires reverse engineering the optimizations in the SCoP extractor.

parametric tiling This configuration includes all stages of data locality, adding the generation of iteration tiling using parametric tile sizes (using the Ptile [24] pass). That is,

the generated code supports arbitrary tile sizes and selecting the tile size at runtime. Because non-affine expressions involving iterators multiplied with parameters (tile sizes) are needed, the generated code is not an affine program,

full tile separation This configuration includes all stages of parametric tiling, adding an AST-based post-processing to separate partial tiles from full tiles in the code structure (using the Ptile [24] pass). The generated code is not an affine program.

Results Overview

Table 2.6 compares the coverage of PolyCheck and ISA [127], the state-of-the-art equivalence checker tool for affine programs. Logically ISA cannot handle transformed programs that are not affine, such as the parametric tiling cases. ISA is not applicable (N/A) for those. ISA uses the Polyhedral Extraction Tool (PET) to automatically detect affine regions, implementing a powerful SCoP extractor on top of Clang. However, both the AST-based transformations from PoCC evaluated were not handled by the current implementation of the tool (reported as "not supported", N/S). While we believe it is possible to improve the implementation of ISA's SCoP extractor to handle these cases as the generated code still is semantically an affine program, this behavior reinforces the need for a verification tool that is agnostic to how the code has been transformed and generated. PolyCheck fulfills this role and for all optimizations tested can successfully verify equivalence (for correct optimizations) and find bugs (for bugs randomly inserted in the generated code).

We have verified the optimizations applied using PoCC for all of the benchmarks listed in Table 2.5. For all cases where ISA could be applied, both PolyCheck and ISA provide the same positive answer regarding the equivalence between the original and transformed

Table 2.6: Summary of tool coverage (PoCC evaluation)

Optimization	PolyCheck	ISA 0.13
passthru	√	√
data locality	√	√
fixed tiling	√	√
AST-based unrolling	√	N/S
AST-based bound optimization	√	N/S
parametric tiling	√	N/A
full tile separation	√	N/A

codes. For all other cases where only PolyCheck could be applied, it also reported equivalence between the original and transformed codes. In other words, we have shown the absence of iteration reordering bugs for these test cases in the generated codes.

Finding Bugs in the Generated Codes

To further demonstrate the power of our approach, we emulated bugs in the software by randomly introducing problems in the transformed code (shown in Table 2.7). Often, bugs translate into an immediate effect of memory corruption (segmentation fault), which usually has visible effects to the user. PoCC already implements a checker that encapsulates all memory references in a wrapper function to detect out-of-bound memory accesses. Here, we assume such a test has already passed. We designed pernicious bugs that do not lead to segmentation fault and that could be easily missed by classical testing procedures that check the bit-by-bit equivalence of the produced outputs by the reference and transformed versions. Indeed, in this case, an open problem is to find datasets that will trigger visible differences in the produced output. In contrast, our approach does not require any reasoning on the input data because we track the satisfaction of data dependences and not the result of the computation performed.

Table 2.7: Summary of bugs tested (PoCC evaluation)

Bug	Description
loop bound	decrease by 1 some loop upper bound
array access	divide by 2 some array subscript
permutation	interchange two non-permutable loops
code motion	move around some loop nest

A bug has been introduced randomly in each transformed program, for all benchmarks, for both loop bounds and array accesses. For the permutation and code motion, we manually modified the transformed program to introduce a change of semantics when applicable. For some benchmarks, all loop permutations are valid (e.g., dgemm), so no loop permutation bug can be created.

Discussions

Verifying polyhedral transformations Loop transformations, especially iteration reordering ones, are not exclusive to polyhedral compilers. Most compilers implementing some loop transformations will support loop unrolling, interchange, fusion, distribution, and tiling (when possible). When the generated code is a purely affine program, both PolyCheck and ISA can successfully determine the correctness of the transformed program, given the input one. However, the execution time of the two techniques can differ vastly. For ISA, it can prove equivalence extremely quickly, especially on simple programs—in much less than a second. On the other hand, it is at the mercy of polyhedral analysis complexity, which is NP-complete in general. That is, some particular codes could end up taking hours or GB of space to check [127]. In contrast, our approach has a sort of predictable time to terminate: on the order of the time needed for the transformed code to run

on a machine, irrespective of the transformation complexity.[3] Still, this time depends on the problem size used.

By definition, polyhedral programs considered have a static control-flow, so it is not necessary to test various datasets of the same size to conclude equivalence with PolyCheck for the scope of bugs it can find. This is in contrast to classical testing, where the output produced by the transformed code is compared bit by bit to the output of a gold, original version. In this case, even for polyhedral programs, the dataset has a fundamental impact on the output values (think about testing dgemm by multiplying 0-filled matrices). We claim PolyCheck is suited to accelerate and even replace such testing procedures because of its insensitivity to dataset values.

Verifying polyhedral and AST transformations It is widely accepted that affine iteration reordering transformations alone are not enough to achieve the best performance. That is, practical polyhedral compilers must complement affine transformations with AST-based transformations for best performance. Shirako et al. recently showed an example of such integration in PoCC [110]. Our approach is robust to complementary transformations implemented at the AST level, as shown with the unrolling experiments. One interesting observation was made in regard to ISA not being able to handle the unrolled code. While the code is still semantically affine, the way it was syntactically generated challenged the ISA SCoP extractor. Clearly, this is not indicative of a limitation of the ISA algorithm/method, merely one of a tool to extract polyhedral representation. Nevertheless, it shows the advantage in having a framework that operates independent of how the code is actually generated. Notably, PolyCheck will not require any update or change for any new optimization applied on affine codes.

[3] Assuming there is no infinite loop in the generated program.

Verifying non-affine transformations Non-affine transformations of affine programs must be properly anticipated for. Existing optimizations, such as parametric tiling, already generate non-affine code and cannot be checked by tools such as ISA. PolyCheck seamlessly handles such codes. The optimization on loop bounds has proven to be another challenge for SCoP extraction in PET. However, this time, the optimization's complexity makes it nearly impossible to create a canonical affine representation of the code at SCoP extraction time without actually reverse-engineering the optimization. To that extent, it can almost be considered as a non-affine transformation, yet there is a compelling need to verify its correctness. Going further, many other code transformations, such as recursive decomposition for cache-oblivious algorithms, lead to non-affine programs, even if the input is an affine program. We believe PolyCheck perfectly complements tools like ISA by enabling seamless support of such transformations, and it offers a more practical alternative to trace-based checking (due to its much lower space complexity) or output difference checking (based on its insensitivity to the dataset content).

2.6.2 Evaluation Using Cilk and the Pochoir Stencil Compiler

We further evaluate PolyCheck by verifying the correctness of Cilk and Pochoir implementations of affine programs.

Benchmarks and Setup Cilk [31, 32] is a programming model that supports fork/join parallelism based on random work stealing [33]. Cilk programs recursively divide (fork) the computation into sub-computations and combine their result (join) to produce the final result. Cilk is a simple extension of the base C/C++ language with the serial elision property: removing the Cilk keywords results in a sequential recursive program. We evaluated the correctness of the single-threaded execution of the recursive Cilk implementations of

affine programs, distributed as part of MIT Cilk 5.4.6 [7]. For each benchmark, we wrote the loop version as the input program specification to verify the Cilk implementation. Then, the checker code was inserted into the Cilk implementation at each statement that generates an operation to be checked.

Pochoir [118, 117] is a embedded domain-specific language for stencil computations. The programmer specifies the computation in terms of a grid and the statements to compute the value of a grid point in a multidimensional spatial grid at time t as a function of neighboring grid points at times before t. Such a specification is compiled into C++ by the Pochoir compiler to generate divide-and-conquer stencil codes [55, 56] based on cache-oblivious algorithms [54, 97]. Note that the iterations or loop space are completely implicit in the program and immediately not available to the programmer. Each benchmark in the Pochoir distribution (version 0.5) includes a reference loop implementation used for testing. We engaged these implementations as the input source program. The computation to be performed at each grid point is a set of statements. We inserted a checker code to verify each instance of these statements directly in the Pochoir program. The program is then compiled through Pochoir and executed to verify the Pochoir compiler's transformations.

Table 2.8: Errors found in transformations by PolyOpt/C 0.2.0

Benchmark	Original Program (Pass)	Transformed with tile_8_1_1(have errors)	
cholesky	A[1][0](1)=x(2)*p[0](1) assert(x(2)==2)	A[1][0](1)=x(1)*p[0](1) assert(x(1)==2)	Fail
reg_detect	mean[0][0](1)=sum_diff[0][0][15](1) assert(sum_diff[0][0][15](1)==1)	mean[0][0](1)=sum_diff[0][0][15](0) assert(sum_diff[0][0][15](0)==0)	Fail

Table 2.9 shows the Cilk and Pochoir benchmarks used in the evaluation. The default dataset size was used for all benchmarks. All benchmarks were compiled with the ICC-15.0.3 compiler with -O3 optimization.

Table 2.9: Cilk and Pochoir benchmarks

Benchmarks		Description
Cilk	matmul	Matrix-multiply C=A.B
	rectmul	Multiply two rectangualar matrizes
	spacemul	A dag-consistent Matrix Multiply
	heat	Heat diffusion
	lu	Martix LU decomposition
Pochoir	heat_2D	heat equation on 2D grid
	heat_2P	heat equation on 2D torus
	apop	American put stock option pricing
	3d7pt	order-1 3D 7 point stencil
	3d27pt	order-1 3D 27-point stencil

Evaluation and results The evaluation of our approach on Cilk and Pochoir programs is similar to Polybench/C test suite. To emulate bugs, we randomly introduce problems to programs (shown in Table 2.10). As in the PoCC study, we ensure the bugs introduced do not lead to segmentation violation.

Table 2.10: Summary of evaluation using Cilk and Pochoir benchmarks

Evaluation		Description	Detect
Cilk	no bug	original Cilk program	Pass
	loop bound	decrease some loop upper bound	$\surd$
	array access	decrease some array subscript	$\surd$
Pochoir	no bug	original Pochoir program	Pass
	loop domain	decrease some loop domain	$\surd$
	array access	decrease some array subscript	$\surd$

All bugs are introduced randomly for all transformed programs, loop bound, domain, and array access. Of note, there are no explicit loops in the Pochoir program, but macro functions specify the loop domain. Table 2.10 shows the results.

2.6.3 Identifying Bugs in PolyOpt/C

CodeThorn [109] identified two bugs in PolyOpt/C 0.2.0 [95] polyhedral compiler. These bug results in incorrect code generated for the cholesky and reg_detect benchmarks. CodeThorn checks the transformed program for a given problem size by explicitly enumerating the trace of statement instances and performing a verification. Table 2.8 shows these bugs, illustrating the array element following the version number (in parenthesis). PolyCheck found both bugs efficiently. We also present performance comparisons with CodeThorn below, which showcase how PolyCheck can be orders of magnitude faster than CodeThorn.

2.6.4 PolyCheck Overhead

Runtime overhead We conclude our evaluation with a detailed reporting of the execution time of the transformed programs modified to integrate the checkers. To evaluate the checker-only overhead, we replaced the actual computation with the checker's actions. Each program must be run to completion to provide verification information. Figure 2.9 reports the timing (normalized to the execution time of the transformed program) of the fixed-tiling transformation, without and with checker optimization described earlier. Figure 2.10 shows the same but for the parametric-tiling transformation. We remark that for reg_detect, the checker code is initially slower than the transformed code. This is because our checker code can disrupt SIMD vectorization optimizations, an effect exacerbated for reg_detect. In general, the checker's execution time is proportional to the performance of

Table 2.11: Overhead comparison with CodeThorn [109]

Benchmark	CodeThorn	PolyCheck
covariance	1.56 secs	0.005 ms
covariance-tile-8-1-1	1.71 secs	0.030 ms
fdtd-2d	1.58 secs	0.047 ms
fdtd-2d-tile-8-1-1	3.14 secs	0.071 ms
jacobi-2d-imper	0.692 ms	0.027 ms
jacobi-2d-imper-tile-8-1-1	1.50 secs	0.037 ms
seidel-2d	1.05 secs	0.031 ms
seidel-2d-tile-8-1-1	2.51 secs	0.032 ms

the transformed program because it has almost identical memory traffic. The arithmetic operations performed in the actual computation are replaced by checker instructions, which can possibly represent more workload for the checker version. The checker optimization dramatically reduces any such effect (shown in both Figures 2.9 and 2.10). The -Opt timing is systematically lower than the transformed program, and there are cases, such as bicg and lu, where the overhead becomes marginal. Nevertheless, as we always execute the program, the execution time remains dependent on the problem size. In Figure 2.11, we show the execution time to check fixed- and parametric-tiled versions of LU. It is evident that the time to execute the checkers is significantly lower than the transformed program. This is due to the use of optimized checking of full tiles. Figure 2.12 shows the execution time for reg_detect, where full tile optimization cannot applied.

Overhead Comparison We further evaluate PolyCheck's runtime overhead by presenting a comparison experiment of CodeThorn, a trace-based tool by Schordan et al. [109]. Table 2.11 compares the two tools using benchmarks and problem sizes chosen from [109]. CodeThorn runtimes are directly from Schordan et al. [109], and we report the time of the

optimized runtime checking procedure only for PolyCheck (the static analysis time does not depend on the problem size and needs to be done only once per ***input*** program). PolyCheck can be orders of magnitude faster than codeThorn, thanks to very efficient runtime checking by actual execution of the program, and limited space overhead. However PolyCheck requires the input program to have a static/affine control-flow, while codeThorn can handle programs with data-dependent control-flow.

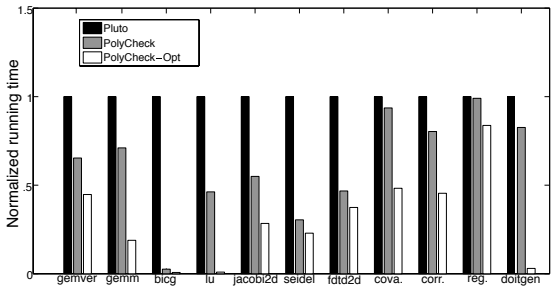

Figure 2.9: Checker running time with fixed tiling.

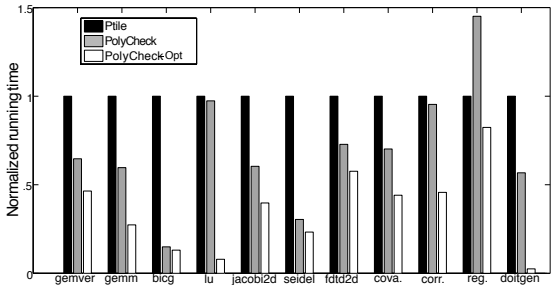

Figure 2.10: Checker running time with parametric tiling.

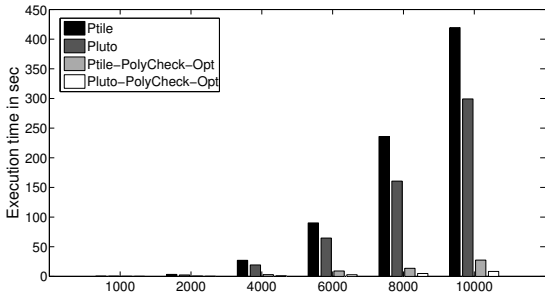

Figure 2.11: Checker running time across problem size (LU).

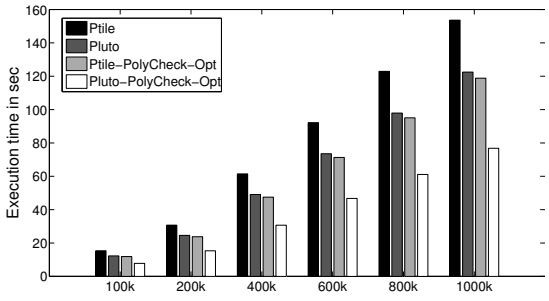

Figure 2.12: Checker running time across problem size (Reg_detect).

2.7 Conclusions

Using compositions of loop transformations to restructure the program for improved performance, optimizing compilers become increasingly complex and capable of generating transformed programs that are extremely far from the original code syntactically. It is critical to assess the correctness of compiler-generated code as these compilation frameworks themselves rely on millions of lines of code. In this chapter, we presented a new approach that exploits the properties of affine programs to generate, at compile time, a lightweight checking code. This checking code then is embedded into the transformed program and run for equivalence checking. Our approach addresses the main drawbacks

of alternative solutions for finding bugs in iteration reordering transformation frameworks, and its correctness and effectiveness have been extensively evaluated on several compilation frameworks that combine numerous kinds of program transformations.

CHAPTER 3

Static and Dynamic Frequency Scaling on Multicore CPUs

3.1 Introduction

Energy efficiency is of increasing importance in a number of use cases ranging from battery-operated devices to data centers striving to lower energy costs. DVFS (Dynamic Voltage and Frequency Scaling) [17] is a fundamental control mechanism in processors that enables a trade-off between performance and energy.

Typical DVFS approaches fall into two categories: schemes that employ specific frequencies for default policies (e.g., `powersave` vs `performance` governors in Linux) or schemes that observe the CPU execution and dynamically react to CPU load changes (e.g., the `on-demand` governor). Although energy savings have been demonstrated with these approaches [59, 69, 80, 72], we observe several limitations. First, as we show in this chapter, the frequency/voltage that optimizes CPU energy can vary significantly across processors. Even for the same compute-bound application, different processors have different energy-minimizing frequencies. Second, optimizing energy for a parallel application is highly dependent on its parallel scaling, which in turn depends on the operating frequency, an aspect mostly ignored in previous work. Third, dynamic schemes remain constrained to runtime inspection at specific time intervals, implying that short-running program phases

(e.g., a few times longer than the sampling interval) will not see all the benefits of dynamic DVFS compared to using the best frequency for the phase from the start.

In this work, we propose to address these limitations using two complementary strategies. First we present a simple lightweight runtime which throttles frequency based on periodic measurements of the energy efficiency of the application. This approach can exploit the specific properties of the CPU power profile, and is applicable to arbitrary programs. Then, we develop a compile-time approach to select the best frequency, but for program regions that can be analyzed using the polyhedral model [48], which is typical of compute-intensive kernels / library calls such as BLAS operations [91, 9] or image processing filters [92, 9]. Specifically, we develop static analysis to approximate the *operational intensity* (i.e., the ratio of operations executed per bytes of data transferred from the RAM) and the *parallel scaling* (i.e., the execution time speedup as a function of the number of cores) of a program region, in order to categorize a program region (e.g., compute-bound, memory-bound, etc.). The frequency and number of cores to use for each program category are chosen from the result of a one-time energy and energy-delay product profiling of the processor using microbenchmarks representative of different workload categories. Our extensive evaluation demonstrates significant energy and EDP savings over static (`powersave`) and dynamic (`on-demand`) schemes, validating our approach. We make the following contributions.

- We demonstrate that limitations of purely load-based approaches to DVFS can be addressed using a lightweight runtime approach we develop, optimizing CPU energy efficiency.

- We develop a compilation framework for the automatic selection of the frequency and number of cores to use for affine program regions, by categorizing a region based on its approximated operational intensity and parallel scaling potential.
- We provide extensive evaluation of our approach using 60 benchmarks and 5 multi-core CPUs, demonstrating average energy savings of 10% and EDP improvements of 40% on average over the `powersave` Linux governor.

3.2 Motivation and Overview

Previous research has shown that the increase in execution time when throttling down the frequency can be limited to a very small quantity by performing careful DVFS, leveraging the fact that on bandwidth-bound applications, the processor stalls a lot waiting for data transfer. Therefore, processor frequency could be reduced without significant wall-clock time increase (if the latency of main memory operations is not affected by DVFS), resulting in energy savings [59, 69].

Moreover, when considering the CPU energy alone (in isolation of the rest of the system), it could be intuitive to think that the lower the frequency, the lower the energy consumption: as power is often approximated to have a cubic relationship with frequency (assuming frequency and voltage have a linear relationship), using the minimal frequency (e.g., as with `powersave`) is expected to increase execution time linearly but decrease power in a nearly cubic way, thereby leading to the minimal CPU energy. We now show that this is not always true.

Energy profiles of CPUs We now discuss in detail the energy profiles of four off-the-shelf Intel x86 CPUs running on a Linux desktop. Table 3.1 outlines their key characteristics. We report in Fig. 3.1 a series of plots obtained by actual power and time measurements

using hardware counters available with Intel PCM [71], on DGEMM/MKL [70] for the top row. DGEMM/MKL is run on a large out-of-L3 problem size (reported to achieve near peak compute performance on these machines), and its execution time scales nearly perfectly with the frequency and number of cores used: it captures a compute-bound scenario exploiting fully all the processor capabilities. The data plotted is the measured CPU energy for the computation, where only one benchmark is running (no co-execution) and where Turbo-boost and other hardware DVFS mechanism have been turned off to obtain deterministic data. Each frequency was set prior to benchmarking using the `userspace` governor using the cpuFreq package, implicitly changing voltage along with frequency change as per the CPU specification.

Table 3.1: Processor characteristics

Intel CPU	Microarch.	Node	Cores	L1	L2	L3	Freq. GHz	Voltage V
Core i7-2600K	SandyBridge	32nm	4	32K	256K	8192K	$1.6 \sim 3.4$	$0.97 \sim 1.25$
Xeon E3-1275	IvyBridge	22nm	4	32K	256K	8192K	$1.6 \sim 3.5$	$0.97 \sim 1.09$
Xeon E5-2650	IvyBridge	22nm	8	32K	256K	20480K	$1.2 \sim 2.6$	$0.97 \sim 1.09$
Core i7-4770K	Haswell	22nm	4	32K	256K	8192K	$1.2 \sim 3.5$	$0.76 \sim 1.15$

We make several key observations from this data. First, *the optimal CPU energy is not achieved for the minimal nor maximal frequency* in most cases. Taking the case where all cores are used, on SB-4 and HSW-4 minimal energy is achieved at 1.6GHz (the minimal frequency for our SB setup), but it is achieved near the (but not at) maximal frequency for the two Ivy Bridge processors. The reason relates to the voltages used, and in particular to the ratio of voltage changes versus the ratio of frequency changes for each machine. Table 3.1 shows that for the two Ivy Bridge processors, the voltage range is much smaller

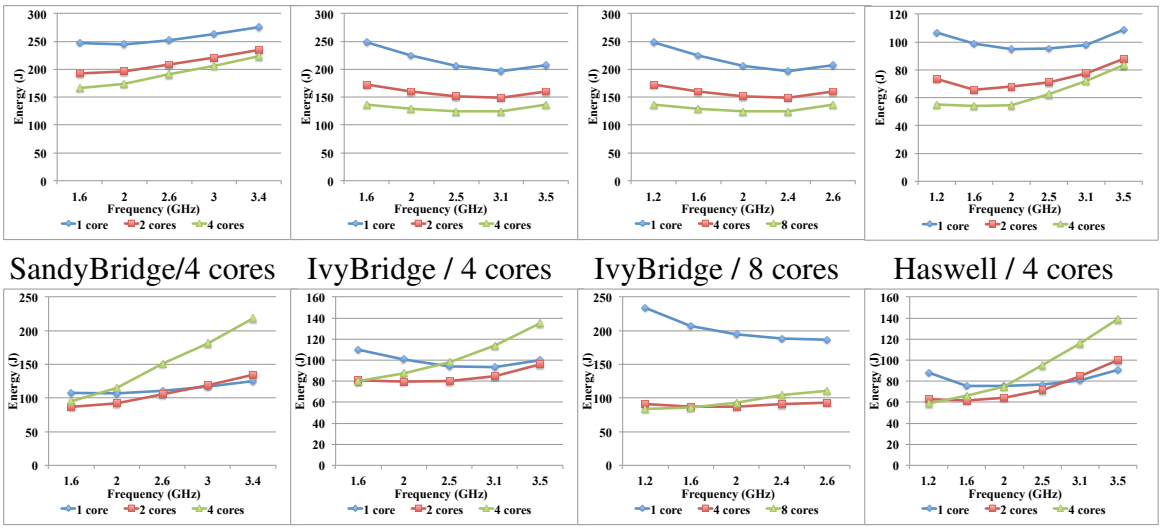

Figure 3.1: Energy for DGEMM/MKL (top row) and Jacobi 2D (bottom row)

than the other two (from 0.97V to 1.09V). Overly simplified power equations ignore key effects such as the relation between leakage current and temperature, and frequency and voltage relations. A more realistic power equation [114] captures the Poole-Frenkel effect, which relates leakage to temperature, and careful derivation of the evolution of the power equation as a function of changes in V, f and Temp demonstrates that the slope of increase of voltage vs. frequency can influence what is the optimal frequency for energy efficiency. De Vogeleer et al. [40] developed an analogous characterization on a mobile processor, modeling the energy variation between frequency steps as a function of voltage and temperature and obtaining a curve similar to our result for Haswell. They derived formulas to characterize the convex shape of the CPU energy efficiency for a fixed workload, as a function of CMOS characteristics including voltage and frequency increase relations. Here we observe across four different x86 Intel processors four different cases for the most energy efficient frequencies, for compute-bound codes. A runtime approach focusing only

on workload properties (e.g., the lack of stall cycles) and not taking into account these processor-specific effects would fail at selecting the optimal frequency for CPU energy minimization. Second, *the optimal CPU energy may be achieved at different frequencies depending on the number of cores used*. This is seen in particular on Haswell, where on 1 core 2GHz is the best frequency, but for 2- and 4-cores it is 1.6GHz. A similar situation is observed on Sandy Bridge, albeit the difference is very small. On the other hand, this is not observed for the Ivy Bridge cases.

To make the situation more complex, in practice the most energy efficient frequency is also affected by how the execution time evolves as a function of frequency. When the execution time decreases at a slower rate than the frequency increases (e.g., the expected acceleration is not achieved) this shifts the optimal frequency towards lower values than for the compute-bound cases like DGEMM/MKL. This is exemplified with a bandwidth-bound benchmark, as shown in the bottom row of Fig. 3.1. J2D is a Jacobi 2D code from PolyBench/C 3.2, which is parallelized naively among rows of the image, and uses out-of-L3 data too. It represents a bandwidth-bound case which is more realistic (e.g., less exacerbated) than the STREAM [86] benchmark.[4] We see a systematic shift of the most energy efficient frequency towards the left, that is lower frequencies. In addition, due to bandwidth saturation effects, the code does not have good weak scaling: adding cores does not decrease the execution time linearly. It leads to higher energy consumption increase for the 4-core/8-core cases than for single-core, when the frequency increases. *This motivates the need for an approach that also considers the application characteristics to determine its most energy efficient frequency*.

[4]STREAM suffers from insufficient number of loads emitted at low frequencies on single-core: it did not always saturate the bandwidth in our experiments.

Proposed approach Based on the observations above, we conclude that the best frequency to use to minimize CPU energy is per-processor-per-workload specific: one cannot limit to looking at the CPU load (e.g., using `on-demand`) or using the minimal frequency (e.g., using `powersave`) to minimize CPU energy. We propose two approaches to address this problem. (1) A lightweight runtime that adapts the frequency based on the CPU energy changes, dynamically during the application execution, see Sec. 3.3. (2) A compile-time approach for static selection of the ideal frequency for each affine computation kernel within a full application, based on (a) a new static analysis of the program's operational intensity and its potential for weak scaling across cores, see Sec. 3.4; and (b) a characterization of the processor's power profile for a handful of extreme scenarios (e.g., compute-bound, bandwidth-bound, etc.), via micro-benchmarking of a processor, as described in Sec. 3.5.

3.3 Adaptive Runtime to Optimize Energy Efficiency

We now describe our lightweight DVFS runtime approach. Its motivation is twofold. First, we want to design a runtime algorithm that is able to find automatically a good frequency to improve ***energy efficiency*** for any processor, in light the requirement for this frequency to be potentially radically different for different processors as shown in the previous section. Second, we want an approach to energy savings that does not require analysis of the source code nor profiling of the workload [69], in other words which can work on arbitrary programs. This runtime serves as a baseline for our compile-time frequency/core selection method, the core contribution, described in later Sec. 3.4. We will show how further energy savings can be achieved for specific computation kernels that can be analyzed with the polyhedral model in Sec. 3.6.

Algorithm 1 Runtime DVFS algorithm

Input: Sampling time interval: Δt
Energy efficiency threshold: ΔEff
OI threshold: ΔOI

```
1:  Eff_last = OI_last = 0
2:  lastChange = increaseFreq
3:  for each time quanta Δt do
4:      change = noChange
5:      Eff = computeEnergyEfficiency()
6:      OI = computeOI()
7:      if Eff < Eff_last + ΔEff then
8:          if OI < OI_last − ΔOI then
9:              change = reduceFreq
10:         else if |OI − OI_last| < ΔOI then
11:             change = reverse(lastChange)
12:         else if OI > OI_last + ΔOI then
13:             change = increaseFreq
14:         end if
15:     else if Eff > Eff_last + ΔEff then
16:         change = lastChange
17:     end if
18:     if change != noChange then
19:         changeFrequency(change)
20:         lastChange = change
21:     end if
22: end for
```

Runtime algorithm The main idea of our runtime is to continuously inspect the changes in energy efficiency, that is the energy consumption normalized by the number of instructions executed, and the changes in operational intensity (OI) of the application. A decision to change frequency is made either because the OI has changed, indicating a change in the nature of the computation being performed, or because the energy efficiency has decreased compared to the last sample. Within a single phase, that is a time segment with similar OI, we exploit the energy convexity rule: there is only one frequency maximizing energy efficiency [40], as illustrated in the previous section. Consequently, our algorithm to find this

frequency simply performs a gradient descent in the space of frequencies. The algorithm is shown on the right.

This algorithm has several parameters that can be tuned: the sampling interval Δt, the threshold for significant change in energy efficiency ΔEff, and the threshold for significant change in OI ΔOI. We have implemented this approach using Intel Performance Counter Monitor (PCM) [71] version 2.10, which provides counters to obtain the number of operations executed, the quantity of data transferred to/from RAM, and the average CPU power in between two time points. We compute the OI and energy efficiency from these counters. The runtime program is a separate thread implemented via Unix signals, triggering the monitoring and frequency selection every Δt time. We measured the time overhead of the runtime approach to be well below 1% when using $\Delta t = 50ms$. We chose $\Delta Eff = \Delta OI = 5\%$ in our experiments, and the runtime aggregates energy and OI metrics for all CPU cores, using Intel PCM counters. Precisely, the OI is measured respective to the RAM accesses and total number of CPU instructions executed, and the energy efficiency is the energy consumed by the socket in the time elapsed normalized by the number of total CPU instructions executed. Note that the possible frequencies the changeFrequency function can output are part of a pre-determined list of frequencies, we selected 5 (6 for Haswell) frequencies in between 1.6GHz and the maximal CPU frequency. Therefore the algorithm always output a valid frequency.

Experimental results Fig. 3.2 shows the energy savings, compared to `powersave` , of the `on-demand` Linux frequency governor, and of our runtime approach. We evaluate on 60 computation kernels as detailed in Sec. 3.6.1 to enable comparison with our compile-time approach developed in later sections, using OpenMP parallelization and executing

on all cores of the target machines. We detail the two most interesting machines for this experiment: a 8-core Ivy Bridge machine, where the lowest energy can be achieved for high frequencies, and a 4-core Haswell where the minimal frequency is not always minimizing energy, as shown in Sec. 5.1.

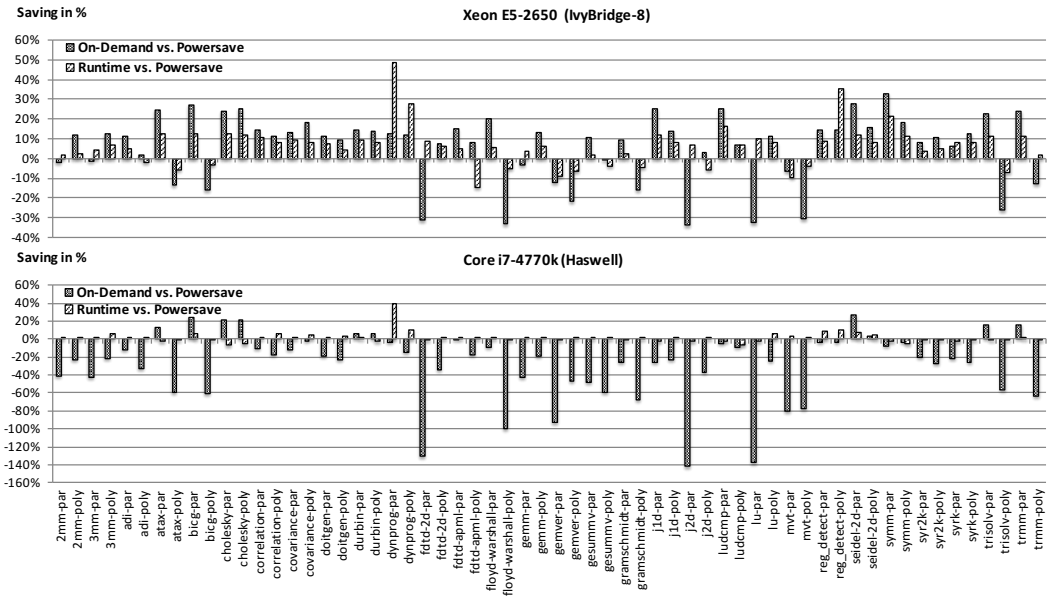

Figure 3.2: Comparison of energy savings for `on-demand` Linux governor and our proposed runtime, versus `powersave`

As expected, the `on-demand` governor can significantly improve energy consumption on Ivy Bridge compared to `powersave`, but can also be highly detrimental in particular for more bandwidth-bound benchmarks. For Haswell, `on-demand` is typically highly detrimental. In contrast, our proposed runtime approach is only rarely detrimental compared to `powersave` but on average significantly boost energy savings for Ivy Bridge. It addresses

the deficiencies of the two Linux governors by looking at energy efficiency instead of simply CPU workload, and offers a viable one-size-fits-all algorithm despite the processor-specific characteristics of the energy-minimizing frequency. Note also that as `powersave` runs at the minimal frequency, our runtime approach can only equal or improve the overall execution time.

However, as we show in the next sections, further gains can be attained by addressing two inherent limitations of a runtime-based approach: (1) the "wasted" time to reach the optimal frequency for a regular computation chunk (i.e., a phase in the program); and (2) the inability to adapt the number of CPU cores allocated to the chunk, based on parallel scaling. We show how these limitations can be efficiently addressed for affine kernels.

3.4 Static Analyses

A fundamental property of affine computations is to have only static control-flow and data-flow, that is the code executed does not depend on the dataset value, but only on the value of loop iterators and program constants. This regularity enables the design of key *static* analyses for these program regions, for instance to characterize their operational intensity. We aim to substitute our runtime approach based on runtime OI inspection by a compile-time characterization of the program region, using novel analyses we now develop.

3.4.1 Approximating Operational Intensity

The operational intensity (OI) of a program, typically in FLOP per byte for scientific codes, is the ratio of operations executed per data moved to execute these operations. The OI may be computed during run-time using data movement count from the transfers from RAM to the last-level cache, or another level of cache. In this work, we are interested in categorizing a program region (i.e. a loop nest) as either memory-bound or compute-bound:

only a coarse approximation of the OI is needed. We also have the goal of developing a very fast analysis that can be applied on large source codes. Indeed, the result of applying polyhedral transformations to a program may lead to a code of thousands of lines from an input loop nest of a few lines, as is the case for numerous benchmarks we evaluate in Sec. 3.6.

We propose a ***fast*** static analysis, which can categorize a program region at compile-time in less than one second for affine programs made of possibly thousands of lines of code. Prior work on cache miss modeling for affine programs, such as the Cache Miss Equations [62] or the work of Chatterjee et al. [39], can provide exact count of cache misses but at the expense of a prohibitively costly static analysis which may take hours to complete even for simple codes [39]. These are not suitable for our objectives. In this work we trade off accuracy for analysis speed, as an approximation of the OI is sufficient for our need of categorizing a program region. Our analysis takes an arbitrary C code as input, automatically extracts affine regions, and for each approximates the OI. It does not perform any transformation, and works in a stand-alone fashion.

Background on polyhedral program representation The polyhedral model is a flexible and expressive representation for imperfectly nested loops with statically predictable control flow. Loop nests amenable to this algebraic representation are called ***static control parts*** (SCoPs) [48, 63], roughly defined as a set of consecutive statements such that loop bounds and conditionals involved are affine functions of the enclosing loop iterators and variables that are invariant during the SCoP execution. Numerous scientific kernels exhibit those properties; they can be found in image processing filters, linear algebra computations, etc. [63]. We now describe the key data structures and objects to represent and manipulate

programs we need in this work. We illustrate the main ideas using the simple example in Fig. 3.3 below, a single-sweep of a 2D stencil.

```
  for (i = 1; i < height - 1; ++i)
    for (j = 1; j < width - 1; ++j)
R:    Out[i][j] = 4*In[i][j]-In[i-1][j]
        -In[i+1][j]-In[i][j-1]-In[i][j+1];
```

Figure 3.3: Jacobi 2D 5 point sweep

Iteration domains For each textual statement in the program, the set of its run-time instances is captured with an integer set bounded by affine inequalities, intersected with an affine integer lattice [25], that is the iteration domain of the statement. Each point in this set represents a unique dynamic instance of the statement, such that the coordinates of the point corresponds to the value the surrounding loop iterators take when this instance is executed. For instance for statement `R` in Fig. 3.3, its iteration domain $\mathcal{D}_R$ is:

$$\mathcal{D}_R = \{(i,j) \in \mathbb{Z}^2 \mid 1 \leq i < \text{height} - 1 \wedge 1 \leq j < \text{width} - 1\}$$

We denote $\vec{x}_R \in \mathcal{D}_R$ as a point in the iteration domain.

Access functions They capture the location of the data accessed by the statement. In static control parts, memory accesses are performed through array references (a variable being a particular case of an array). We restrict ourselves to subscripts that are affine expressions of surrounding loop counters and global parameters. For instance, the subscript function of the fifth read reference to `In` in R, `In[i][j+1]` , surrounded by 2 loops i and j is $F^{In}_{5,R}(i,j) = (i, j+1)$ and its values when evaluated for all pairs $(i,j) \in \mathcal{D}_R$ capture the set of addresses of A accessed by this reference.

Data space We first define the data space of an array A for a program, that is the set of data accessed by all references to a specific array, during the entire program execution. The data space is simply the union of the sets of data elements accessed through the various access functions referencing this array, for each value of the surrounding loop iterators where the reference is executed. The polyhedral program representation enables the use of the image of a polyhedron (e.g., the iteration domain) by an affine function (e.g., the access function) to capture data spaces. The image of a polyhedron $\mathcal{D}$ by an affine function F is defined as the set $\{\vec{y} \mid \forall \vec{x} \in \mathcal{D},\ F(\vec{x}) = \vec{y}\}$.

Definition 1 (Data space)**.** *Given an array A, a collection of statements $\mathcal{S}$, and the associated set of memory references $F^A_{i,S}$ with $S \in \mathcal{S}$, the data space of A is the set of unique data elements accessed during the execution of the statements. It is the union of the image of the iteration domains by the i access functions in each statement:*

$$DS^A = \bigcup_{S \in \mathcal{S}} Image(F^A_{i,S}, \mathcal{D}_S)$$

We remark that DS^A is not necessarily a convex set, but can still be manipulated with existing polyhedral libraries. For example, in Fig. 3.3 $DS^{In} = \{(x,y) \mid 0 \leq x < height, 0 \leq y < width\}$ and $DS^{Out} = \{(x,y) \mid 1 \leq x < height - 1, 1 \leq y < width - 1\}$.

Data space of a loop iteration It is very convenient to be able to restrict the data space to a particular loop iteration, for instance compute the data space for one execution of the j loop (i.e., one iteration of the i loop). We will use such a mechanism to approximate cache misses below. A simple approach to achieve this is to compute a "slice" of interest of the iteration domain, and use this sliced domain in the data space computation. A parametric slice [96] along a set of dimensions (i.e., loops) is defined as follows. Given a loop nest

with a loop l of depth n surrounded by $k-1$ loops, the parametric slice PS of loop l is a subset of $\mathbb{Z}^n$ defined as:

$$PS_{l,\alpha} = \{(x_1,\ldots,x_n) \in \mathbb{Z}^n | x_1 = p_1,\ldots,x_{k-1} = p_{k-1}, x_k = p_k + \alpha\}$$

where $p_1,\ldots,p_n$ are parametric constants unrestricted on $\mathbb{Z}$, and α is an integer value. For example, a slice of the first loop in line 1 of Figure 3.3 for statement S is: $PS_{l,0} = \{(i,j) \in \mathbb{Z}^2 | i = p_1\}$ This is a (parametric) set of 2D integer points with the first component of each point always having the same (unknown yet constant) value. When intersecting a slice with an iteration domain one fixes certain loops to a unique "generic" value which is by construction in the set of possible values the loop iterators can take when the program execute.

We can now adapt the definition of a data space to the subset of data which is accessed by a loop iteration.

Definition 2 (Data space of a loop iteration)**.** *Given an array A, a collection of statements $\mathcal{S}$ surrounded by a loop l and their associated set of memory references $F^A_{i,S}$ with $S \in \mathcal{S}$, and $PS_{l,0}$ a PS for loop l, the data space of A is the set of unique data elements accessed during one iteration of l:*

$$DS^A_{PS_{l,0}} = \bigcup_{S \in \mathcal{S}} Image\left(F^A_{i,S}, \left(\mathcal{D}_S \cap PS_{l,0}\right)\right)$$

The data spaces of loop iterations we manipulate are union of $\mathcal{Z}-polyhedra$, and operations such as intersection ($\cap$), union ($\cup$), difference ($\setminus$) and critically computing a counting function (e.g., $\#DS(In) = \text{height} * \text{width}$) can be done at compile-time using specialized libraries such as the Integer Set Library [124] and Barvinok [128].

Approximating Data Movement

The DL (Distinct Lines) model was designed originally to estimate the number of distinct cache lines, or TLB entries, accessed in a loop nest [106]. It essentially represents the footprint of a computation in terms of cache lines accessed, thereby taking into account spatial locality. DL formulas are typically used to analytically find values for the loop bounds (e.g., tile sizes) to ensure the number of distinct lines accessed is below the cache capacity, therefore ensuring the data reuse is implemented in cache.

When the DL of a loop nest is lower than the cache size, it is then a good approximation of the number of cache misses: only cold misses will occur, one per line of cache accessed, as the reuse will be fully implemented without any data being evicted. Conflict misses are ignored here, and in the following we will assume they do not dominate the miss behavior, e.g., arrays have been properly padded. On the other hand, when the DL is larger than the cache size, DL does not allow to determine an estimate of the number of misses: it depends on the schedule of operations, and in particular of the reuse distance between references to the same array.

In this work, we propose to approximate the data movement between two levels of memory by approximating the number of cache misses at a certain level. We achieve this by (1) formulating DL in the polyhedral framework, using the concepts presented above; (2) extending it to also capture the data reuse between consecutive loop iterations at any loop level; and (3) designing an algorithm that approximates the number of misses when the DL exceeds the cache size.

Computing DL using Polyhedra The data space definitions above are the essential bricks to compute a polyhedral expression of the number of distinct lines: it already provides the set of distinct memory locations accessed by a (slice of the) program. The only missing part is to translate this into distinct cache lines. This is simply done by first representing the mapping from memory address to cache line of size ls (line size), using an affine function of the same dimensionality as the array. For instance for a 2-dimensional array and $ls = 8$ (e.g., `A[N][M]`) we create the function linemap$(i, j) = (i, j/8)$. Then, the set of distinct lines for an array A is simply the image of DL^A by the linemap function. For example DL^{In} using $ls = 8$ is: $DL^{In} = \{(a,b) \mid 0 \leq x < height, 0 \leq y < width, a = x, 8 * b \leq y < 8 * b + 8\}$

In a manner analogous to data spaces, one can compute the DL of a particular loop iteration by using parametric slices of the domain. Such case will be denoted $DL^A_{PS_{l,\alpha}}$ for array A and loop l with offset α.

Algorithm for Miss Estimation We are now equipped to build our procedure to estimate the misses. The idea is the following: we will recursively compute the DL of a loop (summing it for all arrays accessed by that loop), from the inner-most loops to the outer-most loops. For each loop l, its number of misses is estimated as either the product of its trip count and the number of misses of its loop body if the DL of this loop exceeds the cache size, or its DL if it is smaller than the cache size. For inner-most loops bodies, we set their number of misses to the DL of the loop body, whatever its value. Some additional treatment is done to capture the data reuse between consecutive iterations of a loop body, to adjust the number of misses. We optimistically assume if data is reused between two iterations, then it corresponds to the part of the data that way loaded last at the previous

iteration, e.g., if this set is smaller than the cache size, it is not evicted from the cache by other data of the previous iteration. Our algorithm is shown in Alg. 2. For simplicity we assume the entire program is surrounded by a fake loop fk having a single iteration, and we note that DL[] is a map, where DL[x] associates object x to an integer value.

Algorithm 2 EstimateCacheMisses

Input: Number of array elements per cache line: ls
cache size, in lines: C
Polyhedral program: P
Output: Estimate of misses

1: **for all** loops l **do**
2: Misses[l] = DLBase[l] = DLnext[l] = Processed[l] = 0
3: **end for**
4: **for all** Arrays A **do**
5: **for all** loops l **do**
6: DLbase[l] += $\#(DL^A_{PS_{l,0}})$
7: DLnext[l] += $\#(DL^A_{PS_{l,0}} \setminus DL^A_{PS_{l,1}})$
8: **end for**
9: **end for**
10: **for all** loops l in postfix AST order and Processed[l] == 0 **do**
11: **if** DLbase[l] $<$ C **then**
12: Misses[l] = DLbase[l]
13: **else**
14: **if** l is inner-most loop **then**
15: Misses[l] = DLbase[l]
16: **else**
17: Miss = 0
18: **for all** loops ll immediately surrounded by l **do**
19: Miss += Misses[ll]
20: Processed[ll] = 1
21: **end for**
22: Misses[l] = Miss * TripCount(l)
23: **if** DLnext[l] $<$ C **then**
24: Misses[l] -= DLnext[l]
25: **end if**
26: **end if**
27: **end if**
28: Processed[l] = 1
29: **end for**
30: **return** Misses[fk];

Alg. 2 uses the TripCount(l) function to compute the trip count of a loop. For affine programs, this is always computable thanks to the static control-flow property. This function is implemented by first forming the union of the iteration domains of all statements surrounded by the loop, then projecting this set onto the dimension corresponding to the loop of interest.

A key remark about our algorithm is that it performs comparisons of expressions representing the number of points in polyhedra. While techniques exist to create such expressions as a function of the parameters of the program, this poses challenges when attempting to compare two different parameters: given height and width without further information, is height $>$ width? Or height $<$ width? In general, this problem is not decidable. To avoid this issue, we require the algorithm to run on an instance of the polyhedral program where the parameter values are known numerical constants, e.g., 1024. In practice, we extract the polyhedral representation and compute data spaces using parametric representations, but add a context information about the exact parameter value before estimating the size of the polyhedra (the # operation).

Approximating FLOPs, and Getting OI

The last element needed to approximate the operational intensity is the number of op-

Algorithm 3 Estimate Operational Intensity

Input: Number of array elements per cache line: ls
cache size, in lines: C
Polyhedral program: P
Parameter values: $\vec{n}$
Output: Estimate of OI

```
1: P' = attachContextInformation(n, P)
2: Misses = EstimateCacheMisses(ls, C, P')
3: Flops = countFlops(P')
4: return Flops/Misses
```

erations executed in the program, so that we can divide it by the number of memory movements (i.e., number of misses). This is straightforward in the polyhedral representation: we inspect the AST of each statement body, collecting the number of operations not part of an array subscript expression, and multiply it by the number of points in the iteration domain of the statement. As the above process requires explicit values for the parameters, we leverage the known values to obtain a numerical value for the FLOP count. The overall process to compute OI is outlined in Alg. 3.

An interesting aspect of this compile-time approach is that it can be applied incrementally on each loop (nest) of the program, to detect loop nests with significantly different behaviors. That is, this algorithm can be used to detect compute-bound and memory-bound application phases, allowing for individually fine-tuning the DVFS decision for each phase.

3.4.2 Parallelism Features

The parallelism features we extract are simple in comparison of the OI approximation, yet critical in terms of the quality of the approach. We have focused our implementation to cover the cases of automatic parallelization implemented by the PoCC polyhedral compiler [8], that is if a code is parallel then it has at least one OpenMP `for` pragma in the code, and no other type of OpenMP pragmas. This is reminiscent of loop parallelizing compilers.

The first feature we extract is whether the program is sequential or parallel. This is simply done by checking if the program has any OpenMP pragma in it, if not then it is sequential. The second feature attempts to capture poor scalability of the code, that is the performance improvement does not scale with the number of cores. For this feature we again rely on the assumption that there are only OpenMP for pragmas to model the

Algorithm 4 Check Poor Parallel Scaling

Input: Polyhedral program: P
Max. number of cores: c
Output: Compute parallel scaling

```
1: poorScaleWork = goodScaleWork = 0
2: for all parallel loops l in P do
3:     l' = stripMine(l, c)
4:     pds = parametricSlice(l'.inner)
5:     if countHasZeroIteration(pds,c) ≥ c/2 then
6:         poorScaleWork += countFlops(l')
7:     else
8:         goodScaleWork += countFlops(l')
9:     end if
10: end for
11: return poorScaleWork > goodScaleWork
```

parallelization. We analyze the AST to form an estimate of the regularity of the parallel loop trip count, studying the parallel workload properties as detailed in Alg. 4.

Function stripMine performs a strip-mining of the loop so that the resulting outer loop has as many iterations as the maximal number of cores c available on the target machine, enabling to analyze its inner loop $l'.inner$ that is the outer-most serial loop in a thread. We then perform parametric slicing to reason on the different trip counts this loop can have, e.g., for load-imbalanced cases this loop may iterate 0 times. Function countHasZeroIteration counts the number of values of l' for which the trip count of $l'.inner$ can be equal to 0, if it can be for more than half of the available cores then this loop is considered having poor scaling.

Interestingly, this feature has only limited use in our tested benchmarks, as the OI feature is actually a better discriminant for codes which do not scale due to bandwidth saturation, which is the common case in our test suite. Here the poor scalability feature is limited to capturing compute-bound codes with high load imbalance, as arises typically in

pipeline-parallel schedules with large startup and draining phases compared to the steady state.

3.5 Processor Characterization

3.5.1 One-Time Machine Profiling

As shown in Sec. 5.1, the frequency / number of cores configuration needs to be adapted as a function of the nature of the code executing. To find the best frequency/core configuration for a particular machine, we perform a profiling of four benchmarks on the target machine, running them on all frequency steps and cores setup available, and collecting the execution time and power using hardware counters. This enables us to build a profile of the energy and energy-delay product curves for each benchmark, which are the two metrics we optimize for. These benchmarks have been specifically built to exacerbate one of the features that is computed by our static analysis.

Compute-bound code For this case, we use off-the-shelf DGEMM/MKL, in the same setup as shown in Sec. 5.1. We observed the near perfect scaling of this code / problem size across the architectures tested, and while being totally compute-bound by nature it is an actual workload with heavy data traffic.

Bandwidth-bound code Bandwidth-bound programs are frequent, especially if no data locality optimization has been performed on the original benchmark. For this case we created a benchmark that implements both spatial and temporal locality via tiling, with a low arithmetic intensity. It is inspired from an iterative stencil (e.g., Jacobi-2D). This benchmark is bandwidth-bound, but with a relevant balance between computations and communications.

Sequential code The sequential code we use as representative of sequential workloads is a SIMD-vectorizable benchmark, with balanced arithmetic intensity, i.e., there is non-negligible memory traffic. It was built from key code features of the durbin benchmark from PolyBench/C.

Code with poor scalability The benchmark used here essentially consists of an OpenMP parallelization of a triangular loop with low trip count, to ensure high load imbalance between threads. The computation performed in the loop body has balanced arithmetic intensity, and is inspired from LU factorization.

3.5.2 Decision Tree for Frequency/Core pair

Finally we conclude by displaying our process to assign at compile-time a frequency to an affine program region. A program is classified using its operational intensity and its parallelism scaling features, to determine to which of the four above categories it is closest to. In particular, we have found that ordering the decision by prioritizing the cases where the dominant effect has the highest impact delivers the best results.

The decision process is identical whether we optimize for E or EDP, only the frequency/core configuration selected changes between E and EDP, using the best configuration found by profiling for each metric independently. The decision process is as follows:

1) If the code is sequential, choose config_sequential;
2) else if the code is bandwidth-bound (i.e., its is OI below threshold, found during profiling), choose config_bwbound
3) else if the code has poor scaling, choose config_poorscale
4) else choose config_computebound.

3.6 Experimental Results

3.6.1 Experimental Protocol

Benchmarks considered We use the PolyBench/C 3.2 benchmark suite [9], a popular suite of 30 numerical computations written using affine loops and static control flow. It spans computations in numerous domains such as image processing, linear algebra, etc. For each benchmark, we considered two versions of the code: par was generated by applying a simple auto-parallelization without any code transformation. It amounts to inserting OpenMP parallel for pragmas around the outer-most parallel loops in the code, and vectorization pragmas around the inner-most parallel loops. This can very substantially improve the performance of the original codes, which are sequential by default, and substitute for the auto-parallelization schemes of the C back-end compilers used. We used the PoCC source-to-source polyhedral compiler [8] to generate the optimized C files. The second variant, poly, is the result of a complex automatic program transformation stage aimed at improving data locality via tiling, coarse-grain parallelism, and fine-grain parallelism. It uses the Pluto algorithm [34] combined with several other optimizers of PoCC implementing a model-driven pre-vectorization, unrolling, and parallelization. For these variants, the operational intensity is typically significantly improved after transformation: temporal data locality is improved via loop tiling whenever possible. These two versions form a total of 60 benchmarks we evaluated on. The static analysis was also implemented in the PoCC compiler, in a fully automated way, to produce the features of the benchmark for its categorization. Each 60 source code was separately analyzed, the analysis taking at most a couple seconds to complete.

Each benchmark was compiled using GNU compiler GCC-4.8.1 with `-O3 -fopenmp` optimization for the Intel CPUs, along with `-m64 -mcpu=power8 -mtune=power8` for the IBM POWER8.

Collecting energy and time To assess the quality of our framework we needed to compute the optimal frequency / core configuration for each binary individually, by collecting its energy and execution time for each frequency / core configuration evaluated on the target machine. This data was collected using Intel PCM [71] for the Intel chips, and IBM AMESTER [51] for the POWER8. The instrumentation was inserted around each kernel of interest. To obtain stable and sound energy measurement value, Turbo-boost was turned off on the Intel chips. The following process was repeated as needed to achieve an execution time of around one minute, and the average energy and time were obtained: (1) flush data caches; (2) start instruments; (3) execute kernels; (4) stop and collect energy & execution time

On each target machine we used the different frequencies made available by the Linux OS running on these machines, Table 3.1 lists the frequency ranges. We used 5-6 frequency steps per machine, spaced evenly in the frequency range. We tested three core counts: 1, half, and all available. So a total of 78 configurations (15-18 per machine) was tested for each of the 60 binaries. We empirically found the best configuration (frequency / core) to optimize energy or energy-delay product for each benchmark, when using a fixed configuration for the entire kernel execution. In the following these are designated as the Best configuration for a given benchmark and optimization metric.

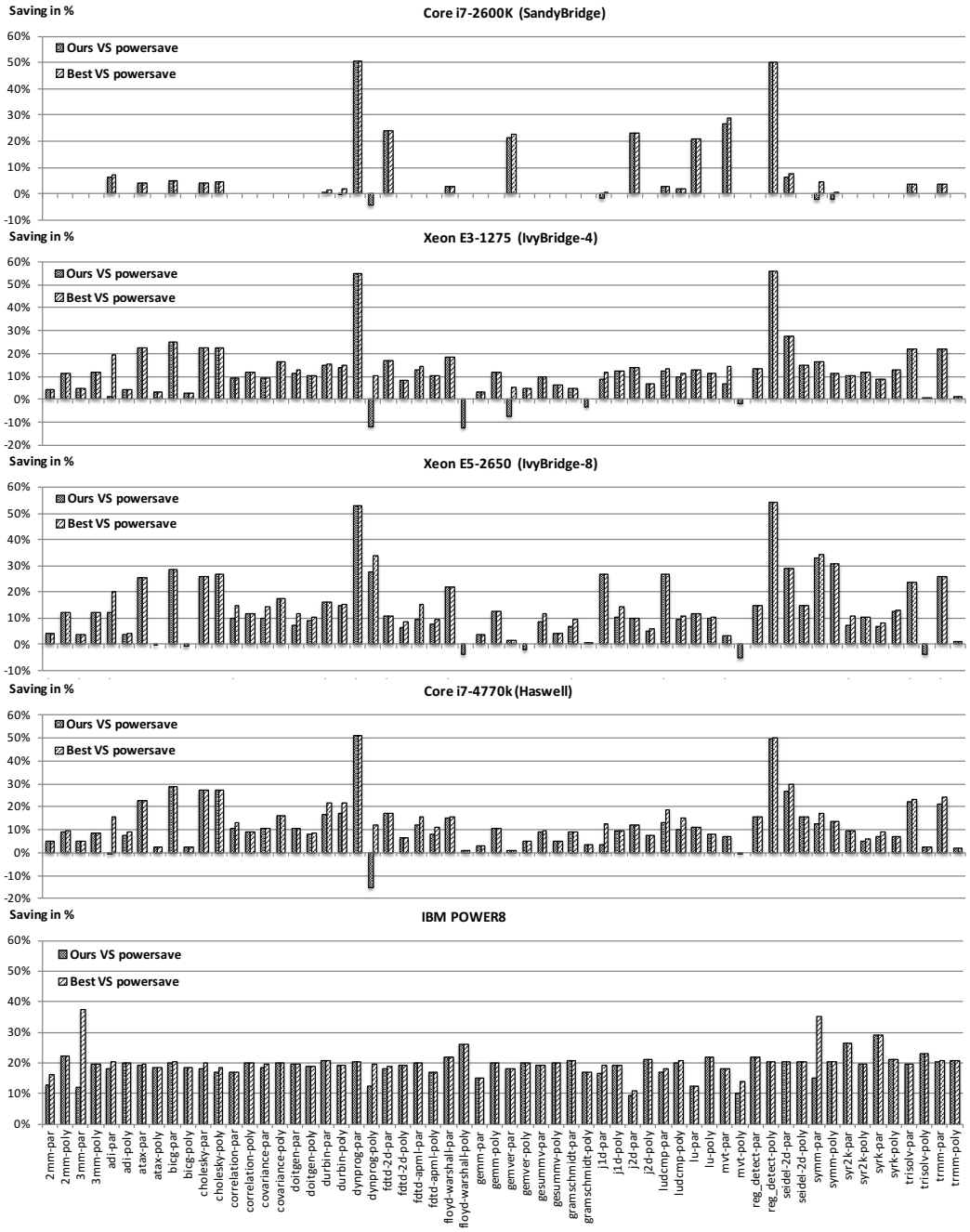

Figure 3.4: Details of static approach

3.6.2 Summary of Results

Fig. 3.4 summarize our performance results. The five bar charts provide the energy savings, on a per-benchmark basis, of our approach (Ours) compared to using the `powersave`

Linux governor, and the savings that can be achieved by using the Best frequency / core configuration found empirically by testing all of them. Fig. 3.5 summarizes the average energy savings E and energy-delay product EDP improvement across all 60 benchmarks, compared to `powersave`, of the `on-demand` Linux governor, our implementation of the CPUMiser[5] run-time approach [59], Ours (static), and Best (static) empirically found.

Energy savings over powersave The `powersave` Linux governor uses all available cores and sets the frequency to the minimal one. This principle takes idea in that a frequency increase has a cubic effect on power increase, but a linear effect in execution time decrease, so using the minimal frequency should minimize CPU energy. But as we have demonstrated in Sec. 5.1, this may not lead to minimizing energy, by far. Furthermore, this approach essentially ignores the rest of the system power consumption. *In this work, we show how CPU energy savings can be achieved by increasing the frequency*, which in turn typically leads to also reducing the kernel execution time compared to `powersave`. As a consequence, the system energy consumption is further reduced with our approach compared to `powersave`, we however do not report it and focus solely on CPU energy reports.

For SandyBridge, we observe minimizing frequency is a viable way to minimize energy, and for numerous benchmarks our approach selects the minimal frequency (therefore showing 0% savings over `powersave`), which was also found to be the best scenario by empirical evaluation of all frequency/core pairs (the Best bars). Still a significant improvement is achieved using our approach for benchmarks which are either sequential or having poor scalability, such as dynprog-par, where we select a reduced number of cores.

The benefits of our approach over `powersave` are highlighted with architectures where the energy-minimizing frequency is not the minimal one, such as on Ivy Bridge and Haswell.

[5]Our IBM POWER8 setup does not allow for run-time DVFS, so CPUMiser results are omitted.

An average savings of 13% is achieved for the 8-core Ivy Bridge, coming from selecting higher frequencies to balance completion time and power consumed following the processor-specific characteristics. For almost all kernels savings above 10% are achieved with our static approach. We also expose savings comparable to the best possible ones for the vast majority of cases, demonstrating the viability of our compile-time categorization of codes. For cases with lower savings than Best, the frequency/core selected was typically one step above the best one (e.g., we used 1.6GHz instead of the ideal 1.2GHz) and suggests further improvements may be achieved using more categories for the applications.

Energy and EDP improvements Fig. 3.5 summarizes the results for all architectures. For Ivy Bridge and P8, the energy difference of `on-demand` over `powersave` is characteristic to these machines, where actually maximal frequency leads to minimizing energy for compute-bound codes. This is exacerbated by the fact that the poly version of the benchmarks have been transformed for improved data locality, and for many have become essentially compute-bound. Our static analysis seamlessly captured these changes, with for instance gemm and seidel-2d moving from being categorized as memory-bound in their par version to compute-bound in their poly version as the result of transformations for locality, tiling and SIMD vectorization.

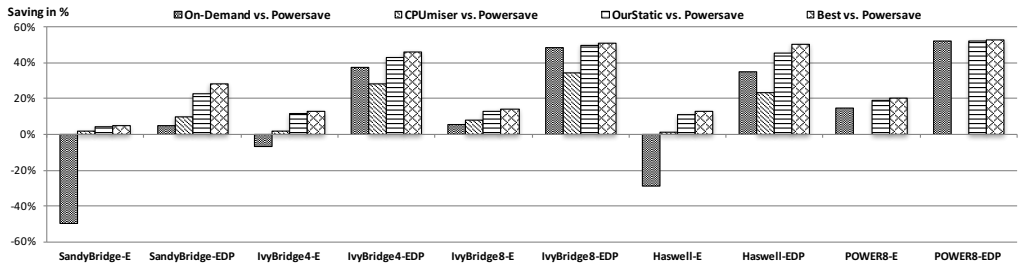

Figure 3.5: Summary of experimental results

CPUMiser by Rong Ge et al. [59] is a run-time approach to adapt the frequency/voltage based on the measured cycle per instructions (CPI) achieved by the workload. We have re-implemented it using exactly the same runtime harness as in Sec. 3.3, also using Intel PCM to capture CPU behavior. CPUMiser is geared towards finding a frequency which does not reduce the execution time beyond a threshold, in Fig. 3.5 we configured CPUMiser to use 100ms sampling interval, and 10% maximum performance loss. More comprehensive results are shown in Sec. 3.6.3. CPUMiser is able to work with arbitrary binaries/programs, contrary to our static approach that is restricted to affine program regions. But CPUMiser is unable to adapt the number of cores to use, leading to increased energy usage compared to our technique for codes with poor scaling. In addition, CPUMiser does not consider energy efficiency as the optimization objective, it only attempts to reduce frequency while keeping CPI increase under a certain threshold. As we have shown in Sec. 5.1 the frequency that minimizes CPU energy may be below the maximal frequency, even for compute-bound codes which would see a significant increase in CPI by using a lower-than-maximal frequency. This is one of the reason why our approach outperforms CPUMiser, this is particularly visible for Haswell.

Finally, when optimizing for EDP we observe consistent conclusions between the different schemes as when optimizing for energy, with our approach consistently outperforming competing schemes and being close to the best achievable in our framework.

3.6.3 Comparison with Runtime DVFS

Fig. 3.6 presents a comparative study of the energy savings of various DVFS schemes. We report the results of CPUMiser with different performance degradation parameters: 10%, 50% and 100% respectively. We also tuned the Linux `on-demand` (Od) and `conservative`

(Co) governors. They are both dynamic schemes based on CPU usage, and can be parameterized for different CPU loads. The `conservative` governor adapts the frequency gracefully, while `on-demand` goes to maximal frequency when the load threshold is reached. We vary the CPU load threshold from 45% to 95% with step of 10% for both governors. We also report Ours, the performance of our static approach as presented above, for comparison. Fig. 3.7 summarizes the energy-delay product savings over the `powersave` governor for the same DVFS schemes.

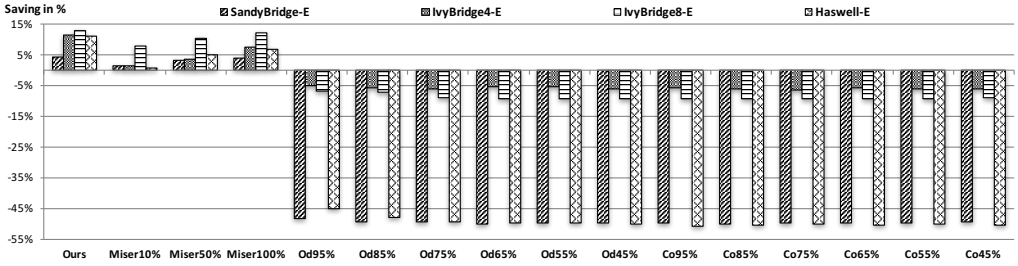

Figure 3.6: Energy savings versus runtime DVFS schemes

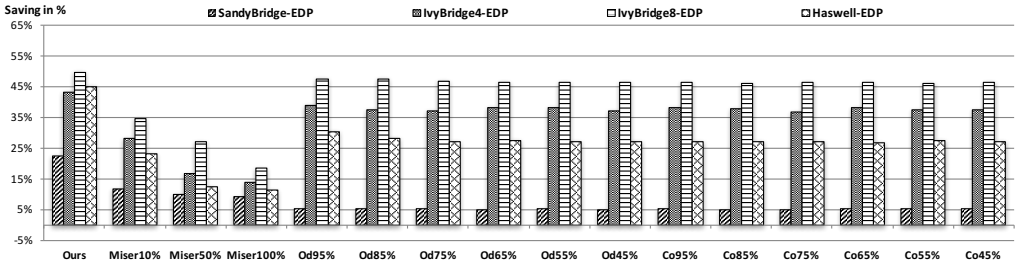

Figure 3.7: EDP savings versus runtime DVFS schemes

Given the nature of the workloads evaluated, both `conservative` and `on-demand` use the maximal CPU frequency most of the time. Consequently, the CPU energy compared to `powersave` increases on average, as for most benchmark maximal frequency is not the way to minimize CPU energy especially for SandyBridge and Haswell. In contrast, for Ivy Bridge where running at maximal frequency is often good for energy optimization, the

energy loss of these governors compared to `powersave` is very small . This analysis holds true for EDP results in Fig. 3.7: significant EDP savings over powersave are achieved by these governors, especially for Ivy Bridge. Note in all cases our static approach Ours outperforms these governors, whether it was set to optimize CPU energy or to optimize energy-delay product.

CPUMiser results show that by allowing for large performance degradation (e.g., 2x, that is 100%) it is possible to reach good levels of CPU energy savings, as shown in Fig. 3.6, although in turn it leads to limited EDP savings, as shown in Fig. 3.7. Our approach significantly outperforms CPUMiser in terms of EDP savings, in fact we also evaluated CPUMiser with 1% as the performance threshold and observed our static approach achieves EDP savings 2x or more higher than CPUMiser for SandyBridge and Haswell.

3.6.4 Summary of Intel ICC and GNU GCC experiments

We conducted an extensive characterization of the energy and energy-delay product of the same benchmarks as in the previous section, for a total of 60 benchmarks. Each was compiled using GNU compiler/GCC-4.8.1 with -O3 optimization, and also Intel compiler/ICC-2013-update5 compiler with -fast optimization, for a total of 120 different binaries evaluated.

Summary of Results Table 3.2 reports the summary of experiments when optimizing for CPU energy minimization. For each 60 benchmarks, the average energy savings (E) compared to `powersave` is reported for three approaches. Best is the best frequency / core configuration found by exhaustive search of all frequency/core configurations, individually for each binary tested and each CPU tested. Perf. is the `performance` Linux governor, and Ours is our static approach.

Microarchitecture	Compiler	E save Best vs powersave	E save Perf. vs powersave	E save Ours vs powersave
Haswell	GCC	12.95%	-28.84%	**11.27%**
	ICC	11.84%	-35.33%	**10.43%**
IvyBridge-4	GCC	12.83%	-6.48%	**11.35%**
	ICC	11.69%	-9.11%	**10.71%**
IvyBridge-8	GCC	14.38%	5.78%	**13.11%**
	ICC	13.39%	5.30%	**12.15%**
SandyBridge	GCC	4.59%	-49.71%	**4.17%**
	ICC	3.75%	-51.11%	**3.06%**

Table 3.2: Energy efficiency summary

A similar table is shown in Table 3.3, where we compare the same three approaches. Note the Best approach here is the result of empirical search for the best frequency / core configuration that maximizes EDP. We additionally report the average wall-clock execution time increase using our predicted configuration versus using performance.

Microarchitecture	Compiler	EDP save Best vs powersave	EDP save Perf. vs powersave	EDP save Ours vs powersave	Time loss Ours vs Perf.
Haswell	GCC	50.52%	35.24%	**45.16%**	**11.64%**
	ICC	47.57%	31.32%	**43.26%**	**11.08%**
IvyBridge-4	GCC	46.10%	37.14%	**43.14%**	**6.87%**
	ICC	44.70%	35.46%	**42.00%**	**3.94%**
IvyBridge-8	GCC	51.17%	48.44%	**49.50%**	**0.17%**
	ICC	51.22%	48.49%	**50.03%**	**1.92%**
SandyBridge	GCC	28.41%	4.87%	**22.61%**	**8.05%**
	ICC	25.46%	5.67%	**20.77%**	**9.04%**

Table 3.3: Energy-Delay Product efficiency summary

We observe that on average, our approach vastly outperforms performance and powersave in terms of both energy savings and EDP improvements: we are in fact very close to the optimal situation for energy, with our approach being within 102% or less (i.e., less than

2% worse) of the truly minimal energy that can be achieved using the best configuration found individually for each binary tested and each processor. This result is even further exacerbated for EDP, where our EDP savings can be up to 50% than using the performance governor. Our predictor approach achieves always within less than 10% of the optimal EDP. The penalty in execution times is also very good: the codes are slowed down on average compared to their maximal speed by at most 11.6%, and for the Ivy Bridge machines where performance is the best configuration for energy efficiency for compute-bound benchmarks, our execution time increase is significantly smaller than our EDP improvement over performance. In fact, such execution times degradation can be so small that even if the system power is large compared to the CPU power, our DVFS approach still leads to significant overall energy savings.

3.6.5 Summary of Features

We conclude our experimental evaluation with Table 3.4 which reports how many benchmark belong to each category. This classification is done per binary, and is independent of the target platform or compiler used. A benchmark may have more than one characteristic features, which is why we use a decision tree to prioritize them as described in Sec. 3.5

We see that for the simple parallelized version, many codes are bandwidth-bound, and this decreases when using program transformations to improve data locality. Our static analysis is capable of adapting to the characteristics of the implementation of the benchmark, i.e., is robust to loop transformations applied to it. This is confirmed by Table 3.5 which zooms into a few benchmarks, displaying how their feature change depending on the structure of the code implementing the algorithm in the original benchmark.

In particular, as tiling improves the operational intensity, we see gemm move from being memory-bound (no tiling) to compute-bound (after tiling). On the other hand, for Jacobi-2D, tiling by itself did not improve the OI enough to make it go beyond our cut-off point, and it was expected as the best frequency / core configuration for this optimized kernel is indeed the best frequency for the bandwidth-bound codes: the program still suffers from bandwidth contention, even after the tiling we applied.

Benchmarks	seq/par	bw-bound	poor scale	comp-bound
polybench-parrallel	12/18	27	4	1
polybench-poly	5/25	15	1	10

Table 3.4: Summary of Features

Benchmarks	version	seq	bw-bound	poor scale	comp-bound
correlation	par			✔	
	poly				✔
gemm	par		✔		
	poly				✔
jacobi-2d	par		✓	✔	
	poly		✔		
seidel-2d	par	✔	✓		
	poly				✓

Table 3.5: Benchmark features

3.7 Phase Analysis

We now discuss the occurences of phases in the programs we evaluated, via a separate set of experiments conducted using the runtime algorithm described in Sec. 3.3. The

objective is to show that thanks to the stability of most affine kernels we evaluated, i.e., they contain only one phase, a purely static approach where we select a single frequency / core configuration for the entire kernel duration can achieve near-optimal results. We then discuss cases where phases occur, and point to future work on using Alg. 2 to detect those phases analytically to enable the selection of different frequency / core configurations for different phases of the program.

3.7.1 Phase Characterization

We plot in Fig. 3.8 the output of our adaptive runtime on several kernels. To better visualize phases, if any, we used a very low threshold for frequency change: as soon as there is a 1% energy efficiency difference between two time quanta (set to 50ms), the runtime is allowed to increase/decrease frequency. Benchmarks were set to using a very large problem size, for better illustration. There is one point per time quanta, and we report both the current frequency (in green), and the "instantaneous" power for the quanta (in purple). We have conducted this study on all 60 benchmarks on Haswell, using 4 cores, and isolated the most representative cases.

The top charts show typical power trace examples for single-phase kernels: the power consumption is mostly stable, and the frequency typically oscillate between two of the frequency points, e.g., 1.2GHz or 1.6GHz, indicating the optimal frequency may be in between these points. We conducted this characterization for all 60 benchmarks, and for 46 of them there is a single phase. These includes gemm and trmm for example. The bottom charts illustrate cases of multiple phases in the program. A total of 14 benchmarks show 2 or more phases (up to 5), visualized as a stable change for a part of the program execution of the frequency used. The effect of program transformations to improve data locality can

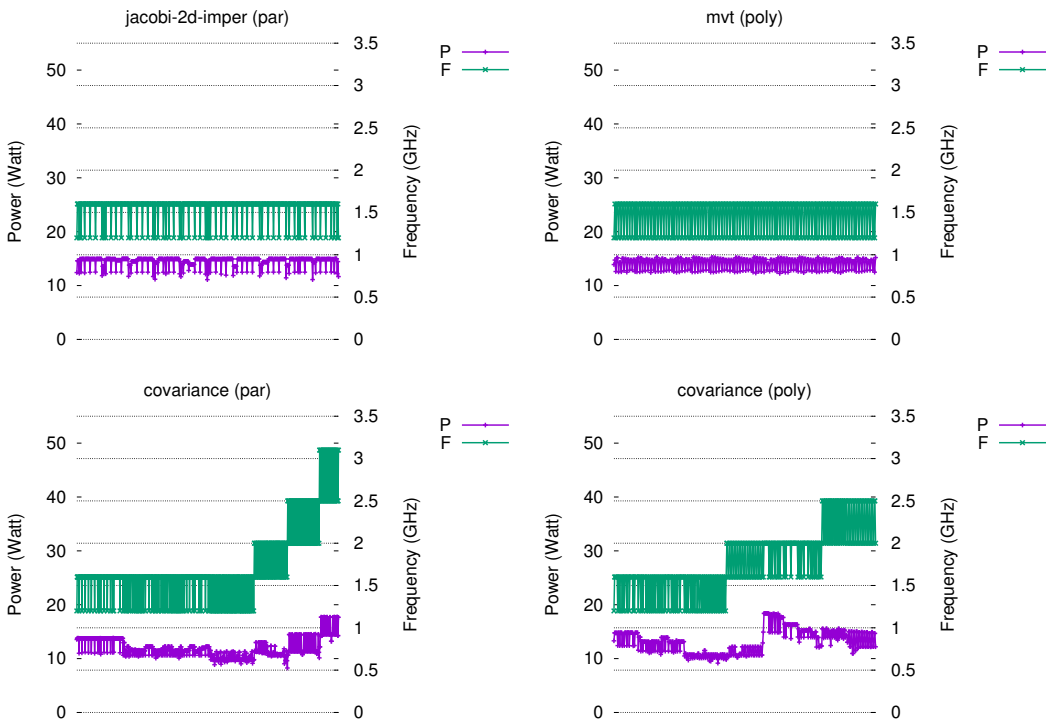

Figure 3.8: Different types of phases on Haswell / 4 cores

be visualized here: phases are not identical between both plots, in particular since data locality was changed by the polyhedral code transformation and therefore the operational intensity was modified.

3.7.2 Possible Improvements

As illustrated in Fig. 3.8, there can be cases with stable phases in the computation, necessitating ideally different frequency / core configurations for ideal energy savings. Algorithm 2 can be applied on any sub-part of the program, in particular, it can be applied on different loop nests to obtain the estimated operational intensity for different program regions. If this intensity differs, then the classification of the region using the decision tree from Sec. 3.5.2 may also change, leading to using different frequencies within the same

program, better suited to exploit its phases. As future work we will investigate how to embed such mechanism in our static analysis to detect at compile-time program phases.

3.8 Conclusion

Dynamic Voltage and Frequency Scaling (DVFS) is a well-known technique to adapt the power consumption based on the application demands and hardware state, by modifying the frequency (and the associated voltage) at which a processor operates. Previous works have focused on two aspects: how to reduce frequency without much wall-clock time penalty, to reduce power and energy used by a computation; and how to increase the frequency on demand to improve completion time and minimize overall energy-delay product.

In this work, we have demonstrated that there are inherent limitations to existing dynamic DVFS approaches, due to processor-specific and application-specific effects. We demonstrated that for a class of computations, namely affine programs, we can develop a *compile-time* categorization of programs by approximating their operational intensity and parallel scaling. It allows to automatically select at compile-time the best frequency and number of cores to use to optimize energy or energy-delay product, without needing any application profiling or runtime monitoring. Our evaluation on 60 benchmarks and 5 multicore CPUs validated our approach, obtaining energy savings of 10% and EDP improvements of 40% on average over the `powersave` Linux governor.

CHAPTER 4

Static Analysis of Hierarchical Set-Associative Cache Behavior for Affine Programs

4.1 Introduction

Optimizing compilers currently use simplified cost models to select the composition of loop transformations applied to programs. For example, Bondhugula et al. maximized idealized measures of data locality [34], Kong et al. maximized data locality under single instruction/multiple data (SIMD) constraints [78], and other works have examined combined objectives for parallelization and data locality (e.g., [48, 83]). While useful in practice, the major drawback of these approaches is that they are limited to abstract performance models, such as "maximizing parallelism" or "minimizing reuse distance," which are only indirectly related to concrete performance metrics, such as data cache misses.

Addressing these limitations requires several challenges to be addressed. First, approaches to accurately model cache behavior typically employ actual execution or simulation to measure or analyze the program's execution on the target device. These approaches can be expensive, with cost proportional to the number of instructions the program executes. For example, running the DineroIV [43] cache simulator on a simple matrix-multiply code for 1024×1024 matrices can take hours. Most importantly, these approaches do not

provide an exact model of cache behavior for use at compile time. Second, nearly all practical modern systems use cache hierarchies. A small, first-level (L1) cache is complemented with a larger second-level (L2) cache, and so on – the Intel Haswell CPU may contain up to four levels of caches. To accurately model the accesses to main memory, which are much slower than any cache accesses and can significantly influence program performance, the full hierarchy of caches must be modeled, where accesses to level L+1 are derived from the misses at level L. Third, most practical caches are set-associative rather than fully associative, where multiple virtual memory addresses may map to a specific set of physical cache lines, requiring accurate models of conflict misses.

In the quest for an approach that would enable compile-time analysis of runtime cache behavior, we need to: (1) model the cache hierarchy; (2) accurately model cold, capacity, and conflict misses of set-associative caches at each level; and (3) achieve an analysis time that is essentially independent of problem size and cache size. In this work, we achieve all of these goals by focusing specifically on the class of affine programs, where key properties of the control and data flows afford accurate capture of runtime cache behavior. We also propose a novel and purely static analysis to accurately compute cache behavior at compile time, without the need for execution or simulation.

Compile-time cache modeling (e.g., avoiding simulation) for affine programs has been studied in the past. For example, tile size selection has been investigated by approximating capacity misses for regular/affine data tiles, such as the distinct line model [107] and its generalization with the minimum line model [111]. Ghosh et al. [62, 61] proposed Cache Miss Equations (CME) to capture situations when a cache miss occurs in perfectly nested loop programs with uniform references. It handles set-associative caches, but only for single-level caches. However, a major limitation of the approach is the difficulty to translate

these equations into a cache miss count, which requires actually solving the CMEs (i.e., determining whether or not they have valid solutions) for a multitude of cases. Furthermore, translating a miss ***count*** at the first-level cache into a set of cache access for the next level is impractical. Thus, this approach is not suitable for effectively modeling hierarchical caches that are ubiquitous in practical systems.

Chatterjee et al. [39] made a significant advance by providing an exact closed-form solution for polyhedral programs for direct-mapped caches using Presburger formulae, which s partially generalized to model a set-associative single-level cache. However, none of these prior formulations of the cache miss problem are suited for modeling hierarchical caches. There is a fundamental difference between simply counting cache misses, as in prior work, and modeling the set of accesses that lead to a cache miss, as developed in this work. Such a set of events should be exactly modeled to compute misses in cache hierarchies because only the subset of accesses to L1 that misses in L1 form the set of accesses to L2 [20].

We make the following contributions:

- We present a closed-form solution to the problem of modeling cache behavior for arbitrary polyhedral programs executing on processors with set-associative virtually-indexed hierarchical caches under a variety of write policies.
- We develop a tool that computes the number of misses at each cache level for set-associative caches, from an input C source file containing affine program(s).
- We perform extensive evaluation of our cache miss analysis and validate it against a simulator with identical hardware assumptions, and present several approximation heuristics.

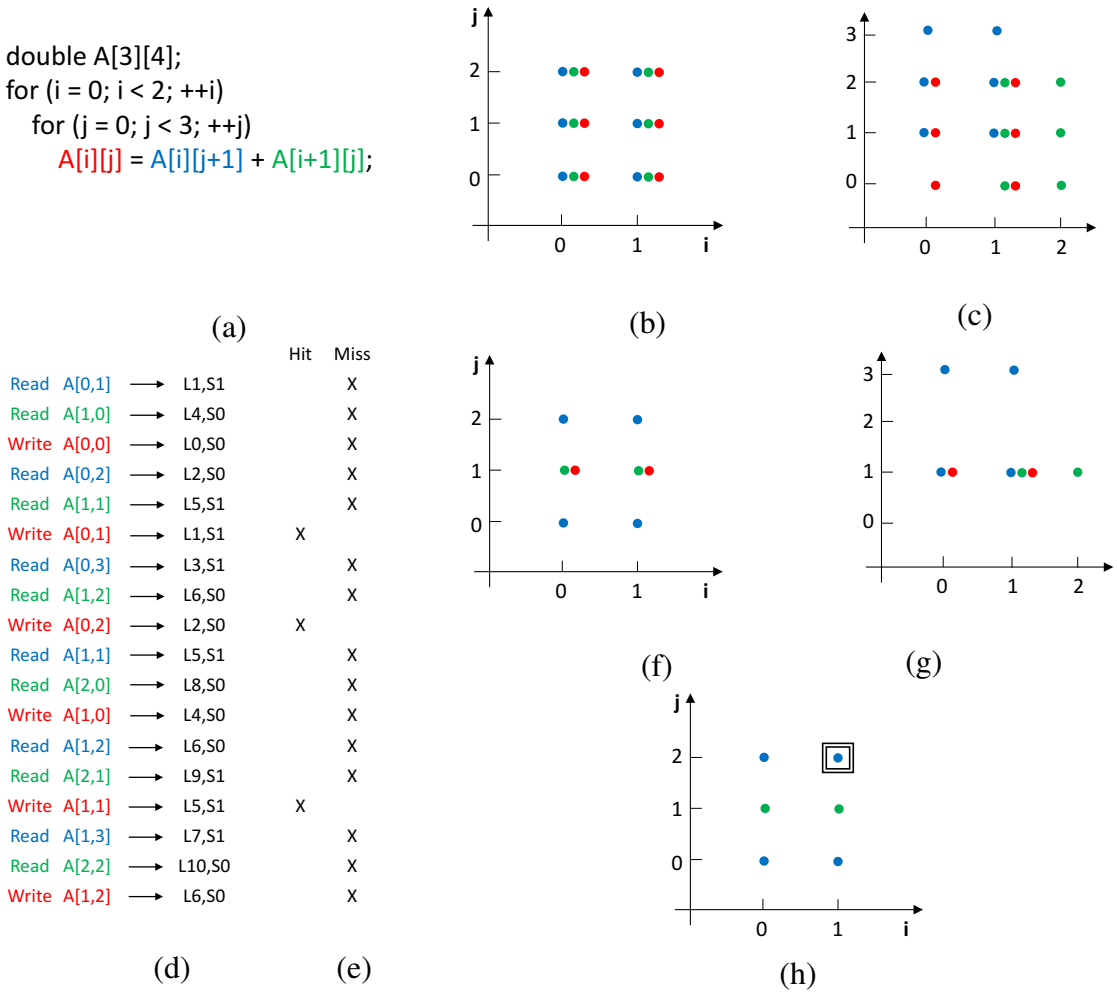

Figure 4.1: Illustrative example of a 2-way set-associative cache

4.2 Overview of Modeling Approach

Fig. 4.1 outlines the key steps in computing cache misses for a sample program (shown in Fig. 4.1(a)). Fig. 4.1(d) shows the execution trace of the various read/writes for execution of the program. The following includes a walk-through for computing the set of memory accesses that lead to cache misses in a simple 2-way set-associative cache.

A K-way set-associative cache is partitioned into cache sets, each of which can store K cache lines. A cache line stores B bytes of data. For a cache of size n bytes, the total number of cache lines, *nblines*, is given by n/B, and the total number S of cache sets is $(n/B)/K$. To determine the specific cache set that a memory address maps to, a classical and typical approach is to use the least significant bits of the memory address: a virtual memory address $addr$ maps to cache set $(addr/B)\%S$. Since the cache is K-way associative, up to K lines mapping to the same cache set can co-exist simultaneously in the cache. We illustrate this in Fig. 4.1(d) for a simple 2-way associative cache of size 32 bytes with 4 cache lines: $B = 8$, $K = 2$, and $S = 2$, that is each cache line stores a single element of `A`. The trace of accesses in Fig. 4.1(d) is typical of input traces provided to cache simulators, such as DineroIV [43]. First, the (annotated) program is executed to build the trace of accesses. Then, the simulator reads this trace in sequential order, one reference at a time. For each address accessed, it computes the mapped cache set and maintains an internal data structure representing the cache state (i.e., what data is currently in cache). When the accessed data item is not currently in cache, a cache miss occurs, possibly leading to the eviction of a cache line previously stored in this cache set, and the loading of the data into a line in this set from a copy in a later level cache (or main memory). Conversely, when the accessed data is already in cache, a cache hit occurs. Fig. 4.1(d) shows the cache line and cache set to which each address maps. The "virtual cache line" is computed by $L_{id} = addr/B$ and the cache set by $S_{L_{id}} = (addr/B)\%S$. We remark that $S_{L_{id}} = (addr/B)\%S = ((addr/B)\%nblines)\%S$ because we have $nblines = K * S$, $K \geq 1$. That is, L1,S1 is read as $L_{id} = 1, S_{L_{id}} = 1$ for this reference. Fig. 4.1(e) shows whether each access is a hit or miss for this cache configuration.

Consider a cache line L_{id} to be accessed. This access is a miss if it is the first time L_{id} is accessed or K distinct cache lines that map to the same set have been accessed more recently than L_{id}. Otherwise, it is a cache hit. Indeed, in a set-associative cache, each line maps to a specific set with K slots. A cold miss (an unavoidable miss even in an infinite cache) occurs the first time L_{id} is accessed. A capacity/conflict miss (one that can be avoided in an infinite cache) occurs if K different lines mapping to the same set have been accessed prior to (re)accessing L_{id}. In other words, for a particular cache set, all $K+1$ distinct accessed lines mapping to this set will lead to a cache miss.

Focusing on Set 1 in the 2-way set-associative cache, Fig. 4.1(d) shows that first `A[0,1]` and then `A[1,1]` are loaded in Set 1 (they are both cold misses), and `A[0,1]` then is accessed again (it is a hit: no line has been evicted from Set 1 yet). When `A[0,3]` is loaded, Set 1 is already full, and the least recently used (LRU) line is evicted. `A[1,1]` is deleted, and `A[0,3]` is stored in the cache. A cache simulator such as DineroIV operates exactly the same way. Following the sequence order in the program trace, it determines where a reference maps, using L_{id} and $S_{L_{id}}$, and keeps track of the current cache state to determine if an access is a hit or a miss, performing evictions as needed, e.g., based on the LRU policy. The total number of misses is reported, and, for hierarchical caches, accesses that lead to a miss in L1 cache are used as accesses to L2 cache, etc.

In this chapter, we describe a compile-time analysis intended to produce the same output as a cache simulation, i.e., the set of accesses that lead to cache misses, ***without actually generating the program trace or simulating cache state***. We achieve this by restricting the class of modeled programs to ***polyhedral programs*** (defined in Sec. 4.3). In a nutshell, for polyhedral programs, the exact ordered set of all accessed memory locations can be described in a compact form using integer sets and relations, as presented in Sec. 4.3. From

this example, we observe that the following are needed for such an analysis: (1) the ordered set of memory accesses (capturing the same information as the trace in Fig. 4.1(d)), (2) a mechanism to determine which line and set an address is mapped to (modeling exactly Fig. 4.1(d)'s mapping), and (3) the ability to determine sequences of accesses to $K+1$ distinct lines mapping to the same set. Item (1) above is a direct consequence of polyhedral programs. The exact ordered set of accesses can always be built for these programs. For (2), the L_{id} and $S_{L_{id}}$ expressions can be modeled as Presburger formulae if n, B and S are known at compile time, as [39] showed. Achieving (3) is a core contribution of this chapter (Sec. 4.4).

We illustrate the approach with Fig. 4.1(f)-(h), using the previously described code and cache configuration. First, a polyhedral representation of the program is generated. The set of executions of the statement is captured by a polyhedron, Fig. 4.1(b) plots this iteration domain of the program. Each dot corresponds to an execution of the associated color-coded reference, and the coordinates of the dot capture the value of the corresponding loop iterators. The program execution order (the schedule) can always be determined. In Fig. 4.1(b), it corresponds to the lexicographic ordering.

For each array reference (e.g., `A[i][j+1]`), an access function from each iteration point to the accessed memory cell is built (e.g., $\mathbf{F^A} : \{[i,j] \rightarrow [i,j+1]\}$). Fig. 4.1(c) displays the set of memory cells from the array `double A[3][4]` that are accessed by this program. Fig. 4.1(c) is obtained from Fig. 4.1(b) by applying the access functions to the iteration domain.

Fig. 4.1(g) corresponds to the ***subset*** of cells from `A` that map to Set 1. This can be obtained by adding constraints so that only accesses with $S_{L_{id}} = 1$ are kept, i.e. $(addr/4)\%2 == 1$, where $addr$ is the accessed memory address. $addr$ is obtained from $\mathbf{F^A}$ by linearizing

the access function. We do require the array sizes to be known and constant, to obtain $\{[i,j] \rightarrow [A + i * 4 + j + 1]\}$. We remark that because the hit/miss pattern of a cache set is not impacted by the accesses to other cache sets, we can model accesses to each cache set independently and aggregate the results for all cache sets at the end. Fig. 4.1(f) displays the same subset of references that map only to Set 1, by showing the subset of points in the iteration domain that correspond to these references. This pruned iteration domain forms the (ordered) set of references to Set 1 only, obtained in a manner similar to how Fig. 4.1(g) is built, but reasoning on iteration points instead.

The last and most critical step is to obtain Fig. 4.1(h), the subset of only the accesses that lead to a miss (cold and capacity/conflict). This is built in two stages. First, we determine the subset corresponding to cold misses, which is computed as the first access to a unique cache line (unique L_{id}). Given a function **IterToLine** that maps iteration points to the virtual cache line L_{id} they access (as defined in Sec. 4.4), the first access to each L_{id} is computed by taking the lexicographically first (with respect to the program schedule) iteration accessing L_{id}, i.e., the preimage of $lexmin(\textbf{IterToLine})$. This can be computed analytically for polyhedral programs. In Fig. 4.1(h), these accesses correspond to the dots ***not*** inside a square.

Second, we must determine the ordered sequence of virtual lines L_{id} accessed by the program, where for a particular set $S_{L_{id}}$, every $K+1$-th accessed line mapping to that set will incur a cache miss, and thus must be captured. In Fig. 4.1(h), it corresponds to the dot inside the square. We solve this problem by building several functions, typically from an iteration point to another (set of) iteration point(s) with some specific properties. For example, the composition $\textbf{SameSet} = \textbf{IterToSet} \circ \textbf{IterToSet}^{-1}$ maps an iteration point $\vec{i}$ to the other iteration points $\{\vec{j}\}$ accessing the same cache set, some of which will

be miss events. We successively refine **SameSet** to end up with only the iterations $\{\vec{j}\}$ which do incur a cache miss. Iterations mapping to the same virtual line are computed as **SameLine** = **IterToLine** $\circ$ **IterToLine**$^{-1}$. We can combine **SameSet** and **SameLine** to keep only the iterations that correspond to accesses to distinct cache lines within a set. For a K-way associative cache, we then successively remove the second, third, ..., K-th unique line accessed mapping to the same cache set and retain only the $K+1$-th ones, i.e., the collection of miss events. This is achieved by building a function from a point in an iteration domain to the immediately next executed point in this domain. This function can be composed with others to get the immediately next ***relevant*** iteration for our analysis, e.g., the next iteration accessing the same cache set and a different cache line. We proceed by successively removing such next iterations, K times, to end up with only the iterations incurring a miss, as detailed in Sec. 4.4.

The resulting set (e.g., Fig. 4.1(h)) has two fundamental properties: it can be counted to obtain the number of cache misses, and be used to model accesses to the next level cache, since misses at L1 form the set of accesses at L2. By composing this process, we can seamlessly reason on hierarchical set-associative caches and produce per-level cache miss count, paving the way for simulation-free compile-time analysis and optimization of cache behavior for polyhedral programs (discussed in Sec. 4.9).

4.3 Program Representation

In this work, we target the compile-time analysis of polyhedral program regions [48, 63], also called ***affine*** programs. A program region is polyhedral if the only control-flow structures are `for` loops and `if` conditionals; the iteration set of each syntactic statement

can be described by static analysis using affine inequalities of the surrounding loop iterators and program region parameters (unknown but constant values for one execution of the region); the access functions of arrays are multidimensional affine functions of the surrounding loop iterators and parameters; each `for` loop has loop-invariant loop bounds, a constant scalar stride, and the exit value of the loop iterator is not read outside the loop; and syntactic statements may contain a data-dependent conditional (e.g., using a ternary operator in C), but both the true and false branch will be assumed to be always taken in the analysis.

Polyhedral programs have essential properties that we rely on to build our static analysis (Sec. 4.3.2). This program class covers a wide spectrum of compute- and data-intensive processing kernels typically found in linear algebra methods, image processing, or physics simulation [63, 18, 17] where high performance is a must and, consequently, exploiting cache hierarchies effectively is highly desired. We first describe the key mathematical objects manipulated in this work and their operations. We follow with presenting the representation of affine programs using these structures.

4.3.1 Modeling Integer Tuples

All mathematical structures and operations used in this work involve modeling integer tuples using Presburger formulae to model sets and relations among them. Operations on these structures are readily available in the Integer Set Library (ISL) [124], including more advanced operations, such as counting the number of points in a set or relation [128]. All structures and operations are briefly surveyed in this work. As integer sets are manipulated, the worst-case complexity of most operations is NP-hard. However, in practice, quasi-polynomial time is often achieved.

Integer tuple A point in a set is a multidimensional integer vector, or integer tuple. Its n components take value in $\mathbb{Z}$: $\boldsymbol{i} \in \mathbb{Z}^n$.

Integer sets A set S of integer tuples is a subset of $\mathbb{Z}^n$ defined as:

$$S = [p_1, ..., p_p] \rightarrow \{[i_1, ..., i_n] : c_1 \wedge ... \wedge c_m\}$$

where $i_1, ..., i_n$ index the n dimensions of the set (noted $\vec{i}$); $p_1, ..., p_p$ are invariant parameters (noted $\vec{p}$); and $c_1, ..., c_m$ are m Presburger formulae typically in the form of affine inequalities defining constraints on the values of $\vec{i}$. Presburger formulae in this work involve existential quantifiers, modulo operations, integer division, and ceil/floor.

Integer sets are not polyhedra. They can contain "holes," i.e., they can model the intersection of a polyhedron with an integer lattice, for example, to model a subset containing only some even integers.

Relation A relation $\mathbf{R}$ maps points $\mathbb{Z}^n \rightarrow \mathbb{Z}^m$ and is defined as:

$$\mathbf{R} = [p_1, ..., p_p] \rightarrow \{[i_1, ..., i_n] \rightarrow [j_1, ..., j_m] : c_1 \wedge ... \wedge c_o\}$$

where $\vec{i}$ are the input dimensions and $\vec{j}$ the output dimensions. Similarly to sets, Presburger formulae are used to express a relation between points in two sets of possibly different dimensions but also to express constraints on the input and output sets.

Domain and range of a relation To retrieve the set of points where a relation can be applied (its domain) and the set of points that may be produced (its range), use:

$$\text{domain}(\mathbf{R}) = \{[i_1, ..., i_m] : \boldsymbol{i} \text{ s.t. } \exists \boldsymbol{j},\ \boldsymbol{j} = \mathbf{R}(\boldsymbol{i})\}$$

and:

$$\text{range}(\mathbf{R}) = \{[j_1, ..., j_n] : \boldsymbol{j} \text{ s.t. } \exists \boldsymbol{i},\ \boldsymbol{j} = \mathbf{R}(\boldsymbol{i})\}$$

Composition Two relations $\mathbf{R}_1 : \mathbb{Z}^n \to \mathbb{Z}^m$ and $\mathbf{R}_2 : \mathbb{Z}^m \to \mathbb{Z}^o$ can be composed to form a new relation:

$$\mathbf{R}_1 \circ \mathbf{R}_2 = \{[i_1, ..., i_n] \to [k_1, ..., k_o] : \exists \boldsymbol{i} \in \text{domain}(R_1),\ \exists \boldsymbol{k} \in \text{range}(R_2)\ :\ \boldsymbol{k} = \mathbf{R}_2(\mathbf{R}_1(\boldsymbol{i}))\}$$

Inverse A relation can be inversed or reversed, i.e., the input and output dimensions are reversed. For a relation $\mathbf{R}$ as defined previously, its inverse is:

$$\mathbf{R}^{-1} = [p_1, ..., p_p] \to \{[j_1, ..., j_n] \to [i_1, ..., i_m] : c_1 \wedge ... \wedge c_o\}$$

Application A relation can be applied to a set, which results in a set of points that are the image of the input set S by the relation $\mathbf{R}$:

$$S_{\mathbf{R}} = \mathbf{R}(S) = \{\boldsymbol{x} \in S : (\exists \boldsymbol{y} \in S \wedge (\boldsymbol{y} \to \boldsymbol{x}) \in \mathbf{R})\}$$

Union and intersection We manipulate union, intersection, and difference of sets and relations. These binary operations are written $\cup$ for union, $\cap$ for the intersection, and $-$ for the difference.

Counting Finally, an essential operation we extensively rely on is the ability to build a counting formula for an integer set (or relation). This formula takes the form of an Ehrhart quasi-polynomial and can be computed using the Barvinok algorithm implementation in ISL [128, 1]. We note this operation $\#S$ for the cardinality of a set S.

4.3.2 Representing Programs

Programs with affine data flow and static control flow are called ***static control parts*** (SCoP) [48, 63]. Polyhedral programs are represented in this work using (a union of) integer sets and integer relations. Three key structures are needed in this work to define a program. For all statements in the program, we capture its iteration domain, data access relations, and schedule of iterations. We use the illustrative example in Fig. 4.2.

```
  for (i = 0; i < M; i++)
     for (j = 0; j < N; j++) {
        for (k = i+1; k < M; k++)
R:          B[i][j] += A[k][i] * B[k][j];
S:       B[i][j] = alpha * B[i][j];
      }
```

Figure 4.2: Triangular Matrix-Multiply

Iteration domains Iteration domains capture the set of runtime executions of a statement. Because programs are polyhedral, this set can be exactly captured using integer sets where the loop bounds are used to constrain the number of points in the set. Each statement R is associated with an iteration vector $\vec{i_R}$ with one component per surrounding loop, and the values $\vec{i_R}$ can take are captured by defining its iteration space $\mathcal{D}_R$. For example, the iteration domain of `R` in Fig. 4.2 is:

$$\mathcal{D}_R = [M,N] \rightarrow \{R[i,j,k] : 0 \leq i < M \wedge 0 \leq j < N \wedge i+1 \leq k < M\}$$

It is possible to count the number of points in this set with:

$$\#\mathcal{D}_R = (-1/2 * M + 1/2 * M^2) * N, \text{ with } M \geq 2 \wedge N > 0.$$

We note *ProgDomain* the union of all per-statement iteration domains, i.e., the set of all iteration domains for the entire program.

Data access functions An essential part of our analysis is based on representing the data accessed by each program iteration. For polyhedral programs, the function that maps a statement instance to the array cell being accessed is, by definition, an affine relation, involving surrounding loop iterators and parameters. We will distinguish the read and write references, and the access relation maps an iteration domain to the multidimensional array index being accessed. For example, the function that relates the iterations of R with the location read in array A for the reference $A[k][i]$ in Fig. 4.2 is:

$$\mathbf{Read}^{\mathrm{A}}_{\mathrm{R}} = \{R[i,j,k] \mapsto A[i_1,j_1] : (i_1 = k) \wedge (j_1 = i)\}$$

We note **Write** for the write references, and the set of all read and/or write access functions for the program is obtained by building the union of the per-statement data access relations. Furthermore, it also is possible to build the relation restricted to the set of iterations of R by computing $\mathbf{R} = \mathbf{Read}^{\mathrm{A}}_{\mathrm{R}} \cap \mathcal{D}_R$, i.e., embed the constraints on the possible values for $R[i,j,k]$ directly in the relation. The set of R iterations then can be recovered by computing domain($\mathbf{R}$) or the set of array cells accessed by computing range($\mathbf{R}$).

Finally, we note **ReadRefs**, the union of all **Read** access relations for the program; **WriteRefs**, the union of all **Write** relations in the program; and the union of all access relations for the program **ProgRefs** = **ReadRefs** $\cup$ **WriteRefs**.

Program execution order A schedule is a relation used to specify the execution order of all statement instances. It maps points in the iteration domain to those in an integer set (the set of timestamps). As such, statement instances in the iteration domain are executed following the lexicographic ordering $\prec$ of their associated timestamp. $\prec$ is defined as $(a_1,\ldots,a_n) \prec (b_1,\ldots,b_m)$ iff there exists an integer $1 \leq i \leq min(n,m)$ s.t. $(a_1,\ldots,a_{i-1}) = (b_1,\ldots,b_{i-1})$ and $a_i < b_i$.

The original program schedule is modeled using $2d+1$ timestamps, where d is the maximal nesting depth in the program [63]. For example, the schedules of R and S in Fig. 4.2 are:

$$\textbf{Sched}_R = \{R[i,j,k] \mapsto [0,i,0,j,0,k,0]\}$$
$$\textbf{Sched}_S = \{S[i,j] \mapsto [0,i,0,j,1,0,0]\}$$

where each odd dimension of the output space is a scalar dimension whose value denotes the lexical abstract syntax tree (AST) ordering of the loops surrounding the statement. For statements surrounded by less than d loops, the even schedule components associated with the missing loops is set to 0. This approach seamlessly models imperfectly nested loops. A schedule can be constrained by the iteration domain of its statement, e.g., via $\textbf{Sched}_R \cap \mathcal{D}_R$. Consequently, the set of all distinct statement iterations in the program can be built by making the union of all schedules constrained by their respective statement iteration domain. **Sched** denotes this union.

4.4 Single-Level Cache Analysis

We begin in this section by modeling accesses to a single-level set-associative cache and then extend to hierarchical caches in Sec. 4.5. For a set-associative cache with associativity K and S sets, for each data read/write executed in the program, we represent the specific set in the cache to which the corresponding cache line L is mapped. If it is the first time L is accessed, the access is a miss. It is also a miss if K distinct cache lines that map to the same set have been accessed more recently than L. Otherwise, the access is a cache hit. Indeed, in a set-associative cache, each cache line maps to a specific set, which contains K slots. By reasoning on the sequence of events (i.e., data read/write events in the program) that lead to access of distinct cache lines mapping to the same set, we can build the set of events

corresponding to the $K+1$ distinct line accessed for a set, i.e., the set of events leading to a cache miss. This information can be built in a closed-form for affine programs.

4.4.1 Modeling Cache Accesses

To model events corresponding to accessing different cache lines, we must first translate array indices given by access relations into unique cache line indices and their associated set in the cache. We assume least-recently-used (LRU) replacement policy and define a cache and the mapping of virtual memory address to cache elements as follows.

Definition 3 (Set-associative cache)**.** *A set-associative cache C with associativity K, cache line (i.e., block) size of B bytes, and size n bytes contains S sets, with* $S = n/B/K$*. A virtual memory address addr maps to a unique line index* $L_{id} = floor(addr/B)$*, and the line maps to a unique set* $S_{L_{id}} = L_{id}\%S$*.*

We now assume that all array accesses have been linearized. This is always possible if the array extents are known at compile time. For example, the linearization transformation from a two-dimensional access relation $\mathbf{R}^A$ for 2D array A with size sz along the fastest varying dimension (i.e., number of columns of A for row-major linearization as used in C) and start_A as its starting address is:

$$\textbf{Linearize} = \{[i,j] -> [m] : m = \text{start}_A + i * sz + j\}$$

Generalizing to n-dimensional arrays is straightforward.

The value of B and K are assumed to be known at compile time, which means the value of S also is a numerical constant. Given a virtual memory address, the unique cache line index is given by applying the relation:

$$\textbf{MemToLineId} = \{[m] -> [lid] : lid = \text{floor}(m/B)\}$$

The set to which a cache line maps is given by the relation:

$$\textbf{LineIdToCacheSet} = \{[lid] -> [cset] : cset = lid \ \% \ S\}$$

It follows the definition of the relation from an array access function to a cache set.

Definition 4 (Array to Cache set index)**.** *Given an access function* $\mathbf{F}^A$ *to array A of size* $\vec{sz}$ *and starting address* $start_A$ *for a cache as defined in Def. 3, the associated cache line in C is identified by* **AccessToLine** *as:*

$$\textbf{AccessToLine} = \mathbf{F}^A \circ \textbf{Linearize}(\vec{sz}, start_A) \circ \textbf{MemToLineId}$$

The associated cache set in C is identified by **AccessToSet** *as:*

$$\textbf{AccessToSet} = \textbf{AccessToLine} \circ \textbf{LineIdToCacheSet}$$

With these relations, we can now reason in the cache space used by the program. For example, the set of distinct cache lines accessed in the program is modeled as:

$$Clines = \text{range}(\textbf{Sched}^{-1} \circ (\textbf{ProgRefs} \circ \textbf{AccessToLine}))$$

and can be counted immediately with #*Clines*. This expression corresponds exactly to computing, in a general form and for arbitrary affine programs, the Distinct Line (DL) expression used in prior work for tile size selection [107, 111]. Notably, it is significantly simpler than the original DL formulation, with no loss of accuracy.

4.4.2 Miss Events in Set-Associative Caches

Equipped with the ability to reason on cache lines and cache sets being accessed, we can now model ***if*** a particular access is a cache miss by reasoning on the sequence of accesses to distinct lines mapping to the same set.

Modeling the next iteration Intuitively, we want to model consecutive (in terms of program execution order) accesses to the same cache set. Thus, we need to model a notion of

sequencing in program execution by using the program schedule **Sched**. Specifically, we want to capture the point(s) $\boldsymbol{j}$ of the program iteration domain that is executed immediately after point $\boldsymbol{i}$ such that $\boldsymbol{i}$ and $\boldsymbol{j}$ have some properties, such as accessing the same cache set but a different cache line. Given an iteration $\boldsymbol{i}_1$, $\boldsymbol{i}_2$ is consecutive to $\boldsymbol{i}_1$ if there is no iteration $\boldsymbol{i}_3$ in between. The non-existence of such point $\boldsymbol{i}_3$ can be equivalently expressed using relations and set/relation differences.

The relation **LexSucc** maps a point to all points that are executed after it. We have:

$$\begin{gathered}\textbf{LexSucc} = \{[i_1,...,i_n]\rightarrow[j_1,...,j_n] : (\boldsymbol{j}) \text{ s.t.} \\ \exists \boldsymbol{i}, \boldsymbol{j} \in \mathit{ProgDomain} \wedge \textbf{Sched}(\boldsymbol{i}) \prec \textbf{Sched}(\boldsymbol{j})\}\end{gathered}$$

Similarly, we build **LexSuccEq**, the relation that generates points that are executed after an input point, including it; and **LexPrec**, the relation that generates points executed prior to an input point. Note these relations are each specifically built with respect to the particular program we are analyzing.

Modeling iterations accessing the same line/set To model cache misses, we start by modeling a relation from the iteration domain to the cache line indices and cache sets:

$$\begin{aligned}\textbf{IterToLine} &= (\textbf{Sched} \cap \mathit{ProgDomain})^{-1} \circ \textbf{ProgRefs} \circ \textbf{AccessToLine} \\ \textbf{IterToSet} &= \textbf{IterToLine} \circ \textbf{LineIdToCacheSet}\end{aligned}$$

These relations associate to each point of the iteration space the set of cache lines (and cache sets) it accesses. To reason about iterations accessing the same lines or sets, we use the relation inversion to build a map from iterations to iterations accessing the same cache line/set, i.e.:

$$\textbf{SameLine} = \textbf{IterToLine} \circ \textbf{IterToLine}^{-1}$$

Modeling sets of relevant iterations Relations such as **LexSucc** are meant to be intersected with other relations/sets that encode specific properties. For example, one can model the relation from an iteration ***i*** to all iterations accessing the same set ***that are executed after i*** as:

$$\textbf{SameLineSuc} = \textbf{SameLine} \cap \textbf{lexPrec}$$

A complete procedure is built by assembling the set of iterations that access different cache lines for each different set, retaining only the ones leading to the $K+1^{th}$ distinct line accessed in a particular set, as this incurs a miss, as described below.

4.4.3 Algorithm for Miss Calculation

Algorithm 5 provides the complete procedure to obtain the set of iterations incurring in a cache miss event for a set-associative cache, as well as the count of these events.

Set specialization Notably, building a formulation for all cache sets at once is unnecessary. To manipulate simpler systems, it is possible to embed an additional constraint, e.g., in **LineIdToCacheSet**, where the set is fixed to a constant value, e.g., $cset = x$, where $x \in \mathbb{N}$ is a known constant. Then, an iterative algorithm can be built, specializing SingleLevelMisses for each set value $S_i : [0, S]$ and forming the union of all $Miss_i$ sets to form the complete set of all misses for all cache sets.

4.5 Cache Writing Policies and Hierarchical Caches

In the preceding section, we modeled accesses to cache sets of a single-level cache from read and write operations in an affine program. To model a multi-level cache, we need to determine the read and write operations that are performed on the next level of the cache.

Algorithm 5 SingleLevelMisses: Compute Misses for Single-level Cache

Input: Program: **ProgRefs**, *ProgDomain*, **Sched**
Array parameters: $(\vec{sz}_{Array}, start_{Array})\ \forall\ Array$
Cache parameters: B, S, K

Output: Set of iterations incurring a cache miss
miss count

// Apply schedule to iteration domain.
Prog = Sched $\cap$ *ProgDomain*
// Iterations to cache line indices:
IterToLine = Prog^{-1} $\circ$ **ProgRefs** $\circ$ **AccessToLine**
// Iterations to cache sets:
IterToSet = IterToLine $\circ$ **LineIdToCacheSet**
// Compute cold misses, i.e., first access to a line.
coldmiss = range (lexmin (IterToLine) $\circ$ **Sched**$^{-1}$)
// Iterations to iterations accessing the same cache line:
SameLine = IterToLine $\circ$ **IterToLine**$^{-1}$
// Keep only output iterations lexicographically after the input iteration:
SameLineSuc = SameLine $\cap$ **lexPrec**
// Keep only output iterations lexicographically equal or after to the input iteration:
SameLineSucEq = SameLine $\cap$ **lexSuccEq**
// Iterations to iterations accessing the same cache set:
SameSet = IterToSet $\circ$ **IterToSet**$^{-1}$
// Keep only output iterations lexicographically after the input iteration:
SameSetSuc = SameSet $\cap$ **lexPrec**
// Starting relation: all accesses to a cache set:
Miss = SameSetSuc $-$ (**SameLineSuc** $\circ$ **lexSuccEq**)
for all assoc in 2, K **do**
    Next = Miss $-$ (**Miss** $\circ$ **lexPrec**)
    // Build the next iteration relation:
    Miss = Miss $-$ **Next** $\circ$ **SameLineSucEq**
end for
Misses = *coldmiss* $\cup$ range(**Miss**)
return (*Misses*, #Misses)

Table 4.1: Handling of write operations by the different cache writing policies considered

	Write-through	Write-back
Write allocate	All writes are cached. No dirty evictions. All writes forwarded to next cache level.	All writes are cached. All writes lead to dirty evictions. Only dirty evictions are seen as writes at next cache level.
No-write allocate	Write hits are cached. No dirty evictions. All writes forwarded to n ext cache level.	Write hits are cached. Write hits can lead to subsequent dirty evictions. Write misses and dirty evictions forwarded to next cache level.

The reads and writes that arrive at a cache at level $L+1$ depend on the policies employed by the cache at level L.

Thus far, we have modeled caches that manage reads and writes in a symmetric fashion. In this section, we model caching approaches that employ alternative strategies to manage the write operations. When a cache encounters a write operation, it can: (a) ***Write-through***: immediately forward the write to the next cache level, or (b) ***Write-back***: write to the cache and forward to the next level only if the cache line is evicted.

In addition, a write to a line not in the cache (a write miss) can be handled in two ways: (a) ***Write allocate***: allocate a line in the cache, read in the current contents of that cache line, and then perform the write in cache, or (b) ***No-write allocate***: do not allocate a line but forward the write operation to the next cache level.

These policies change the cache contents at any point in time, impacting the cache hits, misses, dirty cache lines, and action taken on evictions. Table 4.1 describes the various scenarios.

Write-allocate write-through policy The cache miss model for this policy is exactly the same as that described in Algorithm 5. All misses at cache level L with write-allocate, write-through policy become reads to the cache at level $L+1$. All writes to such a cache, irrespective of whether they hit or miss in the cache at level L, become writes to the next cache level.

Write-allocate write-back policy The cache miss model for this policy is exactly the same as that discussed in Algorithm 5. All misses at cache level L with write-allocate, write-back policy become reads to the cache at level $L+1$. Evictions of dirty cache lines become writes to the next cache level. This is illustrated in Algorithm 6. The algorithm takes as input the **Miss** relation computed in line 13 of Algorithm 5, the relation from the iteration that allocates a cache line to the nearest following iteration that evicts it.

Algorithm 6 Compute Dirty Evictions: Writes To Next Cache Level For Write-Back Caches

Output: Dirty evictions

//evictions to writes that follow the immediate preceding allocation in execution order

1: **SameLineWriteSuccEq** = (**IterToLine** $\circ$ **WriteIterToLine**$^{-1}$) $\cap$ **lexSuccEq**

2: **A** = **Miss**$^{-1}$ $\circ$ **SameLineWriteSuccEq**

//miss iteration to all iterations that causally follow it

3: **B** = (range **Miss**) $\cap$ **lexSuccEq**

//dirty evictions to writes that made them dirty

4: **C** = **A** $-$ **B**

//dirty evictions to last write to before the eviction

5: **D** = **lexmax C**

//iteration of dirty eviction to the evicted cach line

6: **return** (**D** $\circ$ **IterToLine**)

No-write-allocate write-through policy A cache with this policy caches all read operations and only the write operations that result in a cache hit. In particular, while write

misses do not affect the cache state, a write hit updates a cache line's priority, affecting subsequent evictions under LRU replacement policy. Therefore, modeling the misses for a no-write-allocate cache requires that we first model the write hits, and thus, in turn, write misses. To overcome this challenge, we present an approximate solution. We employ a cache miss formulation that is the same as in Algorithm 5, except that only the read references (**ReadRefs**) are used in formulating the cache misses. All read misses at cache level L become reads to the cache at level $L+1$. All writes to a no-write-allocate write-through cache become writes to the next cache level.

No-write-allocate write-back policy We approximate the cache misses using the same strategy as with a no-write-allocate write-through cache. We compute the read misses using only **ReadRefs**. All read misses at cache level L become reads to the cache at level $L+1$. Write miss operations are forwarded to the next cache level. We compute the write misses using the Algorithm 5 but instead to compute K distinct reads between a write and its immediately preceding read. Specifically, to compute the write misses, line 5 in Algorithm 5 is changed to:

$$\textbf{SameLine} = \textbf{IterToLine} \circ \textbf{WriteIterToLine}^{-1}$$

where **WriteIterToLine** considers only iterations involving write operations. Evictions of dirty cache lines (Algorithm 6) and write misses become writes to the next cache level.

4.6 Cache Modeling Across Program Phases

We have presented an analysis of cache misses for an affine program executed with an initial cold cache (all cache lines are invalid). Programs can consist of affine and non-affine phases. Our modeling approach can be used in the context of a whole program analysis

framework for the affine program phases, with a conventional trace-based simulation being used for the non-affine phases. To enable such a "hybrid" analysis, we need to adapt our modeling approach to produce the actual final state of the cache at the end of an affine phase, so that it can be provided to a conventional cache simulator to model the subsequent non-affine program phase.

4.6.1 Final Cache State

The final cache state is defined by the set of cache lines in the cache, with information on whether or not they are dirty, and the recency order among the lines in each set. This information about the final cache state after the affine phase is needed to ensure correct simulation for cache accesses in the subsequent non-affine phase using standard cache simulation. Given a cache of associativity K, the cache lines resident in the set S in the final cache state can be computed as the last K distinct accessed cache lines that map to S.

Cache lines in final cache state Algorithm 7 shows the steps involved in computing this information for a write-allocate cache.

It starts with computing the last iteration ($\mathbf{A_0}$) that accesses each cache set using the lexicographic maximum operation. Then, all iterations that access the same cache line as this last iteration are removed from the set of all program iterations. This procedure is repeated until K iterations are computed. The union of all these cache lines (identified in terms of the iterations that access them) defines the cache lines in the final cache state. Note that this procedure works even if fewer than K distinct cache lines are accessed for some cache sets. Write-allocate caches allocate cache lines for both read and write accesses. Thus, the algorithm is invoked with the set of all program references (**ProgRefs**) to compute the final state.

Algorithm 7 FinalAccess(Refs)

Input: Program: **Refs**, *ProgDomain*, **Sched**
Array parameters: $(\vec{sz}_{Array}, start_{Array})\ \forall\ Array$
Cache parameters: B, S, K

Output: For each cache set S, compute the references in **Refs** (identified by their schedule time) that access the last K distinct cache lines mapping to S

// Apply schedule to iteration domain
1: **Prog = Sched** $\cap$ *ProgDomain*
// Iteration to cache line:
2: **IterToLine = Prog**$^{-1}$ $\circ$ **Refs** $\circ$ **AccessToLine**
// Iteration to cache set:
3: **IterToSet = IterToLine** $\circ$ **LineIdToCacheSet**
//Cache set to all iterations that access that set
4: **SetToIter = IterToSet**$^{-1}$
//Iteration i to all iterations that access the same cache line as i
5: **SameLine = IterToLine** $\circ$ **IterToLine**$^{-1}$
6: **tmp = SetToIter**
7: **for** assoc in 1, K-1 **do**
8: $\mathbf{A}_{assoc}$ = **lexmax tmp** //Compute the last assoc-th access to each cache set
//Remove the assoc-th access from the list of accesses to be considered
9: **tmp = tmp** $-$ ($\mathbf{A}_{assoc}$ $\circ$ **SameLine**)
10: **end for**
11: $\mathbf{A_K}$ = **lexmax** (**tmp**)
//Relation from each cache set to the iterations that access the last K distinct cache lines
12: **FinalAccess** = $(\bigcup_{i=1}^{K} \mathbf{A_i})$
13: **return FinalAccess**

Given the relation returned by the algorithm, the actual lines in the cache are given by:

$$\mathit{FinalCacheLines} = \text{range}(\textbf{FinalAccess} \circ \textbf{IterToLine})$$

Dirty cache lines In the case of write-through caches, no cache lines need to be marked as dirty because all write operations are also reflected in the next cache level. In the case of write-back caches, cache lines that were written to after they were allocated in the cache for the last time need to be marked as being dirty in the final cache state. This computation is performed as shown in Algorithm 8.

Algorithm 8 Determine Dirty Cache Lines In The Final Cache State For Write-Back Caches

Output: Set of dirty cache lines in the final cache state

```
    //Determine the Misses associated with each cache line
1:  LineToMisses = (IterToLine)^-1 ∩ Misses
    //Determine the last miss for each line in the final cache state
2:  FinalAllocations = lexmax (LineToMisses ∩ FinalCacheLines)
    //Compute all write iterations that follow a given iteration
3:  SameLineWriteSuccEq = (ItertoLine ∘ (WriteIterToLine^-1)) ∩ lexSuccEq
    //Compute the domain of relation that compute all final misses that have a following write
4:  DirtyCacheLines = domain (FinalAllocations ∘ SameLineWriteSuccEq)
5:  return DirtyCacheLines
```

We now have the collection of dirty and non-dirty cache lines in the final cache state. The program's schedule gives us the last iteration that accesses each line in the final cache state. Accessing the cache lines in the final cache state—reading (writing) non-dirty (dirty) cache lines—will reproduce the final cache accesses in program order, preserving the LRU characteristics.

Together, the cache lines, dirty markers, and recency information in the cache final state mirror those that would be observed in an actual program execution.

4.6.2 Initial Cache State

The cache behavior of an affine program phase is dependent on the initial cache state when starting the phase. The cache state at any point can be encoded with an affine program that, if executed, would lead to such cache state. When composing two affine program phases, The final cache state of the first phase can be represented as an affine program to be executed before the second affine phase. This will correctly evaluate the cache behavior.

For arbitrary initial cache states, this program representation can be as large as the cache and expensive to manipulate in the form of integer sets. In this scenario, the first K distinct cache lines to be accessed in each set can be computed analogous to computing the final cache state. We enumerate each such access to check if it is a hit or a cold miss. The remaining accesses are modeled as presented in preceding sections.

The overall procedure to combine simulation of arbitrary program phases and the affine program phase involves the following steps:

1. Formulate a model of the first access to K distinct cache lines to each set in the affine phase.

2. For each such initial access, determine if it is a hit or a miss based on the initial cache state.

3. Model the affine phase using the integer set formulation.

4. Reconstruct the final cache state (Sec. 4.6.1).

We note that while the affine phase is modeled in a problem- and cache-size independent fashion, the interface between the affine and non-affine phases incurs costs proportional to the cache size.

4.7 Experimental Evaluation

We now perform extensive evaluation of our framework, and present comparison with a trace-based simulator for validation purpose.

4.7.1 Experimental Setup

The presented approach to cache modeling has been implemented in the PolyCache tool, which takes as input a C program and information about the cache parameters and size and starting address of arrays. PolyCache outputs the cache miss count of the program, and was used to generate all results presented in this chapter. It is implemented using ISL-0.17 [5] (with barvinok-0.38 and pet-0.07), an integer set library for the polyhedral model. The Polyhedral Extraction Tool (PET) [126], a powerful SCoP extractor on top of Clang, detects affine regions and extracts the polyhedral model from C source code. ISL [124] is used to perform the operations in the algorithms described in previous section. We use the set-specialization approach discussed in Sec. 4.4.3 to compute the analysis for each set independently, and then accumulate the results as appropriate. These computations are run in parallel, using one process per set in the cache, since each set's behavior is independent of the others.

4.7.2 Evaluation of Hierarchical Set-Associative Cache

Benchmarks We evaluate PolyCache's performance with the PolyBench/C benchmarking suite [9], which contains key numerical kernels and programs written as SCoPs. We also select some scientific computing benchmark kernels from HPGMG [10], which is a representative high-performance computing benchmark based on geometric multigrid methods, and CCSD(T) kernels from the computational chemistry suite NWChem [119].

Table 4.6 and 4.7 shows the various codes (64 in total) that were evaluated. We employed the standard dataset size provided for PolyBench/C, which typically leads to a data footprint exceeding 1 MB. The number of data access operations range from millions to billions. These benchmarks provide a variety of challenges, including out-of-cache dataset sizes, multiple arrays, imperfectly nested loops, triangular loops, non-uniform reuse distance, etc.

Tools and setup As a comparison point for both correctness and performance of PolyCache, we used the DineroIV simulator [43], a uniprocessor cache simulator that can handle hierarchical set-associative caches, as well as numerous replacement and write policies. All experiments were performed on a cluster of Intel Xeon E5640 processors running at 2.67 GHz with 32 KB L1 cache. The programs were all compiled using a GCC-6.1.0 compiler with -O3 optimization. We parallelized the PolyCache computation across sets, as mentioned above, using as many processes as sets in the cache and report the time to completion.

Evaluation The first experiment aims to validate the correctness of the framework against a trace-based simulator for a real-life complex scenario: a 2-Level cache memory hierarchy with 32 KB 4-way set-associative L1 cache and 256 KB 4-way set-associative L2 cache, both implementing a write-allocate write-back policy. Both caches have 64-byte cache line size. Table 4.6 and 4.7 compare the cache accesses/misses computed by DineroIV with the accesses/misses obtained with PolyCache, as well as the execution time (in seconds) for both systems. The number of data read/writes issued by the program is not reported herein. Instead, we focus solely on the cache access/miss count.

A key observation is that ***for all situations, there is a perfect match between DineroIV and PolyCache***. We performed this validation for all experiments reported in this chapter, we always obtained an exact match in the number of accesses/misses reported by DineroIV and PolyCache for the non-approximated schemes.

To illustrate the intricacy of the situations modeled by our analytical framework, we detail results for the Floyd-Warshall benchmark. Of note, the total number of misses in L2 is actually slightly higher than the number of misses in L1, 64,000 higher exactly. This is the expected result, but it results from a complicated effect. First, cold misses are the same for L1 and L2. Capacity misses are also the same. Floyd-Warshall uses a matrix of size 1024×1024, and the reuse distance between two iterations of the outer loop is larger than both the 32- and the 256-KB cache. Moreover, there is nearly no conflict miss either in L1 or L2 for this experiment due to the large number of capacity misses. Thus, cache misses for L1 and L2 are expected to be mostly the same, yet there are 64,000 more misses in L2. In the 2-level cache hierarchy with write-back policy implemented for both L1 and L2, a line evicted from L1 is written to L2 to maintain coherency between the caches, and a line evicted from L2 is written back to memory. Writing a line evicted from L1 in L2 is a cache hit (write hit) only if the line is already in L2. Otherwise, it will generate an L2 miss (write miss). Furthermore, it may lead to evicting a line in L2 if the set to which this line maps to in L2 is already full with other lines. Therefore, events in L1, such as dirty line eviction, may result in a cache miss in L2. In fact, there are 64,000 lines written in Floyd-Warshall (using a matrix of size 1024×1024), which corresponds exactly to the additional 64,000 misses in L2.

We note that there is significant variability in execution time across benchmarks and methods. As expected, for a trace-driven simulator, the DineroIV time is proportional to

the trace size. For example, the 2mm benchmark comprised of a sequence of two gemm matrix multiplication takes about twice as much time to simulate than gemm (there are about a billion operations in gemm). Our analytical approach is much faster, up to 365× faster in this case. However, there exists some cases (2 over 64, shown in bold in Table 4.6 and 4.7) where the compile-time approach is slower than the simulation. Unfortunately, predicting *a priori* the execution time of PolyCache is infeasible. By design, we operate on integer linear programs, where even a simple polyhedron emptiness test is NP. However, for many typical cases the observed complexity is polynomial in practice. This issue is addressed in greater detail in Sec. 4.8.

4.7.3 Evaluation of Write Policies

As discussed in Sec. 4.5, the framework is capable of handling exactly various write-allocate policies. In PolyCache, we have implemented the write-allocate write-back policy used in Table 4.6, and evaluate below our implementation for the no-write-allocate write-back policy. Table 4.2 compares the cache misses computed by PolyCache (L1 and L2 columns) with DineroIV (% diff. columns, -0.1% means PolyCache under-approximated the misses by 0.1%). We use L1 and L2 caches of same size and associativity as in Table 4.6, but both using instead the no-write-allocate-write-back policy.

We make three observations. First, as the formulation is significantly more complex for this write policy, the execution time of PolyCache significantly increases compared to Table 4.6. In fact, 8 benchmarks (2mm, cholesky, durbin, fdtd-apml, lu, reg-detect, symm and trmm) timed out, that is PolyCache did not complete the calculation within 120 minutes, our timeout for this experiment. Second, for most benchmarks in our experimental setup, there is 0% difference in the cache miss count computed by PolyCache compared

Table 4.2: Summary of No-Write Allocate Write Back Cache

Benchmark	Cache Misses				Time (sec)	
	L1	L1 %diff	L2	L2%diff	Dinero	PolyCache
3mm	3224954880	0%	3225139200	0%	8663	16
adi	39376448	0%	39066432	0%	633	107
atax	4194560	0%	1049600	0%	54	8
bicg	3523070	0%	1049856	0%	57	3
correlation	2097280	0%	2097408	0%	5	199
covariance	1067548736	0%	1067582080	0%	2024	34
doitgen	270828544	0%	263168	0%	525	8
dynprog	71540106	0%	60921211	0%	737	4
fdtd-2d	22928000	0%	22928000	0%	299	30
floyd-warshall	67107968	0%	67104232	-0.1%	2823	583
gemm	1074984960	0%	1075046400	0%	2869	2
gemver	21736183	0%	18882296	0%	209	41
gesummv	3645950	0%	2097920	0%	56	18
gramschmidt	268706560	0%	268706248	-0.003%	465	171
jacobi-1d	2124600	0%	11248	0%	2	26
jacobi-2d	44398240	0%	44398240	0%	79	4
ludcmp	371409006	0%	365825035	0%	592	318
mvt	17844214	0%	17832188	0%	170	14
seidel-2d	1310720	0%	1310720	0%	73	2
syr2k	166298694	0%	134468056	-0.1%	3487	7720
syrk	67173444	0%	67169874	0%	1685	764
trisolv	530944	0%	527488	0%	12	2
chebyshev	409900120	0%	409900120	0%	1003	3135
heat	630612960	0%	630612960	0%	2573	324
minigmg	659660352	0%	659660352	0%	803	776
poisson	630874080	0%	630874080	0%	4650	3650
j3d7pt	630612960	0%	630612960	0%	1953	208
j3d13pt	3179606544	0%	3179606544	0%	3780	1664
j3d27pt	630874080	0%	630874080	0%	5986	4749
ccsd_d1_1	269549568	0%	2531989	0%	438	3
ccsd_d1_6	286261248	0%	1798021	0%	534	4
ccsd_d2_1	2162688	0%	1444382	0%	398	3
ccsd_d2_9	17845248	0%	2466976	0%	405	3
ccsd_s1_1	1114128	0%	1114128	0%	27	2
ccsd_s1_9	1052688	0%	1052688	0%	27	2

to DineroIV's output. That is, for these experiments, approximating the no-write-allocate by only considering read references does not change the miss count at either L1 or L2.

This highlights that ignoring write-hits on LRU in these experiments does not change the miss count. Third, only three benchmarks (shown in bold) shows differences between our approximation and DineroIV: PolyCache under-approximates the miss count by less than 0.1%. Note all CCSD benchmarks showed 0% difference between PolyCache and DineroIV, we only display representative ones.

4.7.4 Evaluation of Loop Tiling

We now report cache miss data when loop tiling is applied to the programs. We process all 30 Polybench benchmarks with the PoCC compiler [8], using the Pluto algorithm [34] to automatically compute a complex sequence of loop transformation to make loop tiling legal and then apply loop tiling wherever valid, using a tile size of 32. This transformation can dramatically increase the code size with up to hundreds of syntactic statements and tens of loop nests in the generated code. Table 4.3 reports the number of misses without and with this tile size selection for a single-level 32 KB, 4-way set-associative L1 cache. All of the cache miss numbers were validated for correctness against DineroIV.

In two cases (Table 4.3, in bold) the execution times are longer than simulation, for example, fdtd-apml, where it takes almost 1 hour to complete. The reason is the same as explained earlier. Due to the inherent difficulty of predicting when this analytical formalism will lead to prohibitive execution time, we advocate a timeout-based approach. Users may set the timeout to a few minutes. If no answer is provided, users can, for example, fall back to use of approximation heuristics (refer to Sec. 4.7.5).

4.7.5 Evaluation of Approximation Heuristics

Here, we discuss possible acceleration techniques with the goal of reducing static analysis time, while allowing for an approximate result. Thus, the methods presented in this

Table 4.3: Summary of Tiling (PolyCache is Poly. below)

Benchmark	L1 Cache Misses		Time (sec)	
	No tiling	Naive tiling	Dinero	Poly.
2mm	2149969920	4362076160	4184	60
3mm	3224954880	519479636	6195	485
adi	39376448	454822592	663	385
atax	4194560	4194560	57	3
bicg	3523070	18530940	67	20
cholesy	11495238	11495301	161	11
correlation	2097280	4100	982	38
covariance	1067548735	539130944	983	19
doitgen	270828544	1327104	463	80
durbin	34089213	34152702	102	91
dynprog	46880104	47070106	746	10
fdtd-2d	22928000	15187943	**333**	**436**
fdtd-apml	5481573	250113688	**741**	**3403**
floyd-warshall	67107968	99645071	3123	176
gemm	1074984960	71303168	2000	6
gemver	21621493	22011647	121	50
gesummv	3645950	33604605	70	13
gramschmidt	268706048	268713488	288	52
jacobi-1D	250000	5000	20	16
jacobi-2D	5235200	4198090	350	330
lu	22647007	370408773	766	72
ludcmp	370932158	34195872	656	83
mvt	17844214	18531068	68	23
reg-detect	300	300	4	2
seidel	1310720	2152592	77	14
symm	1607465303	1607465299	3858	148
syr2k	166298694	2161626400	3875	219
syrk	67173444	1077905600	1879	62
trisolv	530688	535167	13	13
trmm	34110526	49805889	872	318

section do not guarantee exact miss count, is in contrast to the results presented earlier that are exact by definition.

Per-array analysis We have developed a simple heuristic that analyzes only the references to a single array in the program at a time, and the process is repeated for all arrays.

In other words, we only capture self-interference in this process. Table 4.8 reports the number of misses computed for each array in the program, where for programs with more than four arrays, we have summed the number of misses obtained for all subsequent arrays. We compare the sum obtained by approximation of misses in the program based solely on self-interference, versus the exact number of misses in the full program, as well as the execution time of both this heuristic and the full program analysis with PolyCache. The average speedup of the analysis time is 8x, and, in many situations, there are no cross-interference misses between the various arrays in the benchmark.

However, for benchmarks such as bicg, only considering self-interference is not a valid approach. In this case, we compute capacity misses because some arrays can fit into the cache separately but not all together, resulting in the difference in cache miss numbers.

Sampling cache sets We also evaluate another heuristic based on the key observation that there is often a high similarity between the misses across different sets, especially for regular applications. Table 4.4 summarizes our set-0 heuristic. This heuristic is limited to computing the misses for the cache set-0 and estimates the total misses by multiplying the resulting value by the number of sets (128 in this experiment). We show several metrics on the number of misses for all of the sets, such as the minimum, maximum, average, and standard deviation, and report the miss count error (% difference) and execution time of the heuristic compared to the execution time of PolyCache. Note an essential difference in the setups between the set-0 heuristic and full program analysis. For the set-0 heuristic, we compute only one set, meaning PolyCache is executed on a single core without any parallelism. In contrast, for "Time all sets," the time reported is the time for PolyCache to complete when using S-way parallelism, i.e., using S cores.

Table 4.4: Summary of Set-0 Heuristic

Benchmark	Min	Max	Avg.	Stdev	Set-0 Approx.	% diff.	Set-0	All Sets
2mm	16795648	16796672	16796640	179	2149842944	0%	7	8
3mm	25193472	25195008	25194960	268	3224764416	0%	6	6
adi	306803	358054	307629	6375	39270784	0%	56	57
atax	32770	32770	32770	0	4194560	0%	4	4
bicg	27522	27524	27524	0	3523072	0%	3	3
cholesy	552	262641	89807	79529	33618048	192%	6	7
correlation	12602	20168	16385	2225	2581504	23%	16	19
covariance	8336440	8344009	8340224	2225	1068033152	0%	7	11
doitgen	2115584	2116608	2115848	450	270794752	0%	4	4
durbin	134303	396386	266322	77049	17190784	-50%	14	33
dynprog	4	650001	366251	188950	23040000	-51%	4	4
fdtd-2d	179100	179150	179125	25	22931200	0%	6	10
fdtd-apml	32832	68796	42825	10183	5197312	-5%	318	2657
floyd-warshall	524281	524281	524281	0	67107968	0%	15	22
gemm	8397824	8398336	8398320	89	1074921472	0%	1	1
gemver	168910	168918	168918	1	21621120	0%	23	23
gesummv	28482	28484	28484	0	3645696	0%	2	3
gramschmidt	2099204	2099328	2099266	37	268698112	0%	11	18
jacobi-1D	1800	2000	1953	85	256000	2%	1	1
jacobi-2D	40900	40900	40900	0	5235200	0%	3	3
lu	8128	262639	176930	77374	1040384	-95%	3	5
ludcmp	323544	4202981	2897907	1166399	536934272	45%	54	70
mvt	139403	139408	139408	1	17843584	0%	3	4
reg-detect	0	4	2	1	512	71%	0	1
seidel	10240	10240	10240	0	1310720	0%	1	1
symm	4280917	20824324	12558323	4859581	550904576	-66%	50	83
syr2k	1298961	1299472	1299209	256	166267008	0%	12	17
syrk	524792	524794	524793	1	67173504	0%	2	3
trisolv	2114	6178	4146	1187	790784	49%	2	3
trmm	8441	524535	266488	151925	67140352	97%	4	5

First, we observe a variety of benchmarks where the set-0 heuristic provides actually near-exact results. These illustrate regular cache access patterns. Conversely, the approach may dramatically fail, such as for LU, where the number of misses is under-approximated by 20x. Yet, these result show a surprisingly good approximation can be obtained very quickly. For all but two (LU and symm), the cache misses are off by a factor of 2 or less

with 18/30 benchmarks giving nearly exact value. We believe these results pave the way for additional focused experimental studies of the regularity of cache accesses for affine programs with the goal of determining when the cache behavior may be approximated effectively by the behavior of a single (or few) set(s). As future work, we will investigate sampling heuristics, computing the value of a few randomly chosen sets and using their average miss instead of remaining limited to set-0.

4.8 Time Complexity of PolyCache

Table 4.6 shows very large execution time variation across benchmarks for PolyCache. The explanation is simple to state but the execution time is infeasible to characterize *a priori*. In this chapter, we rely extensively on manipulating sets modeled by Presburger formulae, in particular, on counting the number of points in these sets. This computation is NP-hard. As such, we obviously see situations where a much more complex system (i.e., miss set) is generated and counting takes a very long time, despite an excellent implementation from the Barvinok library [128, 1] integrated in ISL. To the best of our understanding, it is impossible to predict *a priori* the execution time of PolyCache because it relates to the irregularities in the generated cache miss set, not to simple input program features, such as the number of loops or problem size.

In this section, we provide details on two representative cases, Gemm and Symm. While both have roughly similar numbers of cache misses and similar DineroIV execution times (shown in Table 4.6), for Symm, PolyCache is 150× slower to compute the solution than for Gemm. For each operator type invoked by PolyCache, Table 4.5 details the number of times this operator is used (Nb Ops) and the total time to execute the Nb Ops calls.

Table 4.5: Per-operator Execution Time
s: set, m: map, x: set or map, p: integer

Operator	Syntax	Execution time (sec)			
		Gemm	No. Ops	Symm	No. Ops
Domain	$s := \text{domain } m$	0.005	3	0.080	3
Range	$s := \text{range } m$	0.000	4	0.001	4
Composition	$m3 := m1 \circ m2$	0.206	28	7.657	28
Inverse	$m2 := m1^{-}1$	0.058	6	0.132	6
Application	$s2 := m(s1)$	0.001	1	0.024	1
Union	$x3 := x1 \cup x2$	0.005	2	0.021	2
Intersection	$x3 := x1 \cap x2$	0.062	6	1.697	6
Difference	$x3 := x1 - x2$	1.136	16	113.030	16
Coalesce	$s2 := \text{coalesce } s1$	0.717	12	218.142	12
Counting	$p := \text{card } x$	0.081	2	1.518	2
Total time		2.273		342.303	

For both computations, the operators are called exactly the same number of times. This is expected, given that it follows Algorithm 5 using the same cache configuration, but it highlights the fact that the number of operator calls does not determine execution time. Interestingly, we see that the time taken for set difference explodes for Symm. This is due to the shape of the sets being subtracted. The result is not a convex set but a union of small, distinct convex sets (pieces). For Symm, the number of pieces explodes, indicating high irregularity in the Miss set. The coalesce operation similarly explodes. Coalescing is a form of simplification of the representation, and tries to combine smaller pieces into a larger convex piece. It is invoked prior to counting (the final operation of our algorithm), i.e., on the result of the difference operation. Intuitively, the execution time appears to be driven by how complicated the intermediate polyhedral sets we manipulate are, which itself strongly depends on the cache configuration being modeled. Because predicting when

PolyCache's execution time will be very high is impractical, the approximation techniques outlined herein can prove critical to quickly compute a solution.

4.9 Discussion and Pratical Uses

Discussion and limitation We have presented a closed-form formulation of the cache behavior for hierarchical set-associative caches. The formulation is independent of the problem and cache sizes. However, the formulation is influenced by the associativity of the cache being modeled. The integer set formulation has an exponential worst-case complexity [93]. As observed in the preceding sections, this can occasionally result in very high evaluation costs.

A study of exact cache behavior requires mapping the array addresses to linearized memory addresses. While polyhedral approaches can deal with parametric problem sizes, this linearization requirement forces the use of constant array sizes. Therefore, similar to all prior work addressing this problem, the exact set of misses must be computed for each constant array size.

In addition to linearized memory addresses, our formulation requires that the function mapping an address to a cache line be affine. This restricts it to virtually indexed caches. Several architectures (e.g., SPARC, PowerPC, ARM) employ virtually indexed caches for portions of their cache hierarchy, enabling direct use of our formulation.

Even with physically indexed caches where the virtual-to-physical mapping is determined at runtime, our approach can be useful when applications employ huge pages (e.g., 1 GB pages). This mapping, obtained after executing the program, can be extracted as a trace. Given the small number of virtual and physical pages when using huge pages, tracing this mapping is much cheaper than tracing every cache miss. Given a mapping extracted

from a run, one can construct the virtual address to cache set mapping using piece-wise affine functions (one per huge page) and reconstruct the cache behavior for that run using our formulation.

Cache vulnerability analysis Cache vulnerability factor (CVF), the probability that soft errors (e.g. bit flips) impacting the cache get propagated to other hardware components, was introduced by Zhang et al. [135] to evaluate the reliability of the cache hierarchy. Its calculation involves tracking, for each bit, the fraction of program execution time during which a bit flip may escape the cache into an architecturally visible state (registers or memory). For example, corruption of a dirty cache line that is written back to memory leads to an escaping error, whereas a non-dirty cache line that is just replaced can mask bit flips. Vulnerability calculation requires exact information about the operations on each data word of the cache—reads, writes, misses, evictions—throughout program execution, including the time when each event occurs. This is typically achieved using expensive simulations, where the lifetime of cache lines can be precisely monitored via hooks in the simulator. However, it is usually too expensive for practical use in design space exploration to optimize vulnerability. Instead, our formulation can be extended to perform detailed lifetime analysis of each cache line so as to characterize vulnerability of the entire cache hierarchy at compile time.

Array padding Inter-array padding has been shown to impact cross-interference, or, conflict misses between arrays, and can be impact by the padding between arrays (e.g., [104, 68]). The inter-array offset can be cast as a linear parameter that needs to be determined to minimize the cache misses.

Cache design analysis Our formulation enables two optimizations when exploring cache configurations by varying its associativity (cache lines per set) and number of sets, for a given input program. First, as seen in Algorithm 5, an analysis of an input program for a K-way cache with a given number of sets incorporates an analysis of caches with the same number of sets but with lower associativity. Therefore, only the max-associativity for a given number of sets needs to be evaluated. On the other hand, a simulation may have to consider all combinations of associativity and number of sets. Second, given configuration for a cache at level L, our approach can present the reads and writes that reach the cache at level $L+1$. This affords reuse of the cache evaluation at level L to explore the choice of cache configurations at level $L+1$. Together, these can significantly reduce the cost to explore the space of possible cache configurations.

Bounding worst-case execution time The execution time of programs executing on real-time systems is bounded to guaranteed response times [16]. Cache-based systems complicate this analysis by introducing unpredictable data access latencies [14]. Our exact analysis of cache misses can assist in more accurate determination of cache miss costs and, thus, better bounds on the execution times.

4.10 Conclusion

Cache behavior analysis is an essential tool for program optimization. Optimizing the data traffic for cache-equipped processors has been investigated both via software transformations and hardware cache design space exploration. However, the current state of practice to analyze cache behavior is limited to actual execution and/or cache simulation, both with time complexity proportional to the number of executed operations, making compile-time optimization of cache behaviors difficult.

In this chapter, we proposed a fully compile-time approach to static analysis of cache behavior for hierarchical set-associative virtually-indexed caches for the class of affine (polyhedral) programs. The framework was validated by implementation of a tool to analyze affine C programs to compute cache misses at any level, producing cache statistics equivalent to trace-based cache simulators.

Since the framework makes extensive use of complex polyhedral operations with NP-hard worst-case complexity, long analysis times remain possible. As future work, we plan to investigate additional acceleration schemes for the formulations, as well as the development of compiler analyses and optimizations driven by cache miss estimations.

Table 4.6: Benchmarks and Summary of Hierarchical Set-Associative Cache(a)

Benchmark	Description	Cache Accesses		Cache Misses		Time (sec)	
		L1	L2	L1	L2	Dinero	PolyCache
2mm	2 Matrix Multiplications	8589934592	2150100992	2149969920	2150092800	5780	16
3mm	3 Matrix Multiplications	12884901888	3225151488	3224954880	3225139200	8633	27
adi	Alternating Direction Implicit solver	1571123200	59063040	39376448	39066432	639	78
atax	Matrix Transpose Vector Multiply	134217728	5243392	4194560	1049600	54	4
bicg	BiCGStab Linear Solver Kernel	134217728	4571902	3523070	1049856	56	3
cholesy	Cholesky Decomposition	359486976	12019059	11495238	11339977	153	52
correlation	Correlation Computation	9437184	2097408	2097280	2097408	13	13
covariance	Covariance Computation	2156921856	1068171390	1067548735	1067582079	2096	38
doitgen	Multiresolution kernel (MADNESS)	1077936128	271090688	270828544	263168	527	4
durbin	Toeplitz system solver	67149812	50878969	34089213	34010918	182	128
dynprog	Dynamic programming (2D)	1727040000	91000172	46880104	36361199	729	9
fdtd-2d	2-D Finite Different Time Domain	732979500	32752000	22928000	22928000	302	25
fdtd-apml	Anisotropic Perfectly Matched Layer	765734400	8251741	5481573	4241215	**382**	**2859**
floyd-warshall	Shortest paths for weighted graph	7516192768	134215936	67107968	67166336	2829	64
gemm	Matrix-multiply C=a.A.B+b.C	4294967296	1075050496	1074984960	1075046400	2905	2
gemver	Vector Multiply &Matrix Addition	234893312	22793716	21621493	18881787	210	38
gesummv	Scalar, Vector& Matrix Multiply	134230016	3900412	3645950	2097920	57	7
gramschmidt	Gram-Schmidt decomposition	537133568	335954432	268706048	268714496	502	29
jacobi-1D	1-D Jacobi stencil computation	5998800	375000	250000	1250	2	1
jacobi-2D	2-D Jacobi stencil computation	167117440	7851520	5235200	5235200	70	4
lu	LU decomposition	1431130624	45261695	22647007	22671492	562	5
ludcmp	LU decomposition	719498750	371973118	370932158	365399675	577	229
mvt	Matrix Vector Product& Transpose	134217728	17844726	17844214	17832188	188	6
reg-detect	2-D Image processing	11656000	596	300	300	4	2
seidel	2-D Seidel stencil computation	208896800	2618880	1310720	1310720	75	1
symm	Symmetric matrix-multiply	3215988736	2143794634	1607465303	1603222155	3463	323
syr2k	Symmetric rank-2k operations	8592031744	166429766	166298694	134537936	3498	151
syrk	Symmetric rank-k operations	4294967296	67238980	67173444	67169874	1677	39
trisolv	Triangular solver	33566720	530944	530688	527104	13	3
trmm	Triangular matrix-multiply	2145386496	34175998	34110526	34106357	838	36

Table 4.7: Benchmarks and Summary of Hierarchical Set-Associative Cache(b)

Benchmark	Description	Cache Accesses		Cache Misses		Time (sec)	
		L1	L2	L1	L2	Dinero	PolyCache
chebyshev	Chebyshev smoother in HPGMG	2228640704	536867608	471319352	471319352	**1018**	**2861**
heat	2-D heat equation stencil	5836644000	166594560	133301760	133301760	2507	298
minigmg	Compact Geometric Multigrid	7341404672	3703992592	721079584	782498816	820	710
poisson	Poisson solver stencil computation	14856912000	166855680	133562880	133562880	4577	1890
j3d7pt	3-D Jacobi stencil with 7 points	4244832000	166594560	133301760	133301760	1901	195
j3d13pt	3-D Jacobi stencil with 13 points	1573158144	786627840	3179606544	3179606544	3731	390
j3d27pt	3-D Jacobi stencil with 27 points	10612080000	166855680	133562880	133562880	5844	963
ccsd_d1_1	NWchem tensor contraction kernel1	1073741824	270598144	269549568	2777765	444	2
ccsd_d1_2	NWchem tensor contraction kernel2	1073741824	271581184	270532608	2263970	439	2
ccsd_d1_3	NWchem tensor contraction kernel3	1073741824	287309824	286261248	2587673	449	2
ccsd_d1_4	NWchem tensor contraction kernel4	1073741824	270598144	269549568	1723455	537	2
ccsd_d1_5	NWchem tensor contraction kernel5	1073741824	271581184	270532608	1505958	529	2
ccsd_d1_6	NWchem tensor contraction kernel6	1073741824	287309824	286261248	1830622	534	2
ccsd_d1_7	NWchem tensor contraction kernel7	1073741824	270598144	269549568	1784865	494	2
ccsd_d1_8	NWchem tensor contraction kernel8	1073741824	271581184	270532608	1552533	484	2
ccsd_d1_9	NWchem tensor contraction kernel9	1073741824	287309824	286261248	1877197	491	2
ccsd_d2_1	NWchem tensor contraction kernel10	1073741824	3211264	2162688	1444382	397	2
ccsd_d2_2	NWchem tensor contraction kernel11	1073741824	18939904	17891328	1447441	418	2
ccsd_d2_3	NWchem tensor contraction kernel12	1073741824	18954304	17905728	1754040	415	2
ccsd_d2_4	NWchem tensor contraction kernel13	1073741824	3211264	2162688	1489164	397	2
ccsd_d2_5	NWchem tensor contraction kernel14	1073741824	18939904	17891328	1508608	418	2
ccsd_d2_6	NWchem tensor contraction kernel15	1073741824	18954304	17905728	1813508	407	2
ccsd_d2_7	NWchem tensor contraction kernel16	1073741824	3149824	2101248	2101248	399	2
ccsd_d2_8	NWchem tensor contraction kernel17	1073741824	18886144	17837568	2464140	411	2
ccsd_d2_9	NWchem tensor contraction kernel18	1073741824	18893824	17845248	2708672	410	2
ccsd_s1_1	NWchem tensor contraction kernel19	67108864	2162704	1114128	1114128	28	1
ccsd_s1_2	NWchem tensor contraction kernel20	67108864	2162704	1114128	1114128	27	1
ccsd_s1_3	NWchem tensor contraction kernel21	67108864	2162704	1114128	1114128	27	1
ccsd_s1_4	NWchem tensor contraction kernel22	67108864	2162704	1114128	1114128	27	1
ccsd_s1_5	NWchem tensor contraction kernel23	67108864	2162704	1114128	1114128	28	1
ccsd_s1_6	NWchem tensor contraction kernel24	67108864	2162704	1114128	1114128	28	1
ccsd_s1_7	NWchem tensor contraction kernel25	67108864	2101264	1052688	1052688	27	1
ccsd_s1_8	NWchem tensor contraction kernel26	67108864	2101264	1052688	1052688	27	1
ccsd_s1_9	NWchem tensor contraction kernel27	67108864	2101264	1052688	1052688	28	1

Table 4.8: Summary of Per-Array Heuristic

Benchmark	Number of Cache Misses						Time (sec)		Speedup
	Array 1	Array 2	Array 3-...	Sum	Actual	% diff.	Sum	Full	
3mm	65536	1073741824	2147811327	3221618688	3224954880	0%	1	6	5.3×
adi	13053312	13107648	13107712	39268672	39376448	0%	17	57	3.4×
bicg	1048576	256	768	1049600	3523070	-70%	0	3	10.9×
covariance	1066991616	64	557023	1067548703	1067548735	0%	3	11	4.1×
dynprog	156	44800000	156	44800312	46880104	-4%	1	4	3.8×
gemm	65536	1073741824	65536	1073872896	1074984960	0%	0	1	2.9×
gemver	18874368	256	1792	18876416	21621493	-13%	1	23	17.3×
jacobi-2d	2618880	2616320		5235200	5235200	0%	1	3	3.1×
ludcmp	370931709	64	128	370931901	370932158	0%	17	70	4.1×
trisolv	526336	256	256	526848	530688	-1%	0	3	5.8×

CHAPTER 5

Cache Vulnerability Analysis for Affine Programs

5.1 Introduction

Soft errors becomes more and more important due to the exponential growth rate along with the development of technology. Among all the hardware components within processor, memory is the one that most vulnerable to soft errors because of the number of transistors and large area [88]. Moreover, memory at lower level can be protected via Error Correcting Code (ECC). However, for higher level memory, which are close to CPU, it will result in much higher overhead if protecting using ECC. Therefore, cache, as one of the most vulnerable components to soft errors, has been widely studied to characterize and reduce its vulnerability.

The typical approaches to determine the cache vulnerability is through actual execution or cycle accurate simulation to measure or analyze the program execution and cache behaviors on the target platform. However, these approaches are expensive as the cost scales with problem size and also cycle accurate simulations is very slow, only few kilo instructions per second [35].

In this chapter, we focus on the analysis of vulnerability of a program to soft errors in cache. To compute the cache vulnerability, it requires tracking for each bit the ratio of

program execution time during which a bit flip will escape from cache become visible in other architectures such as registers or memory. For example, written back a dirty cache, which contains corrupted data, will lead to an soft error become visible. The computation of vulnerability requires exact information of the operations performed on each data of the cache through the program execution including reads, writes, misses, evictions etc.

Typically, simulation is the way the perform this analysis as the lifetime of cache lines can be precisely monitored in the simulator. However, the simulation time could up to hours and days even for a simple program and cache configuration [112].

Therefore, we propose a novel purely static analysis to compute vulnerability metrics at compile-time based on the analytical cache modeling in previous chapter. This opens the way for software optimization of vulnerability via choosing program transformation automatically.

We achieve this goal by focusing on a specific class of computations, namely affine programs [48, 63], which have a static control- and data-flow that can be exactly represented at compile-time using mathematical structures such as polyhedra of integer tuples and affine relations. Affine programs occur frequently in several scientific computing patterns, such as dense linear algebra [63, 34].

5.2 Modeling Cache Vulnerability

Modeling cache vulnerability requires the ability to reason the lifetime of each data stored in cache lines. However, typical cache modeling approach, which only produce number of cache misses, cannot be used to compute cache vulnerability. Compared to these models, our analytical cache model from previous chapter could generate ordered set of cache events, which could be used to analyze the lifetime of cache lines. Therefore,

this section introduce the way to extend the cache behavior model to perform the cache vulnerability analysis.

Note that the program representation for affine program used in this chapter are the same with previous chapter.

5.2.1 Cache Vulnerability

The concept of architectural vulnerability factor (AVF) proposed by Mukherjee et al. in [89] is the probability that faults in processor architectures results in visible output errors. The AVF provides the insight of reliability of various hardware components by estimating the architecture soft error rate. Similarly, the concept of cache vulnerability factor (CVF), which is the probability that soft errors happened in cache memory got propagated to other hardware components, is introduced [135] to estimate the reliability of cache memory. It can be calculated as the ratio of the time during which a cache line is vulnerable, i.e. susceptible to soft errors, divided by the lifetime of the cache line, for all cache lines as shown as the following equation.

$$CacheVunerabilityFactor = \frac{\sum_{i=1}^{N} VulnerableTime_i}{\sum_{i=1}^{N} LifeTime_i}$$

Where $VulnerableTime_i$ is the time interval that cache line i is susceptible to soft errors, and $Lifetime_i$ represents the lifetime of cache line i from loading the data to the end, and N is the number of lines in cache.

Architecturally correct execution (ACE) is introduced in [89] to calculate the AVF by performing the lifetime analysis, which divides up a bit's lifetime during execution into ACE and un-ACE intervals. Within unACE interval, any bit value change will not affect the program final outcome, in other words, not vulnerable to soft errors. And if the time

interval cannot be proven to be unACE, it will be assumed to be ACE, which is vulnerable interval that susceptible to soft errors.

Therefore, to characterize the CVF, we have to count all the ACE and unACE intervals to compute the ratio of vulnerable ones, which make the equation above into the following.

$$CVF = \frac{\sum_{i=1}^{N} ACE_i}{\sum_{i=1}^{N} (ACE_i + unACE_i)}$$

Where ACE_i is the ACE interval while $unACE_i$ is the unACE interval for cache line i.

The lifetime analysis in [30] divides lifetime into several non-overlapping intervals such as idle, read-to-write, write-to-write, read-to-evict etc and classify each of these components into ACE or unACE interval. Table 5.1 shows the detailed classification for write-back data cache, where in bold we display each of the intervals we model analytically in our framework work. Write-to-Write is the "time" interval from a cache line write event to the next write event to the same cache line; Read-to-Evict is the interval from a read event to the following eviction event of that cache line; etc. (details can be found in [30]). Note that our model is also applicable to write-through cache, but we only focus on write-back caches since the vulnerability of write-through caches is really marginal and not the main focus.

Table 5.1: Lifetime Classification of Write-Back Cache

unACE Component	ACE Component
Idle,**Write-to-Write**	Write-to-End,**Write-to-Evict**
Fill-to-Evict,**Read-to-Evict**	Evict-to-Fill,**Read-to-Read**
Read-to-Write,Fill-to-Write	**Write-to-Read**

In order to compute the CVF, it is needed to determine the lifetime of the data in each cache line. For example, the data in a cache line is vulnerable in between the time it was loaded for the first time (i.e., the first miss leading to loading the line) and the last time the cache line is accessed prior to its eviction. The typical approach to perform this analysis is to rely on precise cache simulation, where events such as miss, accesses, eviction, etc. are recorded. More precisely, system simulators like GEM5 are the typical method of choice, because in addition to being able to record events such as misses and evictions, they provide the number of cycles elapsed in between such events, i.e., the lifetime information. To avoid the need for simulation, an exact compile-time modeling of all cache events is needed, to determine all ACE and unACE intervals, as developed in the following.

5.2.2 Modeling Vulnerability Intervals

We now show how we compute the various event intervals for ACE and unACE components, before presenting the overall algorithm in Sec. 5.2.3. We build on the formalism presented in prior sections, and in particular on Alg. 5 which computes ***the relation*** **Miss** ***from the iteration making the first access to a cache line to the iteration when this line is first evicted***. We first estimate these intervals in terms of iteration space, that is by representing an interval via its start iteration (a point in the iteration domain) and its end iteration.

First, we assume the cache is invalid at the start of the program, and then the Idle component is the interval between the start of the program region analyzed and the time at which the cache line gets filled. To compute the Idle time, Alg. 9 computes the first K distinct lines that maps to a cache set, where K is the associativity of the cache.

Algorithm 9 unACE-idle: Compute unACE idle component

Output: unACE Idle component

1: $\mathbf{A} = \mathbf{IterToSet}^{-1}$
2: $\mathbf{SameSet} = \mathbf{IterToSet} \circ \mathbf{IterToSet}^{-1}$
3: $\mathbf{Idle} = \emptyset$
// Compute first K distinct lines that maps to an index
4: **for all** assoc in 1, K **do**
5: $\mathbf{B} = \mathbf{A} - (\mathbf{A} \circ \mathbf{SameSet})$
6: $\mathbf{A} = \mathbf{A} - \mathbf{B}$
7: $\mathbf{Idle} = \mathbf{Idle} \cup \mathbf{B}$
8: **end for**
// Return Idle in terms of cache lines
9: **return Idle**

We then show in Alg. 10 how to compute the other types of interval. The output of this algorithm is 6 relations, that each represents sets of intervals, defined by the relation from the start iteration to the end iteration, thereby defining the interval. By virtue of our formalism, Alg. 10 reasons on all cache lines at once, that is the intervals produced are in fact sets of intervals, for the various cache lines. The algorithm takes as input the program description and other necessary modeling parameters, and the **Miss** relation computed in Alg. 5, the relation from the iteration allocating the cache line for a memory reference to the iteration where this same line get evicted.

Line 1 to 4 of the algorithm collect all the lifetime events. Line 5 compute *OO*, the relation from an operation to the nearest following one since all components are non-overlapped. Then Line 6 to 11 calculate all possible combinations for each particular lifetime component. For example, $domain(RA)$ contains all read references, its Cartesian product (noted $\times$) with itself produces all possible combinations of relations from one read operation to another and we only need those immediate next one, which can be obtained by intersecting with relation *OO*.

Algorithm 10 Lifetime components

unACE: read-to-write, read-to-evict, write-to-write

ACE: read-to-read, write-to-read, write-to-evict

Input: Program: **WriteRefs**, **ReadRefs**, **Sched**

Array: $(\vec{sz}_{Array}, start_{Array})\ \forall\ Array$, **Linearize**

Cache: **Miss**(cache miss formulation from Alg.1)

Output: ACE: **RtoR**, **WtoR**, **WtoE**

unACE: **RtoW**, **RtoE**, **WtoW**

// Memory address written at a given time

1: **WA** = **Sched**$^{-1}$ $\circ$ **WriteRefs** $\circ$ **Linearize**

// Memory address read at a given time

2: **RA** = **Sched**$^{-1}$ $\circ$ **ReadRefs** $\circ$ **Linearize**

// Memory address evicted at a given time

3: **EA** = **Miss**$^{-1}$ $\circ$ **Sched**$^{-1}$ $\circ$ (**ReadRefs** $\cup$ **WriteRefs**) $\circ$ **Linearize**

// All Memory Operations at a given time

4: **OA** = **WA** $\cup$ **RA** $\cup$ **EA**

// From each operation to its immediate next operation

5: **OO** = **lexmin** ((**OA** $\circ$ **OA**$^{-1}$) $\cap$ **lexPrec**)

// Calculate each lifetime component

// ACE:

6: **RtoR** = **OO** $\cap$ (domain (**RA**) $\times$ domain (**RA**))

7: **WtoR** = **OO** $\cap$ (domain (**WA**) $\times$ domain (**RA**))

8: **WtoE** = **OO** $\cap$ (domain (**WA**) $\times$ domain (**EA**))

// unACE:

9: **RtoW** = **OO** $\cap$ (domain (**RA**) $\times$ domain (**WA**))

10: **RtoE** = **OO** $\cap$ (domain (**RA**) $\times$ domain (**EA**))

11: **WtoW** = **OO** $\cap$ (domain (**WA**) $\times$ domain (**WA**))

// Return all lifetime components

12: **return** **RtoR**,**WtoR**,**WtoE**,**RtoW**,**RtoE**,**WtoW**

We compute all necessary time intervals from Alg. 10. Technically, some intervals cannot be modeled through static analysis, but in practice these components account for a very small portion of the execution, for example Evict-to-Fill is not modeled but a cache line fill after eviction takes typically only a few cycles for L1 cache.

5.2.3 From Vulnerability Intervals to Vulnerability Metrics

The final stage is to compute a vulnerability metric, from the various intervals we generated. Algorithm 11 puts together all the relations computed by the previous algorithm to form the set of intervals in the ACE and unACE components. We then enumerate these intervals to compute an approximation of the time elapsed in this interval. For simplicity, in Alg. 11 we assume each operation (be it a cache miss or a floating point computation) takes one cycle. In practice, this is refined within the implementation of computeIntervalTime() where one can assign weights to different operations as desired.

Algorithm 11 CVF: Compute vulnerability factor

Input: **ACE** = **RtoR** $\cup$ **WtoR** $\cup$ **WtoE**
unACE = **Idle** $\cup$ **RtoW** $\cup$ **RtoE** $\cup$ **WtoW**
Output: CVF: cache vulnerability factor

1: $unace = ace = 0$
2: **for each** $(\vec{x} \rightarrow \vec{y})$ in **unACE do**
3: $unace$ += computeIntervalTime$(\vec{x}, \vec{y})$
4: **end for**
5: **for each** $(\vec{x} \rightarrow \vec{y})$ in **ACE do**
6: ace += computeIntervalTime$(\vec{x}, \vec{y})$
7: **end for**
// Return the CVF
8: **return** $ace/(unace + ace)$

We remark that as the entire formulation stays within the boudary of integer sets as defined in Sec. 4.3, enumerating all points (pairs) in a relation is achieved simply by performing polyhedral code generation to create a C code scanning all and only points in the relation. In practice, we call the codegen method of the ISL library, which produces a loop nest visiting each pair $(\vec{x} \rightarrow \vec{y})$ in the relation. The estimation of the time elapsed between $\vec{x}$ and $\vec{y}$, two iterations in the program, is obtained by restricting the iteration domain of the program such that we only keep iterations $\vec{i}$ such that $\vec{x} \preceq \vec{i} \preceq \vec{y}$, and counting the number of points in this restricted domain.

The above model of time is approximate, that is assuming one cycle per operation is not an exact model for the lifetime, but integrating refinements such as specifying a latency (number of cycles) for a cache miss is straightforward within computeIntervalTime.

5.3 Experimental Evaluation

5.3.1 Implementation detail

The tool to perform the cache vulnerability analysis is named as PolyVul. It was implemented using ISL-0.17 [5] (with barvinok-0.38 and pet-0.07), an integer set library for the polyhedral model. The Polyhedral Extraction Tool (PET) [126], a powerful SCoP extractor on top of Clang, detects affine regions and extracts the polyhedral model from C source code. ISL [124] performs the algorithms described in previous section on the extracted model to compute and output the cache vulnerability based on cache behavior metrics for the input program and cache configuration.

PolyVul takes three different kinds of information as input: 1) Input C affine program, 2) Cache configurations including cache size, block size, associativity and 3) Memory base address to calculate the cache vulnerability.

We use the set-specialization approach discussed in Sec. 4.4.3, that is we compute the analysis for each set independently, and then sum the results as appropriate. These computations are run in parallel, using one process per set in the cache, as each set's behavior is independent of the others.

5.3.2 Time complexity Evaluation

Since the cache vulnerability analysis is based on the cache modeling from previous chapter, it again extensively rely on manipulating sets by Presburger formulae, which meaning the computation in worst cast is NP-hard.

As we observe that the execution time of cache modeling for cache misses is fast in practice for most of the cases. However, vulnerability modeling is much more complex than cache miss modeling as we introduced previously. Thus, in this section, we perform scaling experiments to validate the time complexity on three representative cases, Gemm, Symm and Trmm.

Table 5.2, 5.3 and 5.4 details time cost for each operator used and the total time to for different problem size and cache configurations.

First column lists the benchmarks and second column lists the operators used in the approach such as composition. CVF denotes the cache vulnerablity factor obtained from the analysis. PolyCache meaning the time takes to compute the cache misses for this cache configuration, and similar for PolyVul. Column 3 to 6 show the execution time in second for 1-way set associative cache with cache size varied from 2K to 16K, here C denotes cache size. Column 7 to 9 show the execution time for cache associativity ranged from 1-way to 4-way but with cache size fixed at 2K. In table 5.4, N/T meaning not terminate after 24 hours execution. Hence, no time breakdown for the operators.

Table 5.2: Execution time experiments for problem size N=8

Benchmark	Operation	Problem Size N=8						
		Associativity 1-way				Cache Size 2K		
		C=2K	C=4K	C=8K	C=16K	1-way	2-way	4-way
Dgemm	composition	0.04	0.05	0.05	0.05	0.05	0.05	0.05
	difference	0.53	0.54	0.54	0.54	0.53	0.54	0.37
	coalsecing	0.62	0.63	0.64	0.63	0.62	0.63	0.63
	cardinality	0.01	0.01	0.01	0.01	0.01	0.01	0.01
	lexmin	0.01	0.01	0.01	0.01	0.01	0.01	0.01
	polycache	0.07	0.07	0.07	0.07	0.07	0.07	0.11
	polyvul	1.33	1.31	1.32	1.31	1.29	1.31	1.15
	CVF	0.80	0.80	0.80	0.80	0.80	0.80	0.80
Symm	composition	4.64	4.53	4.63	4.60	4.54	4.55	5.42
	difference	22.10	21.46	20.86	21.29	20.86	20.76	36.09
	coalsecing	201.51	195.99	197.47	195.78	194.15	197.69	203.76
	cardinality	0.02	0.02	0.02	0.02	0.02	0.02	0.02
	lexmin	0.17	0.17	0.17	0.17	0.17	0.17	0.17
	polycache	0.90	0.90	0.90	0.90	0.90	0.90	2.71
	polyvul	234.29	227.26	228.91	227.57	225.46	228.90	251.66
	CVF	0.93	0.93	0.93	0.93	0.93	0.93	0.93
Trmm	composition	0.03	0.03	0.03	0.03	0.03	0.03	0.03
	difference	0.04	0.04	0.04	0.04	0.04	0.04	0.04
	coalsecing	0.65	0.65	0.65	0.66	0.66	0.66	0.66
	cardinality	0.01	0.01	0.01	0.01	0.01	0.01	0.01
	lexmin	0.01	0.01	0.01	0.01	0.01	0.01	0.01
	polycache	0.69	0.69	0.69	0.69	0.69	0.69	0.69
	polyvul	0.82	0.80	0.80	0.80	0.80	0.80	0.80
	CVF	1.00	1.00	1.00	1.00	1.00	1.00	1.00

We make several observations from compare the numbers within these tables. First, the relative size between cache and program will affect the execution time. For problem size N=8 and N=16, cache size 2K and 3K can hold all the date within the program. The main cache misses will be cold misses, thus PolyCache time is marginal. Therefore, there will be less cache events such as evictions, and then ACE/unACE interval calculation will be much easier. However, when problem size N=32, only cache size 16K can hold all the data.

Table 5.3: Execution time experiments for problem size N=16

Benchmark	Operation	Problem Size N=16						
		Associativity 1-way				Cache Size 2K		
		C=2K	C=4K	C=8K	C=16K	1-way	2-way	4-way
Dgemm	composition	0.03	0.03	0.03	0.03	0.03	0.05	0.16
	difference	0.01	0.00	0.00	0.00	0.01	0.04	1.00
	coalsecing	0.22	0.17	0.17	0.17	0.22	0.50	0.40
	cardinality	0.01	0.01	0.01	0.01	0.01	0.03	0.01
	lexmin	0.01	0.01	0.01	0.01	0.01	0.02	0.01
	polycache	0.07	0.06	0.06	0.06	0.07	0.12	1.43
	polyvul	0.38	0.25	0.25	0.25	0.32	0.73	1.67
	CVF	0.59	0.80	0.80	0.80	0.85	0.58	0.68
Symm	composition	2.51	2.20	2.20	2.19	2.48	3.21	13.13
	difference	0.82	0.50	0.49	0.49	0.81	1.82	999.10
	coalsecing	19.42	14.48	14.27	13.88	19.04	19.74	48.42
	cardinality	0.06	0.01	0.01	0.01	0.06	0.07	0.09
	lexmin	0.33	0.16	0.16	0.16	0.33	0.42	0.66
	polycache	0.87	0.54	0.55	0.54	0.87	2.22	84.56
	polyvul	31.48	23.90	23.69	23.27	31.04	33.21	1075.54
	CVF	0.52	0.97	0.97	0.97	0.64	0.64	0.71
Trmm	composition	0.02	0.02	0.02	0.02	0.02	0.02	0.08
	difference	0.00	0.00	0.00	0.00	0.00	0.01	0.56
	coalsecing	0.19	0.20	0.19	0.20	0.20	0.23	0.27
	cardinality	0.01	0.01	0.01	0.01	0.01	0.01	0.01
	lexmin	0.01	0.01	0.01	0.01	0.01	0.01	0.01
	polycache	0.16	0.16	0.16	0.16	0.16	0.21	0.87
	polyvul	0.27	0.26	0.26	0.26	0.26	0.31	0.97
	CVF	1.00	1.00	1.00	1.00	1.00	1.00	1.00

So we can observe the long execution time for cache size less than 16K and also along with many N/T cases.

Second, higher set associativity will result in longer execution time. This can be observed more obviously from benchmark **Symm**. We can see that the time takes for operator **difference** is much longer for higher associativity. This is because the difference operation

Table 5.4: Execution time experiments for problem size N=32

Benchmark	Operation	Problem Size N=32						
		Associativity 1-way				Cache Size 2K		
		C=2K	C=4K	C=8K	C=16K	1-way	2-way	4-way
Dgemm	composition	N/T	0.11	0.55	0.03	N/T	N/T	N/T
	difference	N/T	0.31	0.06	0.09	N/T	N/T	N/T
	coalsecing	N/T	5.10	0.90	0.36	N/T	N/T	N/T
	cardinality	N/T	6.47	0.05	0.01	N/T	N/T	N/T
	lexmin	N/T	0.82	0.10	0.01	N/T	N/T	N/T
	polycache	1.63	0.37	0.15	0.08	1.63	13.34	125.96
	polyvul	N/T	13.13	1.35	0.57	N/T	N/T	N/T
	CVF	N/T	0.69	0.85	0.80	N/T	N/T	N/T
Symm	composition	19.95	N/T	4.20	3.33	19.95	N/T	N/T
	difference	251.09	N/T	7.84	5.40	251.09	N/T	N/T
	coalsecing	2657.81	N/T	139.48	109.73	2657.81	N/T	N/T
	cardinality	2295.82	N/T	0.38	0.01	2295.82	N/T	N/T
	lexmin	539.34	N/T	7.33	4.98	539.34	N/T	N/T
	polycache	200.85	72.37	68.54	102.34	200.85	756.94	17498.48
	polyvul	6008.27	N/T	173.15	129.31	6008.27	N/T	N/T
	CVF	0.57	N/T	0.65	0.98	0.57	N/T	N/T
Trmm	composition	0.58	0.13	0.03	0.03	0.58	N/T	N/T
	difference	3.87	0.12	0.03	0.03	3.87	N/T	N/T
	coalsecing	564.25	133.96	0.69	0.64	564.25	N/T	N/T
	cardinality	1512.54	0.58	0.01	0.01	1512.54	N/T	N/T
	lexmin	372.38	78.32	0.02	0.02	372.38	N/T	N/T
	polycache	2.14	0.81	0.71	0.67	2.14	25.06	162.72
	polyvul	2459.05	214.18	0.81	0.77	2459.05	N/T	N/T
	CVF	0.67	0.55	1.00	1.00	0.67	N/T	N/T

for set will performed more times and each time it could potentially generate more complex and irregular sets, which result in longer time cost of these sets even for other operators.

Moreover, we can also observe the difference between PolyCache and PolyVul. The time difference could be up to 1000x times as we see the execution time for **Trmm**. The difference comes from the operator **cardinality** and **lexmin**. This is because we need to

manipulate and counting of the vulnerable intervals to compute the vulnerability as we introduced in the Algorithm 10.

5.4 Conclusion

Computing cache vulnerability of programs involves tracking the fraction of program execution time during which a cache line is being accessed. Typical approaches to compute vulnerability rely on time-consuming simulation, making the approaches unsuitable for compile-time optimization of vulnerability.

In this chapter, we proposed the first fully analytical simulation-free framework to compute the program cache vulnerability factor (CVF) for arbitrary polyhedral programs. It requires modeling ordered set of cache events for affine programs in previous section. However, it still suffers the time complexity problem as we showed in the experiments through scaling experiments. Thus, as future work, we will investigate approaches to reduce the time cost for the vulnerability modeling. Also, we will investigate efficient exploration of the space of possible program transformations to optimize cache vulnerability.

CHAPTER 6

Related Work

This section is a brief discussion of the related work: Section 6.1 introduces the related work in context of Chapter 2, Section 6.2 provides the related work in context of Chapters 3, and Section 6.3 discusses the related work in context of Chapters 4 and 5.

6.1 Verification of Program Transformation

Verdoolaege et al. proposed a fully automatic technique to prove equivalence between two affine programs [127]. Leveraging polyhedral data-flow analysis, they developed widening/narrowing operators to properly handle non-uniform recurrences. It is implemented in the ISA tool [4]. However, in contrast to our approach, it is limited to verifying affine program transformations. Basupalli et al. developed ompVerify, to find OpenMP parallelization errors in affine programs [26]; and Alias et al. [13, 23] have developed other techniques to recognize algorithm templates in programs. These approaches are restricted to static/affine transformed programs. Karfa et al. also designed a method that works for a subset of affine programs using array data dependence graphs (ADDGs) to represent input and transforming behaviors. Operator-level equivalence checking provides the capability to normalize expressions and establish matching relations under algebraic transformations [74].

Recently, Schordan et al. proposed a trace-based framework to verify if two programs (one possibly being a transformed variant of the other) are semantically equivalent. Their method combines the computation of a state transition graph with a rewrite system to transform floating point computations and array update operations of one program to match them as terms with those of the other program [109]. In contrast to our approach, which only requires the same space as the input program's working dataset size, the space complexity in their approach is a function of the total number of dynamic instances of operations.

Other related works include Mansky and Gunter, who used the TRANS language [73] to represent transformations. The correctness proof implemented in the verification framework [85] is verified by the Isabelle [94] proof assistant. Regression verification has been considered to support recursion, but it actually requires loops to be converted to recursion first [64]. Other works also include translation validation [79, 90].

6.2 Energy Optimization through DVFS

DVFS technology, which originated in real-time and embedded system community [46], is widely applied nowadays since energy/power have become the primary optimization metric on the entire spectrum of computing, from handheld devices [46] to desktops [84] and to clusters [134].

CPU MISER by Rong Ge et al. [59] is a run-time approach to adapt the frequency/-voltage based on the measured cycle per instructions achieved by phases of the running workload. This approach works on arbitrary binaries, however it is geared towards finding the smallest frequency which does not degrade performance.

Hsu and Kremer [69] developed a compiler-assisted technique to change the operating voltage for specific regions of the program, attempting to control execution time penalty,

but they require an actual profiling of each application to be run, while we rely solely on computing compile-time characteristics for new applications. Saputra et al. studied the impact of loop transformations on static and dynamic voltage strategies to reduce power [105]. Jimborean et al. proposed a decoupled access/execute (DAE) model using an access phase generation as a speculative prefetch to make the execute tasks effectively compute bound, so that the DVFS could adjust frequency accordingly [72]. Our compile-time approach to DVFS could be employed on top of both approaches, to further improve energy savings.

Yuki and Rajopadhye [134] studied with attention the race-to-sleep scenario, especially for current supercomputers. Using simplified power equations for the CPU, and looking at the ratio between CPU power vs. the power of the rest of the system, they analytically determined that massive DVFS may not be overall profitable in a race-to-sleep scenario where the entire system goes to sleep when the computation ends. While our results do not contradict theirs, when evaluating using actual programs we observe the execution time penalty may be minimal, leading to overall energy savings as demonstrated in other works [59]. In addition, the race-to-sleep scenario is not always applicable: very often, the operating system (and the entire node) keeps running when a code/kernel completed, so there is still large potential benefits in exploiting CPU DVFS for program segments.

Power modeling methodologies have been also studied [41, 15], to study analytically the evolution of the processor power as a function of voltage, frequency, and temperature. [114] in particular studied the role of temperature on leakage, leading to a more realistic power equation. Recently, De Vogeleer et al. [40] used measurements in a control environment on a mobile CPU to confirm a realistic power/energy equation for CPU power. They showed the existence of an energy/frequency convexity rule, that is the existence of

a unique optimum frequency for energy efficiency for a fixed workload. Interestingly, our data in Sec. 5.1 illustrates exactly their point, showing different optimal frequencies across four Intel x86/64 desktop processors. Note however this principle does not necessarily hold in the case of co-execution of different workloads on the same CPU core(s).

Power and performance adaption based on DVFS is also applied for thread-level parallelism. Li and Martinez [80] proposed a low-overhead heuristics DVFS algorithm to obtain optimal power savings. Similar studies of DVFS to improve energy efficiency also have been applied for GPUs [87, 60, 133], Mei et al. [87] performed a measurement study to explore the efficiency of GPU DVFS on system energy consumption. Ge et al. [60] proposed the GPU DVFS study of performance and energy efficiency on Nvidia K20c Kepler GPU and compared with CPU DVFS. Our proposed compile-time characterization of the operational intensity could help improving these works to make a better DVFS decision.

6.3 Cache Behavior Modeling and Vulnerability Analysis

Cache simulation Simulators such as Dinero [43] and Sniper [36] characterize an application's cache behavior with varying degrees of fidelity. Also, approaches have been developed to simulate multiple levels of associativity simultaneously [67]. In general, simulation cost is proportional to the number of executed operations, with the approaches either executing the program or manipulating large memory reference traces [129, 21]. Moreover, besides cache behavior analysis, previous vulnerability analysis works are mainly depend on simulations to characterize the vulnerability. However, these approaches do not provide a closed-form model that can be used by compile-time optimizers.

Approximate analysis of cache behavior Approximate metrics, such as reuse distance, have been used as an approximate measure of cache reuse by compile-time optimization techniques [83, 34, 12, 37, 75]. Ferrante et al. [49] estimate the number of distinct cache lines accessed by a loop nest. Similar analyses have been developed to predict cache miss ratios for set-associative and fully associative caches [11, 113, 66]. Other approaches to estimate cache misses include probabilistic estimation of an array reference incurring a miss [52, 53] and sampling the iteration to approximate the absolute miss ratio for each static array reference [122], dynamic memory reference [28], and individual instructions [45, 44]. These techniques provide inexact cache behavior analysis while being potentially applicable to a larger class of programs.

Frumkin and Van der Wijngaart [57] developed bounds on cache misses for stencil operations on rectangular grids. Our approach can generate exact cache miss information for a larger class of programs that includes such stencil codes.

Exact analysis of cache misses Cascaval et al. [38] estimate cache misses using stack distances to construct a stack histogram. The model is accurate for fully associative caches with LRU replacement policy and provides approximate solutions for set-associative caches. This work is restricted to perfectly nested loops and dependence characterized as distance vectors. Beyls and D'Hollander [29] develop a compile-time analysis of cache misses for fully associative caches by analytical modeling of reuse distances, defined as the number of other distinct cache lines accessed between two successive references to a particular cache line. By identifying cache misses and hits in certain parts of the program, Alt et al. [14] and [22] model cache behavior using abstract interpretation to improve worst-case execution time bounds.

Ghosh et al. [62, 61] present cache miss equations (CMEs), an approach to counting the exact number of cache misses in perfectly nested loop nests in the form of the number of solutions to a set of affine equations. The number of solutions to the cache miss equations gives an exact count of the set of misses in direct-mapped and set-associative caches. The CMEs have been extended for use in a variety of contexts, including bounding worst-case data cache behavior and execution time for real-time systems [102]Solving cache miss equations is expensive [123, 121], motivating the design of approximate counting strategies [62, 120]. Vera et al.[123, 132] extend the use of CMEs to count cache misses in programs consisting of imperfectly nested loops and non-recursive subroutine calls using abstract inlining and loop normalization, sinking with if-conditionals. CMEs and approaches that build on them are restricted to handling dependences characterized as distance vectors.

Chatterjee et al. present an exact solution for direct-mapped caches using Presburger formulae [39]. Their formulation for set-associative caches is limited to interior misses, but it appears extensible to boundary misses. But unlike our approach, such formulation cannot be used to build the ordered set of events needed for hierarchical caches.

Cache vulnerability analysis Typical approaches to perform vulnerability analysis involves characterizing state as required to correct for architecture correct execution (ACE vs un-ACE) [89, 130]. Specifically, particular state for a bit is said to un-ACE if an incorrect value for that bit is masked or overwritten and does not get propagated to other architectures such as registers or memory and become visible.

These approaches analyze cache vulnerability through simulations [115, 77, 103, 76]. Li et al. employ a probabilistic error propagation model to determine the mean time to

failure for a program executing on a given architecture [81]. While precisely capturing micro-architectural features, these simulations can be expensive, especially when carried out to evaluate a space of possible choices such cache organization, compiler optimizations and effectiveness of cache resilience techniques. This has motivated several approaches to accelerate the AVF calculations. Sridharan et al. [116] accelerate AVF calculations through the use of program vulnerability factors. Li et al. presented an approach to estimate AVF online using hardware support [82].

Approaches to approximate the computation of AVF includes the determination performance metrics that correlate well with vulnerability factors [42]. These approaches execute the program on the target hardware to observe performance events such as cache misses to estimate vulnerability.

Shrivastava et al. [112] presented a static analysis approach to characterize cache vulnerability based on previously introduced CMEs. However, the model is limited to: perfectly nested loops, direct-mapped caches, and single-level caches. Although CMEs are exact in counting cache misses, they are less accurate in computing vulnerability compare to our approach because reuse vectors are not inexact in representing data access and thus the cache behavior.

CHAPTER 7

Conclusion

7.1 Conclusion

Program transformations are classic optimizations to improve performance and better use current hardware components such as cache. However, the implementation of complex transformation are often buggy and results in errors in transformed programs that are difficult to find. Current approaches suffers various problems and there are no efficient approach to tackle this program. Similarly, current compiler optimizations to improve performance are performed based on very approximate cost model rather than accurate model of cache. Simulation based approaches suffers complexity problem and current cache modeling have various limitations for practical use. Moreover, the growth of power cost problems require the optimizations not only for performance but also consider reducing energy consumptions.

This dissertation proposed new compiler approaches to solve above problems. It presents a dynamic verification approach to check the program transformation automatically and in light weight way, a static cache modeling to provide accurate cost model for cache memory and its vulnerability characterization, and a compile-time analysis to determine best frequency and core pair choice for program to reduce energy cost.

BIBLIOGRAPHY

[1] Barvinok, a library for counting the number of integer points in parametric and non-parametric polytopes. http://barvinok.gforgeinria.fr.

[2] Clan, the Chunky Loop Analyzer. http://icps.u-strasbg.fr/ bastoul.

[3] GNU GCC. http://gcc.gnu.org.

[4] ISA 0.13. http://repo.or.cz/w/isa.git.

[5] ISL, the Integer Set Library. http://repo.or.cz/w/isl.git.

[6] LLVM. http://llvm.org.

[7] MIT Cilk. http://supertech.csail.mit.edu/cilk.

[8] PoCC, the Polyhedral Compiler Collection 1.3. http://pocc.sourceforge.net.

[9] PolyBench/C 3.2. http://polybench.sourceforge.net.

[10] ADAMS, M. Hpgmg: a benchmark for ranking high performance computing systems.

[11] AGARWAL, A., HENNESSY, J., AND HOROWITZ, M. An analytical cache model. *ACM Trans. Comput. Syst. 7*, 2 (May 1989), 184–215.

[12] AHMED, N., MATEEV, N., AND PINGALI, K. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. *International Journal of Parallel Programming 29*, 5 (2001), 493–544.

[13] ALIAS, C., AND BARTHOU, D. On the recognition of algorithm templates. *Electronic Notes in Theoretical Computer Science 82*, 2 (2004), 395–409.

[14] ALT, M., FERDINAND, C., MARTIN, F., AND WILHELM, R. Cache behavior prediction by abstract interpretation. In *International Static Analysis Symposium* (1996), Springer, pp. 52–66.

[15] AUSTIN, B., AND WRIGHT, N. J. Measurement and interpretation of micro-benchmark and application energy use on the Cray XC30. In *E2SC* (2014), pp. 51–59.

[16] BAO, W. Power aware wcet analysis, 2014.

[17] BAO, W., HONG, C., CHUNDURI, S., KRISHNAMOORTHY, S., POUCHET, N., RASTELLO, F., AND SADAYAPPAN, P. Static and dynamic frequency scaling on multicore cpus. *ACM Transactions on Architecture and Code Optimization* (2016), 1–26.

[18] BAO, W., KRISHNAMOORTHY, S., POUCHET, L., RASTELLO, F., AND SADAYAPPAN, P. Polycheck: Dynamic verification of iteration space transformations on affine programs. *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'16)* (2016), 539–554.

[19] BAO, W., KRISHNAMOORTHY, S., POUCHET, L.-N., RASTELLO, F., AND SADAYAPPAN, P. Polycheck: Dynamic verification of iteration space transformations on affine programs. Tech. rep., OSU/PNNL/INRIA, Nov. 2015. OSU-CISRC-11/15-TR21.

[20] BAO, W., KRISHNAMOORTHY, S., POUCHET, L.-N., AND SADAYAPPAN, P. Analytical modeling of cache behavior for affine programs. *Proceedings of the ACM on Programming Languages 2*, POPL (2017), 32.

[21] BAO, W., RAWAT, P., KONG, M., KRISHNAMOORTHY, S., POUCHET, L., AND SADAYAPPAN, P. Efficient cache simulation for affine computations. In *International Workshop on Languages and Compilers for Parallel Computing (LCPC'17)* (2017).

[22] BAO, W., TAVARAGERI, S., OZGUNER, F., AND SADAYAPPAN, P. Pwcet: Power-aware worst case execution time analysis. In *43rd International Conference on Parallel Processing Workshops* (2014), pp. 439–447.

[23] BARTHOU, D., FEAUTRIER, P., AND REDON, X. On the equivalence of two systems of affine recurrence equations. In *Euro-Par 2002 Parallel Processing*. 2002.

[24] BASKARAN, M. M., HARTONO, A., TAVARAGERI, S., HENRETTY, T., RAMANUJAM, J., AND SADAYAPPAN, P. Parameterized tiling revisited. In *Proc. of the 8th annual IEEE/ACM international symposium on Code generation and optimization* (2010), ACM.

[25] BASTOUL, C. Code generation in the polyhedral model is easier than you think. In *Proc. of the 13th International Conference on Parallel Architectures and Compilation Techniques* (2004), IEEE.

[26] BASUPALLI, V., YUKI, T., RAJOPADHYE, S., MORVAN, A., DERRIEN, S., QUINTON, P., AND WONNACOTT, D. ompVerify: polyhedral analysis for the OpenMP programmer. In *OpenMP in the Petascale Era*. Springer, 2011, pp. 37–53.

[27] BENDER, M. A., FINEMAN, J. T., GILBERT, S., AND LEISERSON, C. E. On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs. In *Proc. of the 16th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'04)* (2004), ACM.

[28] BERG, E., AND HAGERSTEN, E. Statcache: a probabilistic approach to efficient and accurate data locality analysis. In *Performance Analysis of Systems and Software, 2004 IEEE International Symposium on-ISPASS* (2004), IEEE, pp. 20–27.

[29] BEYLS, K., AND D'HOLLANDER, E. H. Generating cache hints for improved program efficiency. *Journal of Systems Architecture 51*, 4 (2005), 223 – 250.

[30] BISWAS, A., RACUNAS, P., CHEVERESAN, R., EMER, J., MUKHERJEE, S. S., AND RANGAN, R. Computing architectural vulnerability factors for address-based structures. In *32nd International Symposium on Computer Architecture (ISCA'05)* (2005), IEEE, pp. 532–543.

[31] BLUMOFE, R. D., JOERG, C. F., KUSZMAUL, B. C., LEISERSON, C. E., RANDALL, K. H., AND ZHOU, Y. Cilk: an efficient multithreaded runtime system. In *Proc. of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (1995), ACM.

[32] BLUMOFE, R. D., JOERG, C. F., KUSZMAUL, B. C., LEISERSON, C. E., RANDALL, K. H., AND ZHOU, Y. *Cilk: An efficient multithreaded runtime system*, vol. 30. ACM, 1995.

[33] BLUMOFE, R. D., AND LEISERSON, C. E. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM) 46*, 5 (1999), 720–748.

[34] BONDHUGULA, U., HARTONO, A., RAMANUJAM, J., AND SADAYAPPAN, P. A practical automatic polyhedral program optimization system. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (2008), ACM.

[35] BURGER, D., AND AUSTIN, T. M. The simplescalar tool set, version 2.0. *ACM SIGARCH computer architecture news 25*, 3 (1997), 13–25.

[36] CARLSON, T. E., HEIRMAN, W., EYERMAN, S., HUR, I., AND EECKHOUT, L. An evaluation of high-level mechanistic core models. *ACM Transactions on Architecture and Code Optimization (TACO)* (2014).

[37] CARR, S., MCKINLEY, K. S., AND TSENG, C.-W. Compiler optimizations for improving data locality. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 1994), ASPLOS VI, ACM, pp. 252–262.

[38] CASCAVAL, C., AND PADUA, D. A. Estimating cache misses and locality using stack distances. In *Proceedings of the 17th annual international conference on Supercomputing* (2003), ACM, pp. 150–159.

[39] CHATTERJEE, S., PARKER, E., HANLON, P. J., AND LEBECK, A. R. Exact analysis of the cache behavior of nested loops. *ACM SIGPLAN Notices 36*, 5 (2001), 286–297.

[40] DE VOGELEER, K., MEMMI, G., JOUVELOT, P., AND COELHO, F. The energy/frequency convexity rule: Modeling and experimental validation on mobile devices. In *Parallel Processing and Applied Mathematics*, vol. 8384. Springer Berlin Heidelberg, 2014, pp. 793–803.

[41] DIOP, T., JERGER, N. E., AND ANDERSON, J. Power modeling for heterogeneous processors. In *Proceedings of Workshop on General Purpose Processing Using GPUs* (2014), p. 90.

[42] DUAN, L., LI, B., AND PENG, L. Versatile prediction and fast estimation of architectural vulnerability factor from processor performance metrics. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture* (2009), IEEE, pp. 129–140.

[43] EDLER, J., AND HILL, M. Dinero IV Trace-Driven Uniprocessor Cache Simulator. http://pages.cs.wisc.edu/markhill/DineroIV, 1999.

[44] FANG, C., CAN, S., ONDER, S., AND WANG, Z. Instruction based memory distance analysis and its application to optimization. In *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)* (2005), IEEE, pp. 27–37.

[45] FANG, C., CARR, S., ÖNDER, S., AND WANG, Z. Reuse-distance-based miss-rate prediction on a per instruction basis. In *Proceedings of the 2004 workshop on Memory system performance* (2004), ACM, pp. 60–68.

[46] FARKAS, K. I., FLINN, J., BACK, G., GRUNWALD, D., AND ANDERSON, J. M. Quantifying the energy consumption of a pocket computer and a Java virtual machine. *ACM SIGMETRICS Performance Evaluation Review 28*, 1 (2000), 252–263.

[47] FEAUTRIER, P. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming 20*, 1 (1991), 23–53.

[48] FEAUTRIER, P. Some efficient solutions to the affine scheduling problem, part II: multidimensional time. *International Journal of Parallel Programming 21*, 6 (1992), 389–420.

[49] FERRANTE, J., SARKAR, V., AND THRASH, W. On estimating and enhancing cache effectiveness. In *International Workshop on Languages and Compilers for Parallel Computing* (1991), Springer, pp. 328–343.

[50] FLANAGAN, C., AND FREUND, S. N. Fasttrack: Efficient and precise dynamic race detection. In *Proc. of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09)* (2009), ACM.

[51] FLOYD, M., BROCK, B., WARE, M., RAJAMANI, K., DRAKE, A., LEFURGY, C., AND PESANTEZ, L. Harnessing the adaptive energy management features of the power7 chip. *HOT Chips 2010* (2010).

[52] FRAGUELA, B. B., DOALLO, R., AND ZAPATA, E. L. Automatic analytical modeling for the estimation of cache misses. In *Parallel Architectures and Compilation Techniques, 1999. Proceedings. 1999 International Conference on* (1999), IEEE, pp. 221–231.

[53] FRAGUELA, B. B., DOALLO, R., AND ZAPATA, E. L. Probabilistic miss equations: Evaluating memory hierarchy performance. *IEEE Transactions on Computers 52*, 3 (2003), 321–336.

[54] FRIGO, M., LEISERSON, C. E., PROKOP, H., AND RAMACHANDRAN, S. Cache-oblivious algorithms. In *Proc. of the 40th Annual Symposium on Foundations of Computer Science* (1999), IEEE.

[55] FRIGO, M., AND STRUMPEN, V. Cache oblivious stencil computations. In *Proc. of the 19th annual international conference on Supercomputing* (2005), ACM.

[56] FRIGO, M., AND STRUMPEN, V. The cache complexity of multithreaded cache oblivious algorithms. *Theory of Computing Systems 45*, 2 (2009), 203–233.

[57] FRUMKIN, M. A., AND VAN DER WIJNGAART, R. F. Tight bounds on cache use for stencil operations on rectangular grids. *Journal of the ACM (JACM) 49*, 3 (2002), 434–453.

[58] GACHET, P., MAURAS, C., QUINTON, P., AND SAOUTER, Y. Alpha du centaur: a prototype environment for the design of parallel regular alorithms. In *Proc. of the 3rd international conference on Supercomputing* (1989), ACM.

[59] GE, R., FENG, X., FENG, W.-C., AND CAMERON, K. W. Cpu miser: A performance-directed, run-time system for power-aware clusters. In *ICPP* (2007), pp. 18–18.

[60] GE, R., VOGT, R., MAJUMDER, J., ALAM, A., BURTSCHER, M., AND ZONG, Z. Effects of dynamic voltage and frequency scaling on a K20 GPU. In *ICPP* (2013), pp. 826–833.

[61] GHOSH, S., MARTONOSI, M., AND MALIK, S. Precise miss analysis for program transformations with caches of arbitrary associativity. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 1998), ASPLOS VIII, ACM, pp. 228–239.

[62] GHOSH, S., MARTONOSI, M., AND MALIK, S. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems (TOPLAS) 21*, 4 (1999), 703–746.

[63] GIRBAL, S., VASILACHE, N., BASTOUL, C., COHEN, A., PARELLO, D., SIGLER, M., AND TEMAM, O. Semi-automatic composition of loop transformations. *International Journal of Parallel Programming 34*, 3 (June 2006), 261–317.

[64] GODLIN, B., AND STRICHMAN, O. Inference rules for proving the equivalence of recursive procedures. *Acta Informatica 45*, 6 (2008), 403–439.

[65] GRIEBL, M., FEAUTRIER, P., AND LENGAUER, C. Index set splitting. *International Journal of Parallel Programming 28*, 6 (2000), 607–631.

[66] HARPER, J. S., KERBYSON, D. J., AND NUDD, G. R. Analytical modeling of set-associative cache behavior. *IEEE Transactions on Computers 48*, 10 (1999), 1009–1024.

[67] HILL, M. D., AND SMITH, A. J. Evaluating associativity in cpu caches. *IEEE Transactions on Computers 38*, 12 (1989), 1612–1630.

[68] HONG, C., BAO, W., COHEN, A., KRISHNAMOORTHY, S., POUCHET, L., RASTELLO, F., RAMANUJAM, J., AND SADAYAPPAN, P. Effective padding of multidimensional arrays to avoid cache conflict misses. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'16)* (2016), 129–144.

[69] HSU, C.-H., AND KREMER, U. The design, implementation, and evaluation of a compiler algorithm for CPU energy reduction. In *ACM SIGPLAN Notices* (2003), vol. 38, pp. 38–48.

[70] INTEL. Intel Math Kernel Library (Intel MKL). https://software.intel.com/en-us/intel-mkl.

[71] INTEL. Intel performance counter monitor. www.intel.com/software/pcm.

[72] JIMBOREAN, A., KOUKOS, K., SPILIOPOULOS, V., BLACK-SCHAFFER, D., AND KAXIRAS, S. Fix the code. don't tweak the hardware: A new compiler approach to voltage-frequency scaling. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (2014), ACM, p. 262.

[73] KALVALA, S., WARBURTON, R., AND LACEY, D. Program transformations using temporal logic side conditions. *ACM Trans. on Programming Languages and Systems (TOPLAS) 31*, 4 (2009), 14.

[74] KARFA, C., BANERJEE, K., SARKAR, D., AND MANDAL, C. Verification of loop and arithmetic transformations of array-intensive behaviors. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems 32*, 11 (2013), 1787–1800.

[75] KELLY, W., AND PUGH, W. A framework for unifying reordering transformations. Tech. rep., College Park, MD, USA, 1993.

[76] KIM, S., AND SOMANI, A. K. Area efficient architectures for information integrity in cache memories. *ACM SIGARCH Computer Architecture News 27*, 2 (1999), 246–255.

[77] KIM, S., AND SOMANI, A. K. Soft error sensitivity characterization for microprocessor dependability enhancement strategy. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on* (2002), IEEE, pp. 416–425.

[78] KONG, M., VERAS, R., STOCK, K., FRANCHETTI, F., POUCHET, L.-N., AND SADAYAPPAN, P. When polyhedral transformations meet simd code generation. In *ACM SIGPLAN Notices* (2013), vol. 48, ACM, pp. 127–138.

[79] KUNDU, S., TATLOCK, Z., AND LERNER, S. Proving optimizations correct using parameterized program equivalence. *ACM SIGPLAN Notices 44*, 6 (2009), 327–337.

[80] LI, J., AND MARTINEZ, J. F. Dynamic power-performance adaptation of parallel computation on chip multiprocessors. In *HPCA* (2006), pp. 77–87.

[81] LI, X., ADVE, S. V., BOSE, P., AND RIVERS, J. A. Softarch: an architecture-level tool for modeling and analyzing soft errors. In *2005 International Conference on Dependable Systems and Networks (DSN'05)* (2005), IEEE, pp. 496–505.

[82] LI, X., ADVE, S. V., BOSE, P., AND RIVERS, J. A. Online estimation of architectural vulnerability factor for soft errors. In *Computer Architecture, 2008. ISCA'08. 35th International Symposium on* (2008), IEEE, pp. 341–352.

[83] LIM, A. W., AND LAM, M. S. Maximizing parallelism and minimizing synchronization with affine transforms. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1997), POPL '97, ACM, pp. 201–214.

[84] LORCH, J. R., AND SMITH, A. J. Improving dynamic voltage scaling algorithms with pace. In *ACM SIGMETRICS Performance Evaluation Review* (2001), vol. 29, ACM, pp. 50–61.

[85] MANSKY, W., AND GUNTER, E. A framework for formal verification of compiler optimizations. In *Interactive Theorem Proving*. Springer, 2010.

[86] MCCALPIN, J. D. Stream: Sustainable memory bandwidth in high performance computers. Tech. rep., University of Virginia, Charlottesville, Virginia, 1991-2007. A continually updated technical report. http://www.cs.virginia.edu/stream/.

[87] MEI, X., YUNG, L. S., ZHAO, K., AND CHU, X. A measurement study of GPU DVFS on energy conservation. In *Proceedings of the Workshop on Power-Aware Computing and Systems* (2013), p. 10.

[88] MITRA, S., SEIFERT, N., ZHANG, M., SHI, Q., AND KIM, K. S. Robust system design with built-in soft-error resilience. *Computer 38*, 2 (2005), 43–52.

[89] MUKHERJEE, S. S., WEAVER, C., EMER, J., REINHARDT, S. K., AND AUSTIN, T. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture* (2003), IEEE Computer Society, p. 29.

[90] NECULA, G. C. Translation validation for an optimizing compiler. *ACM SIGPLAN Notices 35*, 5 (2000), 83–94.

[91] NETLIB. Netlib blas. http://www.netlib.org/blas/index.html.

[92] OPENCV. Opencv: Open source computer vision library. http://opencv.org.

[93] OPPEN, D. C. A $2^{2^{2^{pn}}}$ upper bound on the complexity of presburger arithmetic. *Journal of Computer and System Sciences 16*, 3 (1978), 323–332.

[94] PAULSON, L. C. Isabelle Page. https://www.cl.cam.ac.uk/research/hvg/Isabelle.

[95] POUCHET, L. Polyopt/C: A polyhedral optimizer for the rose compiler, 2011.

[96] POUCHET, L.-N., ZHANG, P., SADAYAPPAN, P., AND CONG, J. Polyhedral-based data reuse optimization for configurable computing. In *FPGA* (2013).

[97] PROKOP, H. *Cache-oblivious algorithms*. PhD thesis, Massachusetts Institute of Technology, 1999.

[98] QUINLAN, D., LIAO, C., MATZKE, R., SCHORDAN, M., PANAS, T., VUDUC, R., AND YI, Q. ROSE Web Page. http://www.rosecompiler.org, 2014.

[99] QUINTON, P., AND VAN DONGEN, V. The mapping of linear recurrence equations on regular arrays. *Journal of VLSI signal processing systems for signal, image and video technology 1*, 2 (1989), 95–113.

[100] RAJOPADHYE, S. V., PURUSHOTHAMAN, S., AND FUJIMOTO, R. M. On synthesizing systolic arrays from recurrence equations with linear dependencies. In *Proc. of the 16th annual conference on Foundations of Software Technology and Theoretical Computer Science* (1986), Springer.

[101] RAMAN, R., ZHAO, J., SARKAR, V., VECHEV, M. T., AND YAHAV, E. Scalable and precise dynamic datarace detection for structured parallelism. In *Proc. of the 2012 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12)* (2012), ACM.

[102] RAMAPRASAD, H., AND MUELLER, F. Bounding worst-case data cache behavior by analytically deriving cache reference patterns. In *11th ieee real time and embedded technology and applications symposium* (2005), IEEE, pp. 148–157.

[103] REBAUDENGO, M., SONZA REORDA, M., AND VIOLANTE, M. An accurate analysis of the effects of soft errors in the instruction and data caches of a pipelined microprocessor. In *Proceedings of the conference on Design, Automation and Test in Europe-Volume 1* (2003), IEEE Computer Society, p. 10602.

[104] RIVERA, G., AND TSENG, C.-W. Data transformations for eliminating conflict misses. *SIGPLAN Not. 33*, 5 (May 1998), 38–49.

[105] SAPUTRA, H., KANDEMIR, M., ET AL. Energy-conscious compilation based on voltage scaling. In *In Proc. ACM SIGPLAN Joint Conference on Languages, Compilers, and Tools for Embedded Systems & Software and Compilers for Embedded Systems* (2002), Citeseer.

[106] SARKAR, V. Automatic selection of high order transformations in the IBM XL Fortran compilers. *IBM J. Res. & Dev. 41*, 3 (May 1997).

[107] SARKAR, V., AND MEGIDDO, N. An analytical model for loop tiling and its solution. In *Performance Analysis of Systems and Software, 2000. ISPASS. 2000 IEEE International Symposium on* (2000), IEEE, pp. 146–153.

[108] SAVAGE, S., BURROWS, M., NELSON, G., SOBALVARRO, P., AND ANDERSON, T. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. on Computer Systems (TOCS) 15*, 4 (1997), 391–411.

[109] SCHORDAN, M., LIN, P.-H., QUINLAN, D., AND POUCHET, L.-N. Verification of polyhedral optimizations with constant loop bounds in finite state space computations. In *Proc. of the 6th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. Springer, 2014.

[110] SHIRAKO, J., POUCHET, L.-N., AND SARKAR, V. Oil and water can mix: Reconciling polyhedral and ast transformations. In *IEEE/ACM Conference on Supercomputing (SC'14)* (2014), IEEE.

[111] SHIRAKO, J., SHARMA, K., FAUZIA, N., POUCHET, L.-N., RAMANUJAM, J., SADAYAPPAN, P., AND SARKAR, V. Analytical bounds for optimal tile size selection. In *International Conference on Compiler Construction* (2012), Springer, pp. 101–121.

[112] SHRIVASTAVA, A., LEE, J., AND JEYAPAUL, R. Cache vulnerability equations for protecting data in embedded processor caches from soft errors. In *ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'10)* (2010), pp. 143–152.

[113] SINGH, J. P., STONE, H. S., AND THIEBAUT, D. F. A model of workloads and its use in miss-rate prediction for fully associative caches. *IEEE transactions on computers 41*, 7 (1992), 811–825.

[114] SKADRON, K., STAN, M. R., SANKARANARAYANAN, K., HUANG, W., VELUSAMY, S., AND TARJAN, D. Temperature-aware microarchitecture: Modeling and implementation. *ACM Trans. Archit. Code Optim. 1*, 1 (Mar. 2004), 94–125.

[115] SRIDHARAN, V., ASADI, H., TAHOORI, M. B., AND KAELI, D. Reducing data cache susceptibility to soft errors. *IEEE Transactions on Dependable and Secure Computing 3*, 4 (2006), 353–364.

[116] SRIDHARAN, V., AND KAELI, D. R. Using pvf traces to accelerate avf modeling. In *Proceedings of the IEEE workshop on silicon errors in logic-system effects* (2010), pp. 23–24.

[117] TANG, Y., CHOWDHURY, R., LUK, C.-K., AND LEISERSON, C. E. Coding stencil computations using the pochoir stencil-specification language. In *Poster session presented at the 3rd USENIX Workshop on Hot Topics in Parallelism* (2011).

[118] TANG, Y., CHOWDHURY, R. A., KUSZMAUL, B. C., LUK, C.-K., AND LEISERSON, C. E. The pochoir stencil compiler. In *Proc. of the 32rd annual ACM symposium on Parallelism in algorithms and architectures* (2011), ACM.

[119] VALIEV, M., BYLASKA, J., GOVIND, N., KOWALSKI, K., STRAATSMA, T. P., VAN D., H. J. J., WANG, D., NIEPLOCHA, J., APRA, E., WINDUS, L., ET AL.

Nwchem: a comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications* (2010), 1477–1489.

[120] VERA, X., ABELLA, J., GONZÁLEZ, A., AND LLOSA, J. Optimizing program locality through cmes and gas. In *Parallel Architectures and Compilation Techniques, 2003. PACT 2003. Proceedings. 12th International Conference on* (2003), IEEE, pp. 68–78.

[121] VERA, X., ABELLA, J., LLOSA, J., AND GONZÁLEZ, A. An accurate cost model for guiding data locality transformations. *ACM Trans. Program. Lang. Syst. 27*, 5 (Sept. 2005), 946–987.

[122] VERA, X., BERMUDO, N., LLOSA, J., AND GONZÁLEZ, A. A fast and accurate framework to analyze and optimize cache memory behavior. *ACM Transactions on Programming Languages and Systems (TOPLAS) 26*, 2 (2004), 263–300.

[123] VERA, X., AND XUE, J. Let's study whole-program cache behaviour analytically. In *High-Performance Computer Architecture, 2002. Proceedings. Eighth International Symposium on* (2002), IEEE, pp. 175–186.

[124] VERDOOLAEGE, S. isl: An integer set library for the polyhedral model. In *The 3rd International Congress on Mathematical Software (ICMS'10)*. Springer, 2010.

[125] VERDOOLAEGE, S. Counting affine calculator and applications. In *The 1st International Workshop on Polyhedral Compilation Techniques (IMPACT'11)* (2011).

[126] VERDOOLAEGE, S., AND GROSSER, T. Polyhedral extraction tool. In *Second International Workshop on Polyhedral Compilation Techniques (IMPACT12), Paris, France* (2012).

[127] VERDOOLAEGE, S., JANSSENS, G., AND BRUYNOOGHE, M. Equivalence checking of static affine programs using widening to handle recurrences. *ACM Trans. on Programming Languages and Systems (TOPLAS) 34*, 3 (2012), 11.

[128] VERDOOLAEGE, S., SEGHIR, R., BEYLS, K., LOECHNER, V., AND BRUYNOOGHE, M. Counting integer points in parametric polytopes using Barvinok's rational functions. *Algorithmica 48*, 1 (June 2007), 37–66.

[129] WANG, W.-H., AND BAER, J.-L. Efficient trace-driven simulation method for cache performance analysis. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 1990), SIGMETRICS '90, ACM, pp. 27–36.

[130] WEAVER, C., EMER, J., MUKHERJEE, S. S., AND REINHARDT, S. K. Techniques to reduce the soft error rate of a high-performance microprocessor. In *ACM*

SIGARCH Computer Architecture News (2004), vol. 32, IEEE Computer Society, p. 264.

[131] WOLFE, M. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.

[132] XUE, J., AND VERA, X. Efficient and accurate analytical modeling of whole-program data cache behavior. *IEEE Transactions on Computers 53*, 5 (2004), 547–566.

[133] YOU, D., AND CHUNG, K.-S. Dynamic voltage and frequency scaling framework for low-power embedded GPUs. *Electronics letters 48*, 21 (2012), 1333–1334.

[134] YUKI, T., AND RAJOPADHYE, S. Folklore confirmed: Compiling for speed = compiling for energy. In *LCPC* (2014), pp. 169–184.

[135] ZHANG, W. Computing cache vulnerability to transient errors and its implication. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'05)* (2005), pp. 427–435.

[136] ZUO, W., LI, P., CHEN, D., POUCHET, L.-N., ZHONG, S., AND CONG, J. Improving polyhedral code generation for high-level synthesis. In *IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'13)* (2013), IEEE.