Copyright
by
Bertrand Allen Maher
2010

The Dissertation Committee for Bertrand Allen Maher
certifies that this is the approved version of the following dissertation:

Atomic Block Formation for Explicit Data Graph Execution Architectures

Committee:

Kathryn S. McKinley, Supervisor

Douglas C. Burger, Supervisor

Stephen W. Keckler

Scott A. Mahlke

Keshav Pingali

Atomic Block Formation for Explicit Data Graph Execution Architectures

by

Bertrand Allen Maher, B.S., M.S.C.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2010

Dedicated to my parents Howard and Rita Maher.

Acknowledgments

I would like to thank the numerous people who have helped me along the way to completing this dissertation. I thank my committee, Kathryn McKinley, Doug Burger, Steve Keckler, Keshav Pingali, and Scott Mahlke, for their valuable feedback on my research. Their insight has unquestionably improved this dissertation.

I am particularly indebted to my advisors, Doug Burger and Kathryn McKinley, for their supervision of this work. I count myself lucky to have had scientists of their caliber to oversee my research. Doug and Kathryn have always challenged me to improve my ideas, yet also encouraged me that those ideas are worth pursuing. I hope that my career in computer science will make them proud.

I thank Steve Keckler for his leadership of the TRIPS project along with Doug and Kathryn. My career thus far has been enriched by working on such an ambitious research project as a graduate student.

I thank the TRIPS team without whom this work would have been impossible. My fellow TRIPS compiler collaborators, Katie Coons, Jon Gibson, Behnam Robatmili, Aaron Smith, and Bill Yoder have taught me a great deal over the years. I thank the TRIPS hardware team, Karu Sankaralingam, Ramdas Nagarajan, Sibi Govindan, Divya Gulati, Paul Gratz, Heather Hanson,

Changkyu Kim, Haiming Liu, Nitya Ranganathan, Simha Sethumadhavan, and Premkishore Shivakumar, for their contributions. I particularly thank Mark Gebhart for his excellent work on the TRIPS performance evaluation.

I thank the many friends I've made in graduate school, Walter Chang, Jeff Diamond, Curtis Dunham, Boris Grot, Maria Jump, Alison Norman, and Jenn Sartor, for numerous excellent conversations on matters both technical and not, and for attending practice talks, reading papers, and generally enriching my life over the past several years.

I am grateful to my best friend and soon-to-be wife Katie Coons for always being there for me during our time in graduate school. As a friend and fellow student she has been a continuous source of encouragement; as a researcher she has been an inspiration. I look forward to the next chapter of our lives together.

Finally, I thank my parents Rita and Howard and my sister Marie for their support and love. My parents taught me to love learning from an early age, and their example inspired me to embark on a career in science. I thank them for always encouraging me to do my best, and for their unconditional love. I dedicate this dissertation to them.

Bertrand Allen Maher

The University of Texas at Austin

August 2010

Atomic Block Formation for Explicit Data Graph Execution Architectures

Publication No. ______________

Bertrand Allen Maher, Ph.D.
The University of Texas at Austin, 2010

Supervisors: Kathryn S. McKinley
Douglas C. Burger

Limits on power consumption, complexity, and on-chip latency have focused computer architects on power-efficient designs that exploit parallelism. One approach divides programs into atomic blocks of operations that execute semi-independently, which efficiently creates a large window of potentially concurrent operations. This dissertation studies the intertwined roles of the compiler, architecture, and microarchitecture in achieving efficiency and high performance with a block-atomic architecture.

For such an architecture to achieve high performance the compiler must form blocks effectively. The compiler must create large blocks of instructions to amortize the per-block overhead, but control flow and content restrictions limit the compiler's options. Block formation should consider factors such of frequency of execution, block size such as selecting control-flow paths that

are frequently executed, and exploiting locality of computations to reduce communication overheads.

This dissertation determines what characteristics of programs influence block formation and proposes techniques to generate effective blocks. The first contribution is a method for solving phase-ordering problems inherent to block formation, mitigating the tension between block-enlarging optimizations—if-conversion, tail duplication, loop unrolling, and loop peeling—as well as scalar optimizations. Given these optimizations, analysis shows that the remaining obstacles to creating larger blocks are inherent in the control flow structure of applications, and furthermore that any fixed block size entails a sizable amount of wasted space. To eliminate this overhead, this dissertation proposes an architectural implementation of variable-size blocks that allow the compiler to dramatically improve block efficiency.

We use these mechanisms to develop policies for block formation that achieve high performance on a range of applications and processor configurations. We find that the best policies differ significantly depending on the number of participating cores. Using machine learning, we discover generalized policies for particular hardware configurations and find that the best policy varies significantly between applications and based on the number of parallel resources available in the microarchitecture. These results show that effective and efficient block-atomic execution is possible when the compiler and microarchitecture are designed cooperatively.

Table of Contents

Acknowledgments **v**

Abstract **vii**

List of Tables **xii**

List of Figures **xiv**

Chapter 1. Introduction **1**

1.1 Dissertation Contributions 5

1.2 Dissertation Organization 8

Chapter 2. Related Work **11**

2.1 Block Formation 11

2.1.1 Traces, Superblocks and Treegions 12

2.1.2 Hyperblocks 13

2.2 Phase-Ordering Problems 15

2.3 Atomicity 18

Chapter 3. EDGE Architectures **21**

3.1 Advantages of Block-atomic Architectures 21

3.2 EDGE ISAs 23

3.3 Compiling for EDGE ISAs 26

3.4 TRIPS Microarchitecture 29

3.5 TFlex Microarchitecture 30

Chapter 4. Iterative Block Formation **34**
4.1 Introduction . . . 35
4.2 Phase Ordering Challenges . . . 38
4.3 Iterative Block Formation . . . 40
4.3.1 Head and Tail Duplication . . . 40
4.3.2 Incremental Block Formation . . . 43
4.4 Policy . . . 46
4.5 TRIPS compiler . . . 49
4.6 Experimental Results . . . 50
4.6.1 Comparison to Static Phase Ordering . . . 52
4.6.2 VLIW and EDGE Heuristics . . . 55
4.6.3 Estimated Performance with Block Counts . . . 57
4.7 Summary . . . 61

Chapter 5. Compiler Evaluation **63**
5.1 Comparative Performance Evaluation . . . 65
5.1.1 Methodology . . . 65
5.1.2 Performance Results . . . 68
5.2 Block Size Efficiency . . . 71
5.2.1 Methodology . . . 72
5.2.2 Block Fullness . . . 73
5.2.3 Control Flow Limitations . . . 75
5.3 Summary . . . 89

Chapter 6. Variable-Size Blocks **93**
6.1 Atomicity and Composability in EDGE Architectures . . . 96
6.1.1 Block-Atomic Execution . . . 96
6.1.2 EDGE Support for Composability . . . 97
6.2 Instruction Set Support . . . 99
6.3 Microarchitecture . . . 100
6.3.1 Fixed-Size Blocks . . . 101
6.3.2 Variable-Size Blocks . . . 102
6.4 Architectural Results . . . 105

6.4.1 Methodology 105
6.4.2 One Block Per Core 107
6.4.3 Multiple Blocks Per Core 109
6.4.4 Variable-Size Blocks 112
6.5 Summary 117

Chapter 7. Block Formation Heuristics **121**
7.1 Compiler Structure 124
7.2 Default Block Formation Heuristic 127
7.3 Block Formation Heuristics 128
7.4 Implemented Features 131
7.5 Machine Learning Results 135
7.5.1 Learning Methodology 136
7.5.2 Learned Heuristics Discussion 140
7.5.3 SPEC CPU Results 143
7.6 Summary 144

Chapter 8. Conclusions **146**
8.1 Dissertation Contributions 148
8.2 Future Directions 151
8.3 Final Thoughts 154

Bibliography **156**

Vita **170**

List of Tables

3.1 TRIPS ISA block constraints 25

4.1 Percent improvement in cycle counts of EDGE blocks over basic blocks (BB) with various orderings of Unrolling (U), Peeling (P) Incremental If-conversion (I), and Scalar Optimizations (O). Parentheses indicate merged phases. 51

4.2 Static count of blocks merged/tail duplicated blocks/unrolled iterations/peeled iterations (m/t/u/p), with various orderings of Unrolling (U), Peeling (P) Incremental If-conversion (I), and Scalar Optimizations (O). Parentheses indicate merged phases. 54

4.3 Percent improvement in cycle count over basic blocks (BB) using VLIW heuristics, VLIW with iterative optimization, depth-first (DF) and breadth-first (BF) EDGE heuristics. 56

4.4 Percent improvement in block counts of SPEC benchmarks over basic blocks (BB) with various combinations and orderings of Unrolling (U), Peeling (P) If-conversion (I), and Scalar Optimizations (O). Parentheses indicate merged phases. 59

4.5 Percent improvement in cycle counts of SPEC benchmarks over basic blocks (BB) with various combinations and orderings of Unrolling (U), Peeling (P) If-conversion (I), and Scalar Optimizations (O). Parentheses indicate merged phases. 60

5.1 Reference platforms. 66

5.2 Performance counter statistics for SPEC. 70

5.3 Per-benchmark efficiency of blocks for fixed maximum sizes of 32, 64, and 128 instructions. 75

6.1 Microarchitectural parameters of each TFlex core 106

6.2 Efficiency, in terms of real instructions to total words fetched, of variable-size blocks with different granularities. 114

6.3 Per-benchmark efficiency of blocks for fixed maximum sizes of 32, 64, and 128 instructions. 115

7.1 Operators used to construct cost functions. 126

7.2 Boolean-valued features 132
7.3 Real-valued features 133
7.4 Real-valued features, continued 134
7.5 Heuristics with the best geometric mean speedup over baseline, both hand-written and machine-learned. 140

List of Figures

3.1 Phases of the TRIPS compiler. 26
3.2 TRIPS processor die photo with outlined tiles. 29
3.3 Microarchitecture of one TFlex Core 31
3.4 Mapping blocks to composable TFlex cores 31

4.1 Block formation example. 39
4.2 Classical tail duplication. 41
4.3 Head duplication implements peeling. 42
4.4 Head duplication implements unrolling. 43
4.5 Iterative block formation algorithm. 44
4.6 Compiler flow with iterative block formation. 49
4.7 Cycle count reductions versus block count reductions. 58

5.1 Speedup on SPECINT relative to Core 2/gcc. 67
5.2 Speedup on SPECFP relative to Core 2/gcc. 68
5.3 Dynamic average total and useful instructions per block with various maximum block sizes. 74
5.4 Distribution of block sizes in SPEC CPU2000 benchmarks, weighted by execution frequency. The categories indicate the reason for the compiler's inability to merge that block with the next in the execution trace. 77
5.5 Distribution of block sizes in SPEC CPU2000 benchmarks, after implementing call merging and epilogue duplication. While block fullness is improved, performance suffers. 80
5.6 Cut analysis of 256.bzip2 82
5.7 Cut analysis of 186.crafty 83
5.8 Cut analysis of 254.gap 83
5.9 Cut analysis of 176.gcc 83
5.10 Cut analysis of 164.gzip 84
5.11 Cut analysis of 181.mcf 84

5.12 Cut analysis of 197.parser . . . 84
5.13 Cut analysis of 253.perlbmk . . . 85
5.14 Cut analysis of 300.twolf . . . 85
5.15 Cut analysis of 175.vpr . . . 86
5.16 Cut analysis of 188.ammp . . . 86
5.17 Cut analysis of 173.applu . . . 86
5.18 Cut analysis of 301.apsi . . . 87
5.19 Cut analysis of 179.art . . . 87
5.20 Cut analysis of 183.equake . . . 88
5.21 Cut analysis of 177.mesa . . . 88
5.22 Cut analysis of 172.mgrid . . . 88
5.23 Cut analysis of 171.swim . . . 89
5.24 Cut analysis of 168.wupwise . . . 90

6.1 TFlex: A Composable Chip Multiprocessor EDGE Architecture 98
6.2 Performance with one fixed-size block per core and maximum fixed block size of 32, 64, and 128 instructions. Thus, fewer hardware resources are required with smaller maximum block sizes. . . . 108
6.3 Performance with 128 instructions per core and fixed-size blocks. Thus, with maximum block sizes of 32, 64, and 128 instructions, there are 4, 2, and 1 blocks per core, respectively, and all configurations require the same number of issue queue slots. . . . 110
6.4 Performance with 8-instruction granularity variability in the instruction window with various maximum block sizes using 128 instructions per core. . . . 113

7.1 An example cost function that selects first by branch probability, then by the inverse of block size. . . . 126
7.2 Speedup of learned heuristics on microbenchmarks. Each bar is normalized to the performance of the baseline heuristic running on the same number of cores as the learned heuristic. . . . 137
7.3 Speedup of heuristics learned for 1-, 8-, and 32-cores over baseline heuristic on one core. . . . 143

Chapter 1

Introduction

Since 2006 improvements in single-threaded processor performance have dramatically slowed. Efforts to increase clock rate by increasing pipeline depth have largely been abandoned due to excessive power consumption. Furthermore, microarchitectural techniques to extract instruction-level parallelism (ILP) from a sequential instruction stream have made minimal gains, due to limits on power consumption and design complexity. In response to these limitations, recent industrial designs place multiple processor cores on one chip to provide greater performance.

To achieve performance improvements with multicore processors, software must be concurrent. Most software written today is not highly concurrent, both because programmers have depended on single-thread performance improvements and because parallel programming is more difficult than sequential programming. Although some code can be automatically parallelized by a compiler, most currently cannot. Most of the burden of parallel programming therefore falls to human programmers, who will have to spend more time writing complex—and difficult to debug—parallel algorithms and less time adding useful features to their software.

As an alternative to multicore processors, block-atomic instruction set architectures (ISA) can enable a processor to extract parallelism from a sequential programming model more efficiently than out-of-order superscalar processors. In such an ISA, large regions of instructions are grouped together into blocks, which the processor fetches, executes, and commits as a unit. By executing large blocks the processor can avoid much of the control and bookkeeping overhead present in an instruction-atomic processor. Block-atomic ISAs present a familiar sequential programming model to the programmer, so performance improvements do not require explicit parallelization of software.

For block-atomic architectures to be a plausible complement to multicore designs, compilers must be able to form blocks that execute with high performance. If compilers can perform this optimization automatically, then block-atomic processors will be able to achieve greater performance than superscalar processors without fundamentally changing the programming model. High performance blocks ideally contain a large number of instructions in order to expose parallelism and amortize bookkeeping overheads. The blocks must also contain regions of code that maximize performance and efficiency in the common case. By selecting a large, fixed block size, a block-atomic ISA can limit hardware complexity while remaining a feasible compiler target.

This dissertation investigates the proper balance between hardware and software in the design of a block-atomic ISA, focusing particularly on the architectural block size. Selecting a block size entails tradeoffs between static and dynamic optimization opportunities. Large blocks create the possibility of

efficient execution within a large region of code, but rely more heavily on the compiler to form blocks. Smaller blocks depend more on dynamic techniques such as branch prediction and register renaming to create parallelism within a large window. This thesis will quantify the performance, power, and complexity tradeoffs among these design possibilities and the compiler's capabilities.

A block-atomic architecture relies on the compiler's block formation heuristics for good performance. There are three parameters within the compiler's control. The compiler can often improve performance by constructing large blocks. Larger architectural blocks are more difficult for the compiler to fill, and usually require code-expanding techniques such as loop unrolling, but statically create more opportunities for parallel execution. Predication allows the compiler to eliminate branches, possible avoiding mispredictions, although at the cost of code expansion and reducing the number of useful instructions in the processors window. Block formation also encapsulates dependences within blocks, creating opportunities for efficient execution.

This dissertation presents compiler techniques for generating blocks for a block-atomic architecture. These techniques include both the mechanisms that must be implemented to ensure that the compiler has flexibility to include the most profitable control paths, as well as the policy that guides its decisions, and the techniques used to discover policy. This dissertation analyzes the ability of the compiler to form large blocks in the presence of control flow structures that inhibit block formation. To understand the effects of block formation the correlation of performance with block size will be explored

and understood. To ameliorate cases where the compiler cannot achieve high utilization of fixed-size blocks, an ISA and microarchitecture that supports variable-sized blocks will be presented.

To form optimized blocks the compiler must be free to choose any control flow path to include in a block, and must be able to pack each block full of optimized code. Prior work in this area applies transformations such as loop peeling and loop unrolling prior to block formation in order to expose large regions of code for the block formation algorithm. This dissertation describes a transformation, called head duplication, that combines loop unrolling and peeling into the block formation process, so that the compiler can weigh expanding a loop versus including other paths of control flow. To ensure that the code produced by the block formation stage is highly optimized, the compiler iterates phases of block merging and scalar optimizations, which serve to compact the block and thus enable additional merging phases.

EDGE block formation must be guided by policy, which is generally a heuristic function that informs the compiler which basic blocks should be included in a block. Previous work has found functions that use a combination of basic block size, execution frequency, and dependence height to select blocks for inclusion. While these features will likely be used for block-atomic architectures, additional features such as minimizing communication between blocks will become important. Developing these heuristics is a time-consuming task for a compiler writer. This dissertation provides the reader with insight into what heuristics and features are likely to perform well, and provide a

methodology for discovering good heuristics by applying genetic programming techniques to a variety of microarchitectural configurations.

Designers of future block-atomic architectures should know the capabilities of compilers when setting parameters such as the block size. This dissertation provides a breakdown of block sizes in programs, indicating which control-flow features most inhibit block formation, and providing solutions to these limitations where feasible. Block size is typically well correlated with performance, but there is a point of diminishing returns as well. This dissertation shows the relationship between block size and performance, and identify the other variables that affect this relationship.

Given the limitations of the compiler uncovered by the block size analysis, the processor must be able to recoup some of the efficiency loss when blocks cannot be completely filled. This dissertation describes architectural and microarchitectural support for variable-sized blocks. Variable-sized blocks give a block-atomic architecture more of the resource flexibility enjoyed by instruction-level processors, for example, the ability to use only the required amount of space in the the instruction cache and issue queue. This flexibility mitigates the impact of small blocks, but still allows the processor to retain the advantages of block-atomic execution when large blocks are available.

1.1 Dissertation Contributions

This dissertation evaluates the software capabilities necessary to compile code for block-atomic EDGE architectures. The following are the contri-

butions.

- We propose *iterative block formation*, which is an approach that is well-suited to constrained, block-atomic architectures. This algorithm integrates scalar optimizations with block formation, ensuring that blocks are maximally full even when optimizations are taken into account.

- We show that the transformations of loop unrolling and loop peeling are essentially equivalent to tail duplication. We call those transformations *head duplication*, to emphasize that while tail duplication removes side entrances to a trace at forward edges, head duplication removes side entrances at loop headers, i.e., back edges. We use this new transformation to combine peeling and unrolling into a single-pass algorithm as part of block formation.

- We evaluate the performance of the compiler for the TRIPS system using prototype hardware. We show that the TRIPS compiler can successfully compile the industry standard SPEC CPU2000 benchmark suite. We compare performance to the leading commercial processor platform, Intel's Core 2, and show that while TRIPS performance is promising on compute-intensive floating-point benchmarks, it struggles on control-intensive integer benchmarks

- We evaluate the ability of an aggressive, block-forming compiler to construct large blocks on a suite of realistic benchmarks. We show that the

average dynamically-executed block is relatively under-full and categorize the control-flow constraints that prevent the compiler from filling blocks. We also show the block-sizes have a "U-shaped" distribution—smaller blocks and larger blocks are relatively more common than medium sizes, thus arguing for a flexible microarchitectural block size.

- We propose modifications to an existing EDGE ISA, the TRIPS processor ISA, that allow the underlying microarchitecture to support variable-size blocks. Using this ISA, we propose microarchitectural support for variable-size blocks, which improves utilization of the instruction cache and the instruction window.

- We evaluate the impact of block size on performance, in microprocessors that support both fixed-size and variable-size blocks. We see that performance varies widely across benchmarks. Compute-intensive benchmarks benefit from large blocks compared to small blocks. Control-intensive benchmarks, however, lose little performance from small block sizes and would benefit from having more blocks in flight. We show that an architecture supporting variable-size blocks is able to achieve the "best of both worlds" performance.

- We explore the space of policies for EDGE architectures using machine learning. By feeding a large number of heuristics into a genetic programming system, we cover a significant extent of the space of possible policies. We extract meaning, where possible, from the learned policies

to determine the characteristics of high-performing EDGE blocks.

- We determine that the best policy for forming blocks depends significantly on the core count of the composed processor. For small configurations, little or no predication is desirable due to limited machine resources. For larger configurations more aggressive predication works well.

1.2 Dissertation Organization

This dissertation is organized as follows. Chapter 2 reviews related work. We focus on a few distinct areas. The use of architectural atomicity in processors and the compiler's ability to take advantage of atomicity is the key area of related work. We also focus on compiler approaches to constructing blocks, regardless of whether the blocks are used as an architectural atomic unit.

Chapter 3 describes EDGE architectures and the microarchitectures that will be evaluated in this dissertation. While EDGE architectures themselves are described in detail in other dissertations, an overview is necessary to give context to this work.

Chapter 4 describes compiler support necessary to form large blocks in the presence of ISA constraints. This chapter encompasses iterative block formation and head duplication, which together allow the compiler to form optimized blocks conforming to the ISAs block restrictions. Iterative block

formation allows the compiler to easily consider the constraints and the effects of optimizations with each merge. Head duplication enables the compiler to perform loop unrolling and peeling concurrently with block formation in a single pass algorithm, which resolves the phase ordering tension between these optimizations.

Chapter 5 evaluates the performance of the TRIPS compiler. We compare performance on compiled SPEC CPU2000 benchmarks to several commercial platforms and demonstrate shortcomings in performance. To understand these results, we evaluate of the space efficiency of blocks and analyzes what code features commonly prevent the compiler from forming completely full blocks. We show that function calls and small blocks are the primary cause of small blocks, and that overall the distribution is somewhat bimodal, with clusters at small and large sizes.

Chapter 6 describes the effect of granularities of fixed- and variable-size blocks on the performance of EDGE processors. We evaluate a variety of configurations, varying the architectural size of blocks and the number of blocks per core. We show that several benchmarks scale well to high core counts and large block sizes, while those that do not can attain good performance at low core counts by executing larger numbers of smaller blocks. We propose an ISA and microarchitecture for supporting variable-size blocks that can execute both classes of programs with high performance and efficiency.

Chapter 7 describes policies that attain high performance on EDGE processors. Because the space of possible policies is highly complex, we ex-

plore it using genetic programming techniques. Learning specialized heuristics per-benchmark can achieve significant speedups, while learning generalized heuristics gives us insight into general principles of policy for EDGE architectures. Learning policies for a variety of configurations shows that heuristics are quite different for small numbers of participating cores than for large numbers, as predication becomes less effective at smaller core counts. Chapter 8 summarizes the work and concludes.

Chapter 2

Related Work

Related work falls into three categories. The first category is the use of blocks as a compiler and hardware abstraction, mechanisms for forming blocks, and heuristics for improving performance of block formation. The second category concerns compiler solutions to phase ordering problems, especially those relating to predication. The third category contains architectural and compiler support for atomicity.

2.1 Block Formation

Compilers have used blocks of instructions as a useful abstraction since the use of basic blocks in the FORTRAN compiler [8]. Since then, compilers have included more advanced types of code regions. Typically the goal of creating large regions in the compiler is to broaden the scope of instruction scheduling. Frequently, control-flow transformations are necessary to remove constraints within blocks. In this section we overview techniques for handling large code regions. Section 2.1.1 discusses traces and superblocks, which VLIW compilers use commonly. Section 2.1.2 discusses hyperblocks at length, which closely match the control-flow model of EDGE blocks.

2.1.1 Traces, Superblocks and Treegions

Fisher pioneered *trace scheduling* for VLIW architectures, which attempts to improve execution time by scheduling instructions across multiple basic blocks [24]. This scheme uses static branch prediction to identify frequently executed program paths (traces) at compile time. Trace scheduling optimizes traces by speculatively scheduling instructions in earlier basic blocks and pushing correcting code on to the less frequently taken paths. Such transformations allow the compiler to move instructions around branches to improve the global schedule. While effective, its drawbacks include code and compiler complexity due to side trace entrances and the possibility of exponential code expansion [34, 40].

Superblock scheduling also attempts to improve global instruction scheduling, and improves over trace scheduling by eliminating the need for side entrances [34]. To form superblocks, a compiler uses *tail duplication*, which replicates instructions/blocks after a side entrance, and redirects the side entrance to this copy. After tail duplication each super block is a single-entry, multiple-exit region of code. This structure gives more freedom to the scheduler and optimizer, because the compiler can break constraining dependences outside of the superblock. Because the effectiveness of superblock compilation relies on the accuracy of profiling information, researchers have proposed techniques for mitigating the effect of profile variation on superblock compilation. The *speculative hedge* heuristic for superblock scheduling attempts to minimize execution time across all paths to account for such variations [21].

A related approach, *treegion* scheduling, uses a different type of scheduling region to achieve profile tolerance and improve machine utilization. Treegions, like superblocks, are single-entry, multiple-exit regions; unlike superblocks, however, treegions may contain basic blocks from multiple paths of control [30]. A treegion consists of a tree of basic blocks; thus there can be multiple paths of control, but no merge points. Treegion compilation does make use of tail duplication, however, to increase the scope of scheduling by eliminating merge points. These regions increase the ability of the compiler to schedule instructions from multiple code paths and potentially make better use of a wide machine. By including multiple paths, a treegion compiler can limit the effect of profile variation.

2.1.2 Hyperblocks

The hyperblock generalizes the superblock to enable effective use of predicated execution [47]. A hyperblock, like a superblock, is a single-entry, multiple-exit region of code, but a hyperblock can contain multiple paths of control by replacing control flow instructions with predicates within the hyperblock. The goal of hyperblock formation is to eliminate branching and maximize ILP, while avoiding over-commitment of processor resources [46]. A standard hyperblock-forming compiler performs heuristic-driven hyperblock formation, followed by block-enlargement optimizations (such as predicated loop peeling and hyperblock loop unrolling), and finally applies dataflow optimizations modified to operate on predicated code [14].

Using the hyperblock as an abstraction enables compilers to reason about the value of applying predication using heuristics to evaluate a set of basic blocks in a region. Each basic block is assigned a priority for inclusion in a hyperblock, which allows the compiler to balance control flow and predication. VLIW heuristics focus on balancing dependence height, dependence width (resource utilization of each VLIW instruction and other resources), path frequency, and branch predictability [7, 18, 46, 47, 57]. The hyperblock formation goals for VLIW and EDGE architectures are related since both apply if-conversion to expose scheduling or placement regions by removing branches. Branch frequency is important to both architectures, since it is better to fill a hyperblock with instructions that execute under the same conditions and frequencies. VLIW heuristics have used path frequency as a key heuristic; it is unclear, however, whether EDGE architectures benefit from using path frequency.

To find improved hyperblock formation heuristics, prior work has applied machine learning techniques to search the space of heuristic functions. Meta Optimization [72] uses a genetic algorithm to evolve a custom heuristic using feedback over several generations of experiments. In that work a genetic programming system learns general-purpose heuristics for hyperblock formation that improve performance by 25% over a human-created heuristic. In Chapter 7 this dissertation uses a similar technique to learn heuristics for EDGE block formation, and to explore the effect of heuristics on performance at several composed core counts.

There are two important differences between VLIW and EDGE constraints [45, 66]. First, EDGE blocks also must conform to the architectural restrictions on block size, load/store ids, and register usage. Second, since VLIW machines issue instructions in a statically determined order, there is a high penalty for imbalanced dependence chains. An EDGE block can commit when all of its outputs are produced, and thus long dependence chains with a false predicate do not directly add to the execution schedule length, although they do occupy space in a block that may otherwise have been filled with useful instructions. An additional complication faced by an EDGE compiler, however, is that blocks must be size-constrained to allow dataflow encoding within a block. Nevertheless, both VLIW and EDGE compilers use heuristics to form large scheduling regions.

2.2 Phase-Ordering Problems

Iterative block formation with head duplication solves phase ordering problems between if-conversion, scalar/predicate optimization, and loop unrolling/peeling. Phase ordering problems in general are well-known in compiler optimizations, so we summarize the most closely related here.

Predication creates phase ordering problems, because code can be highly optimized in predicated form using standard techniques in the dataflow domain, but not all code is best left predicated. Reverse if-conversion allows code to be optimized in the predicated form and then reverted to the control flow domain as necessary [6, 75]. Iterative optimization has been used with reverse

if-conversion to produce good schedules for VLIW architectures [6]. After applying predication, optimizations, and scheduling, this algorithm finds the basic blocks that most constrain the schedule and remove them from a hyperblock, allowing a later scheduling attempt to produce better code.

August et al. discuss the interaction of hyperblock formation and scalar optimizations [7]. Their solution iterates on if-conversion, scalar optimizations, and VLIW scheduling. If this algorithm produces a poor schedule, it performs reverse if-conversion to remove basic blocks that constrain the schedule, allowing the algorithm to adjust hyperblock formation decisions after scheduling. Iterative block formation, by contrast, makes decisions incrementally in a single pass to ensure that the block conforms to the architectural constraints.

A related technique is software pipelining, which uses an iterative approach to select an appropriate unroll factor based on scheduling constraints [2, 3, 41, 58]. Rau's iterative modulo scheduling algorithm performs software pipelining for progressively larger values of the iteration interval until it resolves dependence constraints. Iterative block formation unrolls incrementally to fill blocks, but does not consider inter-iteration dependences, because dynamic issue in EDGE processors reduces the static scheduling problem of VLIW machines.

One of the classic phase ordering problems in compilation is between instruction scheduling and register allocation [10, 28]. If register allocation is performed before scheduling, re-use of register names may inhibit scheduling of parallel operations. If scheduling is performed first, it may increase the number

of simultaneous live values, leading to more spills to memory. Prior work has approached this problem using heuristics or algorithms that approach either scheduling or allocation with the other phases constraints in mind [28]. Iteration has been used in a limited form, using an instruction scheduling prepass to determine costs more accurately for register allocation [10]. While these techniques work well in this domain, this phase ordering problem is primarily about optimization: poor schedules or allocations may result in lower performance, but code will still execute correctly. With EDGE block formation, forming overly large blocks will result in failure to compile.

Loop unrolling presents a phase-ordering challenge to many optimizations. For example, scheduling, register allocation, and constant propagation could benefit from occurring after unrolling, since they take advantage of the redundancies found in the unrolled code. Prior research has found that most optimizations are best performed before loop unrolling [19]. Largely this is because heuristics for dealing with large loops break down when applied to the repetitive structure of an unrolled loop. Register allocation, for example, tends to commit too many registers to the body of an unrolled loop, which forces unneeded spills. In the context of EDGE processors, unrolling is problematic because it interacts with all other optimizations by changing the block size. Nethercote et al. resolve this tension by compiling in two phases, the first of which records the loop size after optimization [56]. Like pre-scheduling, this approach can only estimate the effect of unrolling. Iterative block formation solves this problem by merging blocks, performing scalar optimizations,

checking constraints, and undoing if necessary. While both algorithms may need to undo unrolling Nethercote et al. attempt to be conservative to limit undoing. Because iterative block formation eliminates the need for estimation, it is more aggressive, more accurate, and adds algorithmic flexibility.

2.3 Atomicity

Block-structured ISAs improve instruction fetch and issue bandwidth by aggregating instructions into atomic regions [29, 50]. The compiler for a block-structured ISA constructs enlarged blocks that contain a single path of control [49]. The compiler combines basic blocks to increase the processor's fetch bandwidth and scheduling abilities, using an incremental technique that combines basic blocks until meeting a threshold: block size and number of exits. The block enlargement phase is similar to incremental block merging as discussed in this dissertation, but the major phases differ because the block-structured ISA compiler does not include if-conversion, loop unrolling/peeling, or scalar optimizations in its iterative loop.

The execution model of a block-structured ISA differs from EDGE architectures in several key ways. In a block-structured ISA, if an early exit (fault) is taken from a block, the processor aborts and begins executing at the target of the early exit. Side exits from EDGE blocks do not result in an abort, merely predicated-out work; there is no notion of rollback at the ISA level, although the microarchitecture does implement rollback (in a sense) when blocks must be flushed due to misspeculation. A side exit in a block-structured ISA

causes the processor to squash the entire block and fetch a new block. EDGE architectures use predication extensively to form large blocks, which relies less on dynamic branch prediction, but requires a wider machine to tolerate the overhead of predication. EDGE architectures also use dataflow encodings and execution within blocks, which potentially allows for a more energy-efficient implementation.

Transactional memory (TM) provides hardware support for atomic regions [31]. While atomicity is a common feature, EDGE and TM serve very different purposes. Transactional memory provides atomicity constructs to user-level programmers, allowing them to define regions of code that must execute atomically with respect to other atomic regions. Atomicity in EDGE architectures is not visible to the application programmer; rather, the compiler constructs blocks from sequential code, which the architecture maintains atomically. The goals of EDGE and TM differ, as EDGE seeks to improve instruction throughput by eliminating branches and easing the extraction of instruction-level parallelism. TM provides a mechanism for more easily exploiting thread-level parallelism, which is often exposed to the application programmer as a replacement for fine-grained locking. Unlike EDGE blocks, atomic transaction regions are not size-constrained and may contain arbitrary control flow. We do note, however, that granularity is an important issue for TM systems, where large transactions can potentially degrade performance if frequently aborted [9].

Hardware atomicity has also been used to enable compiler optimiza-

tions [54, 55]. This work on optimizations enabled by hardware atomicity, rather than how atomic regions should be selected: the compiler selects regions to optimize heavily using standard profiling information. All paths outside of this region are replaced with abort instructions, so that optimizations may happen without regard for constraints imposed by these exiting branches. If a condition occurs that would exit the hot region, the processor rolls back state to the beginning of the region. Again, the differentiating factor is that flush/rollback is not transparent to the ISA. Speculative optimizations are more difficult to implement in an EDGE compiler, because EDGE ISAs do not provide an explicit abort mechanism.

Chapter 3

EDGE Architectures

This chapter provides background information about block-atomic architectures in general, and about the particular block-atomic ISA—Explicit Data Graph Execution (EDGE)—used in this thesis. Section 3.1 overviews block-atomic ISAs in general and why such architectures are desirable. Section 3.2 describes the features of block-atomic EDGE ISAs, which use dataflow execution to achieve high efficiency. Section 3.3 overviews compilation for EDGE architectures, and Sections 3.4 and 3.5 describe the TRIPS and TFlex microarchitectures, two early instances of EDGE ISAs.

3.1 Advantages of Block-atomic Architectures

Block-atomic instruction set architectures (ISAs) enable greater performance and efficiency than superscalar designs by grouping instructions into larger execution units. A processor implementing a block-atomic ISA fetches and commits such blocks atomically and in-order. Atomic commit of blocks is similar to atomic commit of instructions in conventional pipelined processors, which allows a processor to support precise interrupts [69]. In-order commit of blocks allows a familiar sequential programming model, in which the pri-

mary difference is that the smallest unit of work is a block rather than an instruction.

Various types of blocks may be supported by a block-atomic ISA and its compiler. In a RISC or CISC ISA, each instruction is trivially a block. VLIW architectures use fixed-size instruction words, which are blocks containing independent instructions. VLIW words are as wide as the issue width of the machine, which is typically 4–16 instructions. Larger blocks can be basic blocks, superblocks [34], hyperblocks [47], or other compilation units. The primary restriction on blocks is that they must not have internal control flow. Data dependences between instructions are allowed, however, in contrast to VLIW instruction words.

Processor performance benefits from the use of large atomic blocks [51]. Grouping instructions into blocks improves the front-end bandwidth of a processor by simultaneously fetching large groups of instructions and by alleviating bandwidth pressure on the branch predictor [29]. Without the benefit of large blocks, a superscalar processor larger than four-wide must often predict multiple branches per cycle to fill functional units. By contrast, a block-atomic ISA can combine several basic blocks, thus reducing the frequency of branches. With fewer branches to interrupt the instruction stream, the processor can fetch a larger number of instructions per cycle without the complexity of fetching many discontinuous basic blocks.

Another performance optimization opportunity provided by block-atomic ISAs is the ability to exploit locality of temporary values [53]. Values produced

by an instruction are likely to be consumed by nearby instructions, and most values have short live ranges. By communicating directly between instructions within a block, and writing to a global register file for inter-block communication, Explicit Data Graph Execution (EDGE) architectures facilitate distributed, out-of-order execution with low overhead. Section 3.2 describes EDGE ISAs in greater detail.

Block-atomic ISAs provide an efficiency advantage as well when compared to superscalar architectures. The performance of conventional superscalar processors is limited by power consumption due to the cost of discovering sufficient parallel work in each cycle. Using pipelined execution of many large blocks, a block-atomic ISA enables a larger instruction window within a given power budget, which reveals greater instruction-level parallelism.

This dissertation will focus on EDGE ISAs, which use predication to form large blocks of up to 128 instructions. EDGE architectures are an aggressive compiler target due to the block size. This section describes EDGE ISAs, in particular the TRIPS ISA, which is an early instantiation of an EDGE architecture. We discuss two microarchitectural implementations of EDGE: the TRIPS prototype, which is a physically realized processor, and the TFlex microarchitecture, which is an evolving research platform.

3.2 EDGE ISAs

EDGE architectures exploit concurrency efficiently by combining block-atomic and restricted, intra-block dataflow execution [11]. A program com-

piled for an EDGE architecture consists of a series of *structurally constrained* blocks, which have restrictions on the number and type of instructions that they can contain. The processor fetches and commits blocks in order, but may execute them in a pipelined fashion, similar to instructions in a superscalar processor. By committing blocks atomically and in order, an EDGE ISA can support precise exceptions at block boundaries. By providing a strict ordering of memory accesses, an EDGE ISA can support conventional sequential programming languages.

Within blocks, EDGE instructions communicate via direct dataflow. Rather than reading operands from (and writing results to) a shared register file, instructions encode the consumers of their results and send values to those targets. This direct target encoding reduces the need for global, centralized structures such as the register file and renaming logic, which can be a performance bottleneck and consume a large amount of energy. Due to the reduced need for centralized structures, an EDGE ISA enables a distributed microarchitectural implementation, which can utilize many functional units with less global communication.

The TRIPS ISA is an instance of an EDGE ISA, which is implemented by the TRIPS processor and the TFlex microarchitecture. The TRIPS ISA uses blocks of up to 128 instructions. To facilitate creation of large blocks, the ISA incorporates predication of nearly all instruction types. Using predication, the compiler converts control dependences into data dependences to merge multiple paths of control into a single block. By combining control paths, the

Parameter	Constraint
Instructions	128
Register reads	32
Register writes	32
Load/Store IDs	32
Outputs	A constant number of writes and stores Exactly one branch

Table 3.1: TRIPS ISA block constraints

compiler creates larger blocks without excessive code expansion and improves branch prediction accuracy by removing hard-to-predict branches. Because EDGE blocks have architectural constraints, they are more restricted than conventional hyperblocks, which are single-entry, multiple-exit regions of code in which control flow is replaced by predication [47].

The TRIPS ISA places four fixed architectural restrictions on the contents of blocks. These restrictions reduce hardware complexity at the expense of software flexibility. Table 3.1 summarizes these constraints. They are: (1) a maximum of 128 instructions per block; (2) a maximum of 32 loads and stores may issue per block; (3) a maximum of 8 register reads and 8 register writes to each of four register banks; and (4) a block must always produce a fixed number of outputs (stores, writes, and branches). The compiler for an EDGE ISA uses per-block SSA to ensure that all outputs are produced regardless of the predicate path taken through the block [66]. Section 3.3 describes the overall compiler structure.

While TRIPS and TFlex share this ISA, the microarchitectural implementation differs. The TRIPS hardware implementation demonstrates the

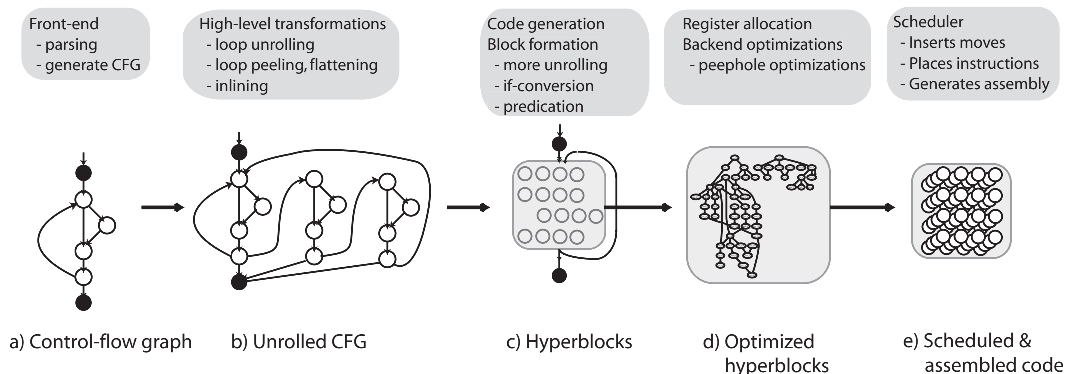


Figure 3.1: Phases of the TRIPS compiler.

feasibility of a tile-based, distributed microarchitecture. It uses heterogeneous tiles that implement execution units, registers and caches. TFlex is a homogeneous tiled architecture that fully distributes all resources.

3.3 Compiling for EDGE ISAs

The research compiler used for the TRIPS ISA, Scale, is a retargetable compiler with C and Fortran front-ends and back-end code generation support for several RISC instruction sets, such as Alpha, SPARC, and PowerPC, in addition to the TRIPS ISA back-end [48]. Scale is designed to be a modular, high performance compiler with many classical optimizations performed on a machine-independent intermediate representation, which makes it an excellent starting point for novel ISA research. Most of the TRIPS compiler research focuses on the back-end of the compiler, although heuristics are varied in the front-end optimizations to take advantage of the large register file and block temporaries in the TRIPS prototype.

Figure 3.1 shows the overall structure of the TRIPS compiler flow. The compiler front-end parses source code and transforms it to an abstract syntax tree (AST), at which level the compiler performs function inlining. The AST is lowered to a machine-independent control-flow graph (CFG), where classical optimizations including loop unrolling (of *for* loops), strength reduction, code motion, etc. are performed. The Scale front-end supports feedback-directed optimization using profiling information; the compiler can instrument programs to collect basic block, edge or path profiles, and read them during a second compilation pass.

After front-end optimizations are complete, the compiler performs a lowering pass, which transforms the machine-independent IR into a list of instructions in standard three-operand format that correspond to TRIPS assembly language instructions. The compiler arranges these instructions into an EDGE block flow graph (EFG). Because no predication is initially used, the EFG nodes correspond one-to-one with basic blocks in the control flow graph. The compiler progressively transforms this format into blocks that satisfy the TRIPS ISA constraints. In this stage the compiler performs block formation using if-conversion, additional loop unrolling, and loop peeling. Chapter 4 describes the block formation algorithms invented for this process. During block formation the compiler maintains a predicate flow graph (PFG) for each block, which enables the use of SSA to satisfy output constraints [66].

Satisfying the ISA output constraints described in Table 3.1 requires inserting store and write nullifications. This complex algorithm is described

in detail elsewhere [66, 68]. The complexity of null insertion has a significant impact on block formation. Because null insertion and fanout can change instruction count in non-linear and unpredictable ways, block formation cannot easily estimate the effects of merging a basic block without actually performing if-conversion, applying the nullification algorithms, and checking the result. This complexity motivates the iterative approach described in Chapter 4.

After forming blocks, the compiler allocates registers [59]. Because register allocation may insert spill code, the block constraints may be violated. To resolve this issue, the compiler performs a block splitting pass to reduce the size of invalid blocks containing spill code. Initially, block splitting was to be the primary method for ensuring the block constraints were met, given large blocks formed by the front end. However, it proved too error-prone and produced suboptimal code compared to an iterative method, because the block formation problem is overly constrained by the time block splitting occurs. Finally, the compiler emits code in TRIPS Intermediate Language (TIL), which is a RISC-like assembly language designed for easy readability by humans.

The instruction scheduler depicted in Figure 3.1 reads TIL and produces TRIPS Assembly Language (TASL) where all instructions are assigned dataflow identifiers that the processor will use to place instructions on ALUs at runtime. With some knowledge about the microarchitectural topology, the scheduler optimizes performance by reducing the critical path through the program, reducing latency between dependent instructions and maximizing parallelism [16, 52]. From TASL, the conventional steps of assembly and linking

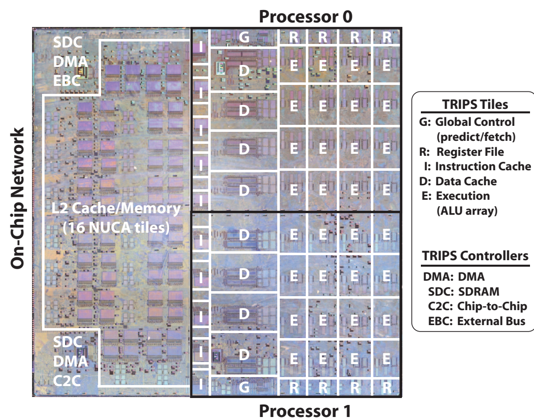


Figure 3.2: TRIPS processor die photo with outlined tiles.

into an executable are handled by modified versions of the GNU binutils [76].

The TRIPS compiler is a collaborative effort with many other researchers. The compiler front-end and machine-independent optimizations were maintained by James Burrill [48]. Aaron Smith implemented back-end code generation for TRIPS and many of the TRIPS-specific optimizations [66, 67]. Register allocation was implemented by Jon Gibson and Behnam Robatmili [59]. Katherine Coons implemented the scheduler used in this work [16]. Earlier versions of the scheduler were contributed by Sundeep Kushwaha and Ramadass Nagarajan [52]. The TRIPS assembler and linker were developed by Bill Yoder [76].

3.4 TRIPS Microarchitecture

The TRIPS processor is a wide-issue, large-window machine with a distributed microarchitecture that implements an EDGE ISA. The TRIPS

microarchitecture can issue 16 instructions per cycle from a window of up to 1024 instructions (eight blocks of up to 128 instructions each). Within each block, eight instructions are mapped to each functional unit to minimize contention and communication latencies [16, 52]. Using speculative next-block prediction, the microarchitecture supports eight blocks in flight, thus yielding the maximum instruction window size of 1024 instructions. The processor commits the oldest in-flight block after it produces all of its outputs: up to 32 stores, up to 32 register writes, and a single branch decision. Each block contains up to 32 register reads and writes in addition to the 128 regular instructions.

The TRIPS microarchitecture is a heterogeneous design, using five different types of tiles to implement the processor core. The global control tile (G) is responsible for managing branch prediction, fetch, flush, and commit protocols; the instruction tiles (I) contain the instruction cache; the data tiles (D) contain the data cache; the register tiles (R) contain the register file; and the execution tiles (E) contain the issue queues and ALUs. Figure 3.2 shows a die photo of the processor with these tiles outlined to show their locations. The tiles are connected via micronetworks that communicate control messages, instructions, and data.

3.5 TFlex Microarchitecture

Dynamic multicore processors, which adapt their parallel resources to the workload at hand, have been shown to provide the best performance trade-

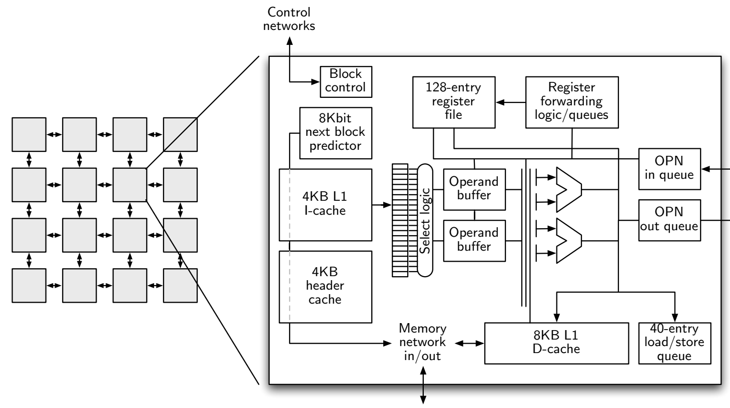


Figure 3.3: Microarchitecture of one TFlex Core

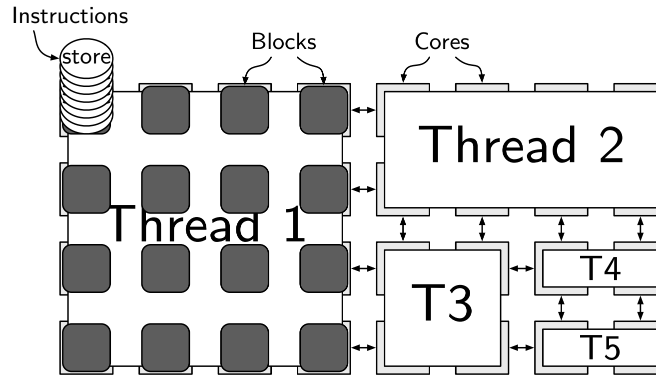


Figure 3.4: Mapping blocks to composable TFlex cores

off given a mix of sequential and parallel work [32]. Block-atomic execution enables an EDGE processor to execute programs on few or many cores by executing blocks independently on cooperating cores, using a shared register file and memory only for coarse grained, inter-block communication [37, 60]. While composability can be achieved using a RISC or CISC ISA as in Core Fusion, limitations such as fine-grained register communication using a centralized register renaming unit and relatively frequent control decisions physically limit composability to four cores [35].

The goal of the TFlex microarchitecture is to create a composable, lightweight processor (CLP) that can be reconfigured to handle a variety of workloads [37]. A TFlex processor consists of many simple cores that the system can compose to accelerate the execution of a single thread, as depicted in Figure 3.4. When multiple cores execute a single thread, TFlex dynamically maps each block to a single core, which allows multiple blocks execute in parallel on different cores [60]. When running a workload with many threads, each core can independently execute a thread. When running few threads with greater ILP needs, many cores can be combined to form more powerful logical processors. With composability, TFlex can adapt to the characteristics of the programs running on it.

To achieve composability, the individual cores must be capable of acting independently. Each tile must have its own cache and register file. Unlike the TRIPS processor where these resources are placed along the edges of the execution grid, each TFlex tile contains registers and cache as shown in Fig-

ure 3.3. When combining cores, these resources become parts of a larger logical register file and cache. When executing a program on a large logical processor, each atomic block may be mapped to some subset of the cores.

TFlex is implemented in simulation only, unlike TRIPS, which has been realized in hardware. A simulator implementation allows variety with architectural policies and microarchitectural implementations. Chapter 7 will discuss how the EDGE block formation problem can vary given differing microarchitectural implementations. This dissertation shows that performance is not entirely portable across microarchitectural configurations, as the best compiler policies for small and large core counts differ substantially.

Chapter 4

Iterative Block Formation

EDGE architectures rely on the compiler to form high-quality blocks for good performance. These compilers typically perform if-conversion, loop unrolling, and scalar optimizations in a fixed order. This approach limits the compiler's ability to exploit or correct interactions among these phases. EDGE architectures exacerbate this problem by imposing *structural* constraints on blocks, such as instruction count and instruction composition.

This chapter describes *iterative block formation*, which iteratively applies if-conversion, peeling, unrolling, and scalar optimizations until converging on blocks that are as close as possible to the structural constraints. To perform peeling and unrolling, iterative block formation generalizes *tail duplication*, which removes side entrances to acyclic traces, to remove back edges into cyclic traces using *head duplication*. Simulation results using an EDGE architecture show that iterative block formation improves code quality over discrete-phase approaches with heuristics for VLIW and EDGE. This algorithm offers a solution to block phase ordering problems and can be configured to implement a wide range of policies.

4.1 Introduction

To form blocks, a compiler has several transformations at its disposal. If-conversion enlarges blocks by converting control dependences to data dependences [4]. While performing if-conversion, a compiler may apply tail duplication to remove side entrances to a region. Loop unrolling removes branches by creating multiple copies of a loop body, and either recognizing that the loop test is unnecessary or replacing it with predication. Loop peeling pulls initial iterations out of the loop body, to allow scheduling of instructions from those iterations with surrounding code. By applying these transformations, the compiler can remove control flow instructions and expose parallelism within the block.

Because these transformations all change the composition of the blocks, phase-ordering problems can arise. For example, a compiler may select different unroll factors for a loop depending on whether the loop body has been if-converted. Conversely, if-conversion may be more or less effective on an already unrolled loop body. Beyond these phase ordering problems, scalar optimizations present an additional challenge. By performing if-conversion or loop unrolling a compiler can create opportunities for optimization; those optimizations, however, may change the desired if-conversion strategy or unrolling factor.

Constrained block-atomic architectures like EDGE present an additional challenge for compilers by placing size and content restrictions on blocks. These restrictions increase the performance sensitivity of optimizations be-

cause poorly sized blocks execute less efficiently on a block-atomic microarchitecture. Furthermore, strict limits transform a performance optimization into a correctness criterion. While overly aggressive if-conversion or unrolling may simply result in slow code on a conventional architecture, it may cause the compiler to generate invalid code for a block-atomic architecture. While it may be possible to correct such errors at the end of code generation—via block splitting—such an approach typically leads to suboptimal code and is best avoided when possible [45, 66]. Thus it is important for the compiler to consider the effect of optimizations on block size at all phases of code generation.

This chapter presents two techniques for dealing with the above phase ordering problems. The first contribution is an iterative, incremental approach to block formation, which allows the compiler to continuously monitor the size of a block as it is formed. Prior work on hyperblock formation forms blocks by examining all basic blocks in a region and ranking them for inclusion in a hyperblock. Incremental block formation considers only the successor basic blocks of a block as candidates for inclusion. At each step the block can be re-optimized to take advantage of newly exposed opportunities, which improves the packing of the block. While improving the fullness of blocks is only one aspect of optimization, iterative hyperblock formation allows the compiler to use heuristics to select the best successor blocks for inclusion. This approach both allows the compiler to satisfy the ISA block constraints, and resolves the phase-ordering tension between block formation and scalar optimizations.

To solve the phase-ordering problem between if-conversion and unrolling/peeling, this dissertation introduces *head duplication*. Head duplication is a transformation analogous to tail duplication. Where tail duplication eliminates merge points created by forward branches by duplicating code, head duplication eliminates merge points created by backward branches, which occur at loop headers. This transformation effectively duplicates the loop body, thus implementing unrolling and peeling. By casting these transformations as tail duplication, the compiler can apply unrolling and peeling as needed during the course of block formation.

Iterative block formation with head duplication preserves the compiler's ability to use heuristics in selecting blocks for inclusion. Prior approaches to hyperblock formation use the features of each basic block or path through a region [46, 47, 72]. Iterative block formation is more limited in scope: it considers only the immediate successors of a block as candidates for inclusion, although larger regions of the graph may be considered as part of the heuristics. Iterative block formation encapsulates this heuristic in a single priority function at the heart of the algorithm. This chapter evaluates the effect of iterative block formation on several basic policies. Chapter 7 presents a more detailed exploration of the policy space for iterative block formation.

We evaluate iterative block formation using simulation of the TRIPS EDGE architecture described in Section 3.4. These experiments show that this approach improves TRIPS microbenchmark cycle counts by 2 to 11% on average when compared to classical phase orderings. These results also establish

a strong correlation between block count reduction and performance improvement. A functional simulator shows that iterative block formation reduces block counts of the SPEC2000 benchmarks, indicating potential performance improvement on real applications.

By resolving phase ordering problems among block formation, loop unrolling, loop peeling, and scalar optimizations, iterative block formation with head duplication enables effective compilation for block-atomic architectures. Combining these techniques yields an elegant, single-pass algorithm that enables high-quality code generation and improves the robustness of the compiler. Our implementation of iterative block formation has significantly reduced complexity in the TRIPS compiler back end. The self-checking nature of the algorithm reduces the possibility of subtle dependences between phases, one of the most common sources of bugs in the compiler.

4.2 Phase Ordering Challenges

Structural architectural constraints on block formation exacerbate a phase ordering problem between block formation, loop unrolling, loop peeling, and scalar optimization. Block formation creates scalar optimization opportunities that are difficult to express in the control-flow domain. Two such examples are instruction merging, which combines instructions from distinct control-flow paths, and implicit predication, where the compiler predicates only the head instruction in a dependence chain, thus implicitly predicating the successors [67]. Since these optimizations typically eliminate instructions

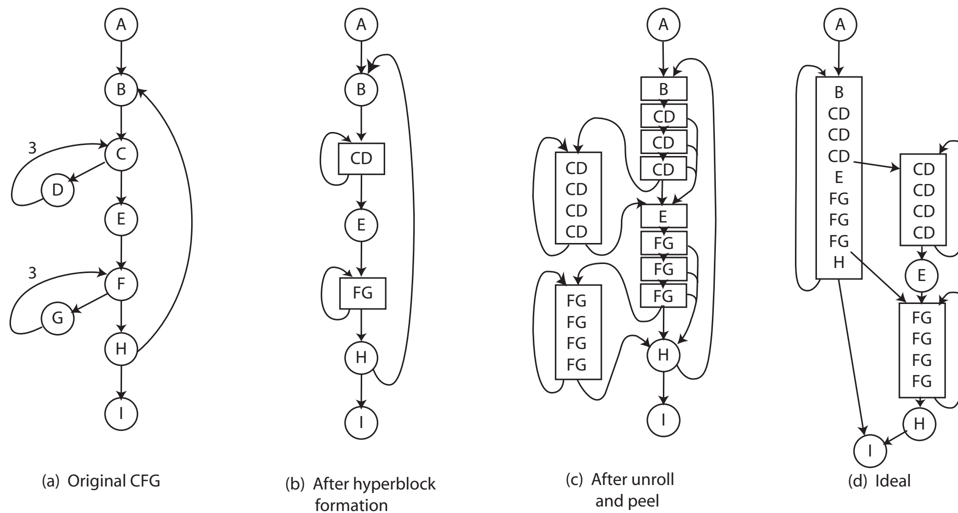


Figure 4.1: Block formation example.

or fanout, their application may enable the compiler to include more basic blocks in the block, improving code density.

Loop unrolling and peeling present a similar challenge. If the compiler performs these transformations before block formation, it may if-convert multiple iterations and combine them into large blocks. Without performing block formation on the body of the loop, however, the compiler cannot determine an appropriate unroll factor. Figure 4.1a shows a CFG extracted from the SPEC benchmark ammp, which consists of an outer loop with two inner loops where all the loops must perform their exit test on each iteration. Such loops are termed *while loops*, but do not assume the C "while" loop construct. While-loop unrolling requires block formation to predicate each iteration, unlike *for-loop* unrolling, which can remove intermediate tests. This example assumes that profiling indicates that each loop typically iterates three times.

Figure 4.1b shows block formation in which the compiler first if-converts the bodies of the inner while loops. Figure 4.1c shows the code when the compiler uses the profile to peel three iterations, and then unrolls the loop four times to fill the block for the less frequent case. Ideally, the compiler would now repeat block formation to produce the code in Figure 4.1d.

Under different conditions, the ideal phase ordering may include another pass of peeling: if the loops execute either three or four times in the common case and the compiler cannot fit four peeled iterations in a single block until after scalar optimizations, the ideal compiler would peel again. Thus, each static phase ordering of block formation, peeling, unrolling and scalar optimizations may miss opportunities to create better blocks.

4.3 Iterative Block Formation

Iterative block formation incrementally merges basic blocks and iteratively applies optimizations to ensure that each block is well constructed and tightly packed with useful instructions. The algorithm incorporates loop peeling and unrolling by generalizing tail duplication to remove back edges.

4.3.1 Head and Tail Duplication

Compilers use tail duplication to expand a block by duplicating code below a merge point and eliminating side entrances. On a VLIW architecture, the compiler copies the merge point, and changes the pertinent branch to target the copy. Other than the branch, the compiler does not need to modify

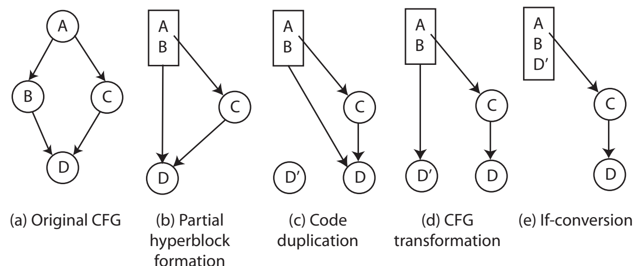


Figure 4.2: Classical tail duplication.

either the copied or the original code.

An EDGE compiler, however, must predicate the merge point for two reasons. First, a branch does not immediately terminate an EDGE block; instead, the architecture requires that a block produce all of its outputs (register writes and stores, in addition to the branch) before committing. Second, unpredicated instructions within the block execute when they receive operands. Therefore, the instructions in an unpredicated merge point would send results to the outputs of the block, even if the processor takes the side exit. Essentially, duplicating the merge point makes it control-dependent on the exit test, and correct dataflow execution requires the compiler to convert this control dependence to a data dependence.

Furthermore, the EDGE compiler must transform the resultant block to meet the structural constraints (i.e., the block must produce a fixed number of outputs) [66]. Thus the side exit must produce the same number of outputs as the main path, potentially adding size overhead to the block, and runtime overhead if the side exit is taken.

Figure 4.2a shows an example of tail duplication in which the compiler

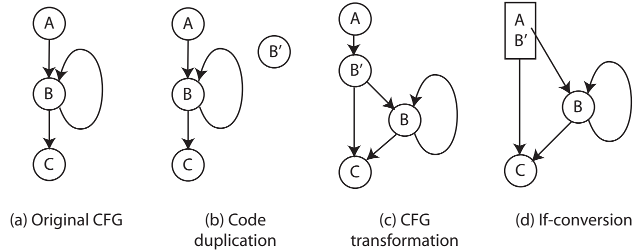


Figure 4.3: Head duplication implements peeling.

chooses to combine A, B, and D. The compiler first if-converts B and merges it with A (Figure 4.2b). The compiler then applies tail duplication to D to eliminate the side entrance. It copies D to create D' (Figure 4.2c). Next, it modifies the CFG, redirecting $AB \rightarrow D$ to D' (Figure 4.2d). The compiler then if-converts D', predicating the instructions on the original branch condition for $A \rightarrow B$, and merges the result into the block (Figure 4.2e).

Compilers typically perform block formation on acyclic regions within a CFG. Head duplication generalizes block formation to include cyclic regions, effectively implementing peeling and unrolling. Encountering a block that is the target of both an edge from the current block and a loop back edge is similar to encountering a block with a side entrance; thus the compiler can apply the same tail duplication process. Tail duplication creates outgoing edges from the duplicated block to the successors of the original. If the original block was a loop, the new edge will either be a loop entrance (peeling) or a back edge (unrolling).

Consider the CFG in Figure 4.3a. Since B is a loop header, tail duplication is insufficient for combining A and B. Head duplication peels a copy

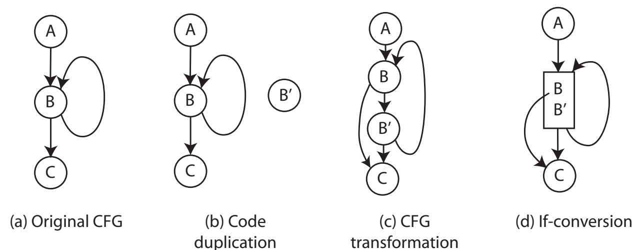


Figure 4.4: Head duplication implements unrolling.

of B to merge with A. The compiler copies B to make B' (Figure 4.3b), then redirects edge $A \rightarrow B$ to B', adds $B' \rightarrow B$, and for all $B \rightarrow X$, inserts $B' \rightarrow X$ (Figure 4.3c). Finally, the compiler if-converts and merges B' and A (Figure 4.3d).

Figure 4.4 shows how head duplication also implements loop unrolling. Consider creating a block starting with basic block B. Since the back edge points to B, the compiler can unroll and still satisfy the single entry constraint. Head duplication creates a copy of the loop body B' (Figure 4.4b). The compiler then replaces $B \rightarrow B$ with $B' \rightarrow B$, inserts $B \rightarrow B'$, and $B' \rightarrow C$ (Figure 4.4c). The last step if-converts and merges B' into B (Figure 4.4d). If the compiler were to apply additional unrolling directly to this CFG, it could only unroll by powers of two. To remove this limitation, the unrolling procedure saves the original loop body and appends one additional iteration at a time.

4.3.2 Incremental Block Formation

Iterative block formation integrates unrolling, peeling, tail duplication, if-conversion and scalar optimizations to form blocks incrementally. Figure 4.5

```
procedure ExpandBlock(HB : block)
 1: candidates := Successors(HB)
 2: while candidates is not empty do
 3:    S := SelectBest(candidates)
 4:    candidates := candidates − {S}
 5:    if not LegalMerge(HB, S) then
 6:       continue
 7:    else if MergeBlocks(HB, S) == Success then
 8:       candidates := candidates ⋃ Successors(S)
 9:    end if
10: end while

procedure MergeBlocks(HB, S : block)
 1: HBcopy := Copy(HB)
 2: Scopy := Copy(S)
 3: HBS := Combine(HBcopy, Scopy)
 4: Optimize(HBS)
 5: if not LegalBlock(HBS) then
 6:    return Failure
 7: else if NumPredecessors(S) == 1 then
 8:    Replace(HB, HBS) // no code duplication
 9:    Remove(S)
10: else if HB → S is a back edge and HB == S then
11:    UnrollLoop(HB, S)
12: else if S is a loop header and HB → S is not a back edge then
13:    PeelLoop(HB, S)
14: else
15:    TailDuplicate(HB, S)
16: end if
17: return Success
```

Figure 4.5: Iterative block formation algorithm.

shows the pseudocode for the algorithm. The procedure `ExpandBlock` starts with a block HB and selects a successor S to merge according to a heuristic. The compiler then calls `MergeBlocks`, which attempts to merge S into HB, duplicating code if necessary. If the merge is successful, the compiler adds the successors of S to the set of candidates.

`MergeBlocks` first copies HB and S to scratch space and attempts to if-convert and merge S into HBS. The compiler optimizes the resulting block, and then checks to determine whether the block violates the structural constraints. If so, the merge fails and the compiler considers other successors. By testing the merge in scratch space before transforming the CFG, the implementation avoids a more complicated undo step.

If the merge is successful, the compiler must transform the CFG appropriately. If $HB \rightarrow S$ is the only entrance to S, the compiler can simply remove S from the CFG and replace HB with HBS (lines 7–9 in `MergeBlocks`). Otherwise, it must perform code duplication. Lines 10–15 of `MergeBlocks` show the cases where the compiler performs unrolling, peeling, and tail duplication. The compiler uses head duplication to implement unrolling and peeling and tail duplication for other cases. The `Optimize` step attempts to eliminate instructions in the merged block. The compiler currently applies dominator-based global value numbering and predicate optimizations that reduce the number of instructions that use each predicate [67].

4.4 Policy

Iterative block formation constructs blocks that obey architectural constraints while increasing code density. To achieve high performance, however, the algorithm must apply heuristics that select the most profitable basic blocks to include in each block. This block selection policy can balance several characteristics of high-performance blocks.

The two simplest heuristics, breadth-first and depth-first, each emphasize one of two opposing goals. By merging basic blocks in breadth-first order, the compiler guarantees the inclusion of some useless instructions, but attempts to decrease the branch misprediction frequency and limit tail duplication. The depth-first policy risks a higher misprediction rate and performs more tail duplication, but seeks to include a greater number of useful instructions.

Branch predictability: Removing conditional branches is important for EDGE architectures because of their large instruction windows. In the TRIPS prototype, each processor has a 1024-instruction window consisting of eight blocks, seven of which are control-speculative. A branch misprediction and subsequent pipeline flush prevents effective utilization of this window. The compiler can improve predictability during block formation by eliminating unpredictable conditional branches. One heuristic that eliminates conditional branches is to end blocks at merge points so that each has a single exit, however this policy may result in under-full blocks.

Limiting tail duplication: On a dataflow architecture, tail duplication requires additional predication below the side exit, including predication of the merge point. This requirement introduces data dependences on the outcome of the test in the duplicated code, while in the original program the instructions were control independent. These dependences may degrade performance, since the resultant code cannot execute speculatively, but must wait on the resolution of a possibly time-consuming test. This effect is especially problematic when the duplicated merge point contains a loop induction variable update that is on the critical path through an otherwise parallel loop.

Loop peeling and unrolling: Because iterative block formation folds loop transformations into the block formation algorithm, the compiler can apply block selection policies to loops as well. To perform peeling accurately, the compiler can use loop trip count histograms to augment an edge frequency profile. A loop peeling policy can then evaluate the benefit of unrolling additional loop iterations versus including post-loop code by using a threshold function to pick an appropriate peeling factor.

Local and global heuristics: Local heuristics consider only the characteristics of the current block when choosing among the candidate successors. Because of the architectural constraints on TRIPS blocks, a local approach works well for TRIPS block formation. By incrementally merging basic blocks, the block gradually converges on the upper bound of the constraints. Because the compiler adds blocks individually to satisfy structural constraints, the algorithm focuses on selecting one of a block's immediate successors for inclusion.

Using lookahead can increase the power of local heuristics. For example, a heuristic that improves branch predictability favors blocks with a single exit. Such a heuristic might first determine if a block has one exit, and then use lookahead to estimate if the compiler can include enough additional basic blocks to reach the next merge point, thus constructing a larger, single-exit block.

Although local heuristics seem most suitable for incremental block formation, the algorithm can use global information to inform block selection by performing a pre-pass analysis. To implement path-based VLIW heuristics [46, 47] using iterative block formation, the compiler analyzes the CFG to create a prioritized list of basic blocks, and then merges blocks in priority order, when possible.

Dependence height: The best-known block selection heuristic for VLIW architectures analyzes all paths through a region to determine which basic blocks to include [46]. Because a VLIW hyperblock is statically scheduled, the dependence height of the longest path determines the execution time of the block, even if that path is not taken at runtime. Paths are therefore prioritized to favor those that execute frequently, consume few resources, and have short dependence heights. These heuristics attempt to avoid over-constraining the static schedule or over-saturating the processor's resources, while still including the most useful paths and removing unpredictable branches.

For EDGE ISAs, minimizing dependence height is less important. EDGE instructions issue dynamically when their operands arrive, and the architec-

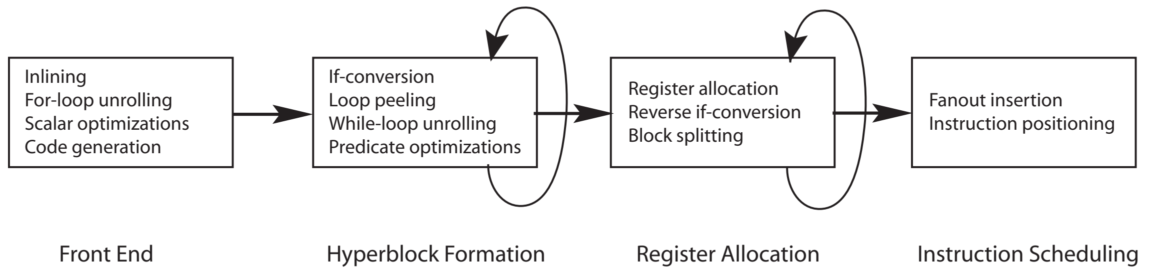


Figure 4.6: Compiler flow with iterative block formation.

ture can commit a block as soon as it produces its outputs. Therefore, if a short path through an EDGE block completes before a longer one, the architecture detects block termination and does not wait for the longer path to finish. Although speculative instructions on an untaken path may contend for resources with instructions on a taken path, this contention reflects constraints on issue width rather than on schedule height.

4.5 TRIPS compiler

We implement iterative block formation in Scale [48, 66], a retargetable compiler with a back end for TRIPS. Figure 4.6 shows the overall compiler flow. The compiler front end operates on a language and machine-independent control-flow graph representation. Scale performs inlining and for-loop unrolling first, followed by classical scalar optimizations. The compiler then lowers the program representation to a RISC-like form. Using this representation, the compiler performs block formation, followed by register allocation, fanout insertion to replicate values for multiple consumers, and finally scheduling.

Since register allocation and fanout insertion add additional instruc-

tions and occur after block formation, the compiler must estimate final block sizes while forming blocks. Although block formation tries to construct blocks of the appropriate size and load/store count, the register allocator may insert spill code that violates the block constraints. If a block has spills that cause it to violate a constraint, the compiler performs reverse if-conversion on the block, and repeats register allocation. Scale rarely needs to split blocks in this manner, both because TRIPS has a large number of architectural registers and because the compiler attempts to avoid inserting spill code in nearly full blocks. Once register allocation completes and all blocks are valid, the scheduler inserts fanout instructions, assigns locations to all instructions in their respective blocks, and translates them to TRIPS assembly language.

4.6 Experimental Results

We evaluate iterative block formation using a TRIPS cycle-level timing simulator, which has been verified to be within 4% of the cycle counts generated by the TRIPS prototype hardware design on a set of microbenchmarks. This simulator models all aspects of the microarchitecture, including global control, data path pipelines, and communication delays within the processor. Because detailed simulation is prohibitively slow (approximately 1000 instructions per second), we restrict the evaluation to microbenchmarks derived by extracting loops and procedures from SPEC2000, and with signal-processing kernels from the GMTI radar suite, a 10x10 matrix multiply, sieve (a prime number generator), and Dhrystone.

	BB	UPIO	IUPO	(IUP)O	(IUPO)
	cycles	%	%	%	%
ammp_1	1544356	18.2	68.6	68.6	65.9
ammp_2	1021042	13.6	59.0	59.0	60.2
art_1	83309	4.5	12.1	4.9	6.0
art_2	128499	-7.5	1.0	-5.3	3.4
art_3	638918	76.6	77.0	74.4	76.1
bzip2_1	478746	22.1	22.1	22.1	22.1
bzip2_2	334299	32.6	32.0	32.0	32.0
bzip2_3	556743	34.6	34.6	34.6	34.5
dct8x8	51988	-0.6	-0.6	-6.3	-6.3
dhry	234345	13.5	22.5	23.1	23.5
doppler_GMTI	85229	21.8	13.2	14.3	16.6
equake_1	114324	0.7	0.7	2.3	12.4
fft2_GMTI	130496	25.9	21.1	25.2	27.9
fft4_GMTI	98538	4.7	7.3	6.6	4.9
forward_GMTI	180900	0.5	2.2	2.0	3.8
gzip_1	29377	22.2	22.2	20.8	48.4
gzip_2	98414	54.8	46.4	48.3	54.9
matrix_1	71814	-25.2	37.9	38.4	42.3
parser_1	395076	46.5	46.5	46.5	46.5
sieve	443064	-13.1	20.9	23.7	22.6
transpose_GMTI	185803	4.2	4.2	1.6	1.5
twolf_1	527166	38.9	39.7	38.9	38.6
twolf_3	588011	0.5	0.5	0.5	0.5
vadd	105407	-2.1	7.9	5.4	9.4
Average		**16.2**	**25.0**	**24.2**	**27.0**

Table 4.1: Percent improvement in cycle counts of EDGE blocks over basic blocks (BB) with various orderings of Unrolling (U), Peeling (P) Incremental If-conversion (I), and Scalar Optimizations (O). Parentheses indicate merged phases.

4.6.1 Comparison to Static Phase Ordering

Table 4.1 compares the performance of block formation with discrete phases of unrolling and if-conversion to the single-phase, but iterative, incremental block formation. Column 2 shows the baseline cycle count of each benchmark using basic blocks as TRIPS blocks. Basic blocks are a good baseline because they are defined by the front end of the compiler, and do not depend on the block formation algorithm or heuristics. The remaining columns apply if-conversion (I), unrolling/peeling (UP), and scalar optimizations (O) in various orders. Phases grouped together in parentheses indicate that the transformations are applied incrementally, using head duplication to implement unrolling and peeling, and iterative optimization to improve code density. All results use a greedy breadth-first policy and use incremental if-conversion to avoid violating the block constraints.

The UPIO and IUPO columns show discrete phase orderings of structural transformations followed by scalar optimizations. UPIO performs loop unrolling and peeling before incremental if-conversion and tail duplication. This approach improves performance by an average of 16% over basic blocks. The second phase ordering (IUPO in Column 4) performs if-conversion before loop unrolling and peeling. It improves performance an additional 8.8% on average compared to UPIO since the unroller has more accurate block counts and size estimates for loops with control flow after if-conversion than before.

The (IUP)O column shows that iterating peeling and unrolling appears to offer no benefit over the distinct phases in IUPO on these benchmarks,

despite adding the capability to generate blocks like Figure 4.1d. Most of these benchmarks consist simply of for loops with high trip counts. Because Scale applies for-loop unrolling in the front end, the only benefit of head duplication is to merge the test for the execution of the post-conditioning loop with the body of the unrolled loop. Sometimes merging this test helps performance slightly (e.g., fft2_GMTI and sieve), and sometimes it hurts slightly (e.g., art_1 and art_2). The best candidates for head duplication are ammp_1 and ammp_2, which contain while loops with low trip counts. However, the compiler's block size estimates are not yet sufficiently accurate to combine peeled iterations of these loops with surrounding code (see Section 4.5).

Integrating scalar optimizations into block formation (the (IUPO) column) attains an additional 2% performance improvement because the compiler can pack blocks more tightly and perform more if-conversion and unrolling. The most significant improvement, gzip_1, occurs because the compiler uses if-conversion and scalar optimizations to fit the entire body of the innermost loop in one block, dramatically reducing the total number of blocks executed.

The m/t/u/p statistics in Table 4.2 show how often the compiler applies if-conversion (m), tail duplication (t), unrolling (u), and peeling (p). For example, the performance improvement of ammp_1 in all columns following UPIO occurs because the compiler unrolls and peels several additional iterations of the critical loops. Using (IUPO) on ammp_1 enables peeling of an additional loop iteration, but this transformation happens to create a less-predictable branch pattern and increases the number of mispredicted branches by 50%.

	UPIO	IUPO	(IUP)O	(IUPO)
	m/t/u/p	m/t/u/p	m/t/u/p	m/t/u/p
ammp_1	18/11/3/0	18/11/11/7	18/11/11/7	18/11/13/8
ammp_2	39/11/3/0	39/11/8/3	39/11/8/3	40/2/10/2
art_1	11/1/3/5	12/0/3/4	13/0/3/2	13/0/4/2
art_2	4/1/2/5	6/0/2/2	8/1/2/1	7/1/4/3
art_3	23/1/2/3	24/0/3/2	25/1/3/1	25/0/3/2
bzip2_1	7/0/0/0	7/0/0/0	7/0/0/0	7/0/0/0
bzip2_2	9/1/3/5	10/0/3/5	10/0/3/5	10/1/3/5
bzip2_3	10/0/3/0	10/0/3/1	10/0/3/1	10/0/3/1
dct8x8	4/0/0/0	4/0/0/0	4/0/0/0	4/0/0/0
dhry	63/2/4/5	64/3/6/9	66/4/6/8	64/4/10/12
doppler_GMTI	7/2/10/9	8/1/11/9	8/1/11/9	8/1/12/11
equake_1	6/0/0/0	6/0/0/0	6/0/0/0	7/0/0/0
fft2_GMTI	11/3/1/3	12/2/1/2	13/3/1/1	13/4/1/1
fft4_GMTI	7/1/0/1	8/0/0/0	8/0/0/0	8/1/0/0
forward_GMTI	10/2/1/3	10/3/1/4	11/4/1/3	11/3/3/7
gzip_1	9/3/0/0	9/3/0/0	9/3/0/0	12/2/0/0
gzip_2	5/2/3/6	6/2/3/4	7/2/3/2	6/2/6/7
matrix_1	10/0/0/1	10/0/3/5	10/1/3/5	10/1/4/6
parser_1	12/0/0/0	12/0/0/0	12/0/0/0	12/0/0/0
sieve	6/3/7/8	7/1/7/4	7/2/7/4	7/2/9/5
transpose_GMTI	6/0/0/0	6/0/0/0	6/0/0/0	7/1/0/1
twolf_1	10/5/1/2	11/4/1/1	12/4/1/0	15/1/1/0
twolf_3	12/0/0/0	12/0/0/0	12/0/0/0	12/0/0/0
vadd	5/1/0/1	5/1/1/5	6/2/1/5	6/2/2/5

Table 4.2: Static count of blocks merged/tail duplicated blocks/unrolled iterations/peeled iterations (m/t/u/p), with various orderings of Unrolling (U), Peeling (P) Incremental If-conversion (I), and Scalar Optimizations (O). Parentheses indicate merged phases.

On the microbenchmarks, the best heuristic for forming TRIPS blocks improves performance compared to basic blocks by 27% on average. Iterative block formation outperforms classical optimization phase orderings by an average of between 2 and 11%.

4.6.2 VLIW and EDGE Heuristics

The above results show that iterative block formation offers potential performance benefits given the right heuristics. Iterative block formation is flexible enough to implement a variety of policies as discussed in Section 4.4. We compare performance using three heuristics: the VLIW heuristic proposed by Mahlke et al. [46, 47], a depth-first heuristic that selects the most frequent path, and a breadth-first heuristic that removes conditional branches.

Table 4.3 shows the performance of the VLIW and EDGE heuristics on TRIPS. Columns 3 and 4 show the VLIW block selection heuristic applied without and with iterative optimization, respectively. Without iterative optimization the VLIW heuristic achieves a 6.1% average speedup over basic blocks, compared to 10.7% with iterative optimization, demonstrating that iterative block formation improves the performance of this heuristic. Column 5 shows the depth-first heuristic, which achieves a small 5.7% speedup. Breadth-first merging shows the greatest improvement, at 27%.

Several of the largest performance differences among these results occur because tail duplication incurs additional predication on a dataflow architecture. The most extreme example of this effect is bzip2_3, where breadth-first

	BB	VLIW	Iterative VLIW	DF	BF
ammp_1	1544356	64.8	61.7	62.8	65.9
ammp_2	1021042	3.8	4.1	1.7	60.2
art_1	83309	3.3	2.6	7.0	6.0
art_2	128499	0.3	7.2	6.9	3.4
art_3	638918	45.0	45.0	29.3	76.1
bzip2_1	478746	-25.4	-25.4	-37.4	22.1
bzip2_2	334299	-59.0	0.9	-40.6	32.0
bzip2_3	556743	-67.9	-68.1	-91.7	34.5
dct8x8	51988	-0.6	18.3	-7.5	-6.3
dhry	234345	17.2	17.2	19.6	23.5
doppler_GMTI	85229	13.1	16.6	19.7	16.6
equake_1	114324	0.7	13.6	12.4	12.4
fft2_GMTI	130496	28.0	28.0	28.7	27.9
fft4_GMTI	98538	5.6	6.6	10.2	4.9
forward_GMTI	180900	4.7	-1.0	5.4	3.8
gzip_1	29377	49.3	46.1	12.1	48.4
gzip_2	98414	30.1	29.2	32.0	54.9
matrix_1	71814	37.9	39.2	40.0	42.3
parser_1	395076	25.1	27.0	45.1	46.5
sieve	443064	11.2	16.6	1.5	22.6
transpose_GMTI	185803	4.2	-0.9	2.5	1.5
twolf_1	527166	-60.8	-42.6	-41.9	38.6
twolf_3	588011	7.4	3.7	4.8	0.5
vadd	105407	7.9	11.2	15.1	9.4
Average		**6.1**	**10.7**	**5.7**	**27.0**

Table 4.3: Percent improvement in cycle count over basic blocks (BB) using VLIW heuristics, VLIW with iterative optimization, depth-first (DF) and breadth-first (BF) EDGE heuristics.

merging achieves a 34.5% speedup while depth-first and VLIW degrade performance by 68.1% and 91.7%, respectively. While breadth-first merges all paths through the main loop, the depth-first and VLIW heuristics exclude an infrequently-taken block, and therefore must tail duplicate the final block in the loop, which contains the induction variable increment. The induction variable is then data-dependent on the earlier test, instead of being indepen-

dent. This dependence results in a slowdown even over basic blocks, where the increment can be executed speculatively.

Improved branch prediction accuracy is another important effect. In parser_1, the VLIW heuristic excludes several rarely taken paths with relatively large dependence heights. Because these branches are rarely taken, they cause mispredictions when they occur, resulting in an 11-fold increase in the misprediction rate (0.4% using breadth-first versus 4.5% with VLIW), which reduces the effective size of the processor's issue window. The depth-first heuristic does not suffer branch mispredictions because it is able to include all paths through the loop, since there is ample space in the block after merging the most frequent path.

4.6.3 Estimated Performance with Block Counts

The cycle-level simulator is too slow to simulate the full SPEC benchmarks. Since successful transformations reduce the number of blocks executed, thus increasing the issue window utilization and decreasing block overhead, block counts and program cycle counts should correlate. We demonstrate this correlation and present block count results for SPEC2000.

The best static phase ordering achieves a 2.1x improvement in number of blocks executed over basic blocks on the microbenchmarks, while iterative block formation achieves a 2.3x improvement. To first order, the relationship between the number of blocks executed and the cycle count is roughly: $cycles_{total} = cycles_{base} + blocks \times overhead$, where $cycles_{total}$ is the total number

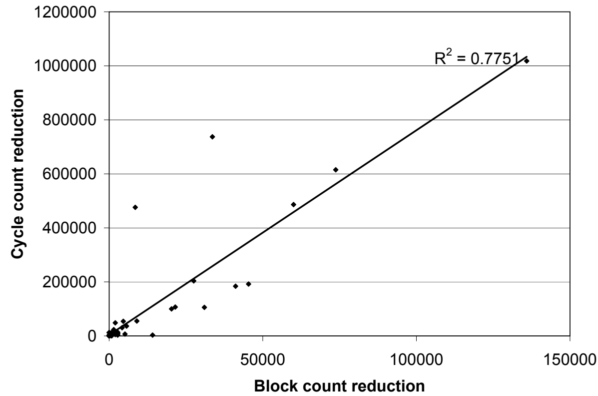


Figure 4.7: Cycle count reductions versus block count reductions.

of cycles the program takes to execute, $cycles_{base}$ is the number of cycles to perform the computation (ignoring block boundaries), *blocks* is the number of blocks, and *overhead* is the fixed architectural overhead associated with mapping a block.

This equation is an oversimplification, because it does not account for increased parallelism exposed by block merging, nor interference from including speculative, useless instructions in a block. To evaluate the accuracy of this estimate, Figure 4.7 plots the change in cycle counts against the change in block counts (as compared to basic blocks) for all the data presented in Table 4.1. The relationship between block count reduction and cycle count reduction is roughly linear ($r^2 = 0.78$ using a linear regression test), with a few outliers due to ammp_1, in which reducing the block count by unrolling while loops dramatically improves performance. This result suggests that reduction in block count is a good but imperfect metric of performance improvement.

The correlation between block count reduction and cycle count reduc-

	BB (M)	Phased UPIO	Phased IUPO	Iterative (IUP)O	Iterative (IUPO)
bzip2	248.8	40.8	45.9	46.5	50.4
crafty	16.7	42.7	46.5	49.3	55.3
gap	20.6	11.7	11.8	11.8	11.9
gzip	86.5	59.4	60.0	60.0	62.3
mcf	28.1	61.2	69.8	70.1	63.8
parser	87.2	51.7	57.0	57.1	57.1
twolf	15.6	57.4	59.6	60.6	62.1
vortex	41.2	63.7	63.8	63.8	62.9
vpr	2.9	60.1	61.2	61.5	62.8
ammp	12.5	65.4	72.9	73.9	73.9
applu	1.6	42.4	42.4	43.9	45.8
apsi	5.9	47.3	47.3	47.5	48.9
art	331.8	65.0	65.0	70.1	72.6
equake	100.1	57.3	57.8	58.1	59.0
mesa	870.3	51.8	51.8	51.8	51.8
mgrid	591.6	4.3	4.3	4.5	5.3
sixtrack	479.3	54.1	54.2	54.2	53.5
swim	2.8	30.1	30.1	30.1	31.7
wupwise	1469.7	47.5	46.9	49.2	52.9
Average		**48.1**	**49.9**	**50.7**	**51.8**

Table 4.4: Percent improvement in block counts of SPEC benchmarks over basic blocks (BB) with various combinations and orderings of Unrolling (U), Peeling (P) If-conversion (I), and Scalar Optimizations (O). Parentheses indicate merged phases.

tion justifies the use of block counts (gathered using a fast, functional simulator) to estimate the performance effect of these algorithms on the SPEC2000 benchmarks. Table 4.4 shows the block count results for 19 of 21 FORTRAN and C benchmarks (at the time of this writing, the toolchain is not stable for 176.gcc and 253.perlbmk), where the baseline (BB) measures millions of basic blocks executed. The remaining columns report percent improvement, using the same configurations and format as Table 4.1. These results use the MinneSPEC small reduced dataset [38], since the ref datasets require too much

	BB (B)	Phased		Iterative	
		UPIO	IUPO	(IUP)O	(IUPO)
bzip2	181.4	-16.8	-11.0	-21.5	5.5
crafty	1098.7	50.9	51.2	56.7	63.0
gap	1739.1	60.9	62.5	57.8	66.6
gcc	81.7	19.5	18.9	-0.4	24.6
gzip	154.9	5.8	6.8	-50.0	1.7
mcf	469.4	-31.1	-28.2	-34.3	6.7
parser	1196.9	9.7	14.4	13.4	25.8
perlbmk	239.0	43.0	43.2	43.0	46.9
twolf	1699.9	51.0	51.7	52.9	56.2
vortex	1947.6	20.3	20.6	20.6	23.6
vpr	571.3	66.7	65.7	65.0	70.8
ammp	803.1	-12.1	6.7	3.7	24.7
applu	594.7	6.9	11.9	5.3	6.9
apsi	1071.3	14.4	17.2	17.7	18.4
art	165.9	44.8	46.4	46.4	59.2
equake	241.9	10.3	9.6	12.4	14.7
mesa	734.9	45.3	46.6	42.6	45.7
mgrid	276.5	14.8	14.9	13.6	15.1
sixtrack	0.0	0.0	0.0	0.0	0.0
swim	272.5	1.3	2.0	0.3	1.2
wupwise	660.5	34.8	38.8	44.3	43.0
Geo. Mean		**36.5**	**40.4**	**34.1**	**54.8**

Table 4.5: Percent improvement in cycle counts of SPEC benchmarks over basic blocks (BB) with various combinations and orderings of Unrolling (U), Peeling (P) If-conversion (I), and Scalar Optimizations (O). Parentheses indicate merged phases.

simulation time, even using a less detailed simulator. The block count results show the same trends as the cycle-accurate results on the microbenchmarks, although head duplication is in general more effective and scalar optimizations are slightly less effective than in the microbenchmark results.

Using the TRIPS hardware, we gather cycle-accurate performance results for the same benchmark set. This data is shown in Table 4.5. Similar general trends occur in performance; the worst static phase ordering (unrolling,

block formation, optimization) achieves a 36% performance improvement over basic blocks, while the fully iterative algorithm achieves a 54.8% performance improvement. Head duplication by itself does not appear to improve performance, despite reducing block count, as applying only head duplication without iterative optimization achieves a 34.1% performance improvement.

4.7 Summary

This chapter described iterative block formation with head duplication. We have shown that this method is more effective at filling blocks and achieves higher performance than other, static approaches. The principle of combining phases to achieve greater compiler flexibility allows the system to make better optimization decisions than are possible without this approach.

Iterative block formation is also more robust than other approaches. Prior to implementing iterative block formation, the compiler relied on block-splitting to form legal blocks. This approach creates many poorly understood dependences between phases and is very difficult to debug from an engineering standpoint. Iterative block formation, by contrast, continuously considers the architectural block constraints, and is therefore much more robust to the effects of optimizations.

We find a noticeable correlation between block size and performance in this work, which shows that as the block count of an executing program decreases, the cycle count also decreases linearly. This finding implies that making small blocks larger has proportionally more effect on performance than

making large blocks somewhat larger. As Chapter 5 will show, the presence of small blocks is a significant problem for block-atomic architectures. Despite this relationship, iterative block formation is able to achieve relatively good speedups over static phase orders, even on a microbenchmark workload that already has relatively large blocks.

The results in this chapter indicate that iterative block formation is more effective than a static phase ordering. The following chapters of this dissertation will explore the limits of iterative block formation. Chapter 5 describes the fundamental limits of block formation due to control flow constraints, and Chapter 7 describes a principled approach to discovering heuristics for use with iterative block formation.

Chapter 5

Compiler Evaluation

The opportunity to evaluate industrial-strength software on an experimental hardware platform is a rarity in modern compiler research. The engineering effort required to build a modern microprocessor is so great that few university teams undertake it. Furthermore, the effort and insight required to write a high-performance compiler for a new architecture often requires several years after first silicon. The TRIPS hardware team has had the opportunity to build a silicon prototype of a novel processor, which is fully functional and validated with no known bugs [27, 62]. The TRIPS software team rose to the occasion with a high-quality research compiler that can handle real-world benchmarks such as the SPEC CPU2000 suite, with 21/21 benchmarks passing validation.

In this chapter we evaluate the performance of the TRIPS system on compiled code, specifically the SPEC CPU2000 benchmarks. We compare to the best commercial processor available at the time of the TRIPS bring up, the Intel Core 2 microarchitecture, to provide the most aggressive comparison point possible. Our goal with this comparison is to determine whether the real world performance of a TRIPS system justifies a move to a radically different

ISA. We find that on the SPEC benchmarks, TRIPS achieves performance parity on SPECFP, but falls short on SPECINT.

Examining the detailed performance counter statistics indicates a few problems with the TRIPS system and compiler. Several integer benchmarks suffer from high instruction cache miss rates and high branch misprediction rates. These issues indicate problems with the block-atomic execution model, which result from a mismatch between the capabilities of the hardware and the capabilities of the software. Because TRIPS uses fixed-size, 128-instruction blocks, the compiler must be able to effectively pack blocks with useful instructions as well as limit code duplication to avoid excessive i-cache misses. Evidence suggests that the system does not successfully meet this goal.

To better understand the software component of the evaluation, we present a dissection of the compiler's block formation capabilities. We identify all possible cases where the compiler is unable to merge blocks, and combine this with dynamic trace information to see what constructs result in small blocks. The distribution of block sizes as well as the fundamental nature of the causes suggest that the compiler would need heroic efforts to fill a fixed-size block. This chapter suggests some of these techniques, but engineering effort prevents a full evaluation.

This evaluation leads to the conclusion that processors using a block-atomic architecture such as EDGE must support variable-size blocks. Without such support, the penalty for under-full blocks is simply too great when executing realistic software workloads. In Chapter 6 we present one such microarchi-

tecture and evaluate its performance. Results demonstrate that variable-size blocks effectively eliminate instruction cache bottlenecks and provide further opportunities for optimization.

5.1 Comparative Performance Evaluation

This section presents the comparison between the TRIPS system and several Intel platforms. It outlines the methodology used to achieve the fairest possible comparison and summarizes the results of the performance evaluation.

5.1.1 Methodology

We compare performance on several hardware platforms using the built-in performance counters. In addition to the TRIPS processor and the Core 2, we supply performance measurements on the Pentium III and Pentium 4. Table 5.1 summarizes the microarchitectural properties of the various platforms.

To account for the wide variance in process technologies across these platforms, we use cycle counts rather than wall clock time as a performance metric. Cycle count is an imperfect metric for comparison, since commercial microarchitectures are tuned to a particular technology and cycle time. In particular, the NetBurst microarchitecture of the Pentium 4 emphasizes cycle time at the expense of IPC. However, we expect that the TRIPS microarchitecture, which has a partitioned design with few global wires, could achieve a cycle time close to that of the Core 2, if implemented in a modern process with a custom layout.

System	Issue Width	Proc Speed (MHz)	Mem Speed (MHz)	Proc/Mem Ratio	L1 Cap. (D/I) (KB)	L2 Cap. (MB)	Mem Cap. (GB)
TRIPS	16	366	200	1.83	32 / 80	1	2
Core 2	4	1600	800	2.00	32 / 32	2	2
Pentium 4	4	3600	533	6.75	16 / 150	2	2
Pentium III	3	450	100	4.50	16 / 16	0.5	0.256

Table 5.1: Reference platforms.

A potential pitfall with this comparison is that the low clock frequency of the TRIPS CPU relative to its DRAM (366 vs. 200 MHz) could make memory accesses unrealistically inexpensive when compared to the reference platforms. We normalize this effect by underclocking the Core 2 such that the ratio of its core clock frequency to DRAM frequency (1600 vs. 800 MHz) is similar to that of the TRIPS system. We do not perform such normalization for the Pentium III or Pentium 4.

We compile all benchmarks on a platform with a common set of optimization flags, rather than customizing flags on a per-benchmark basis. This setting better represents realistic practice, and reduces the possibility of per-benchmark tuning, which represents available engineering effort rather than fundamentals. The TRIPS compiler was set to its highest optimization level, `-Omax`, and enabled tail duplication during block formation, which achieves a small average speedup across the benchmark suite. In this configuration Scale performs its full set of scalar optimizations with heuristics set to use registers liberally, iterative block formation with integrated unrolling and peeling (as described in Chapter 4), and inlining with 100% code increase. Code for the Intel platforms was compiled with `gcc -O3`, an open source compiler with

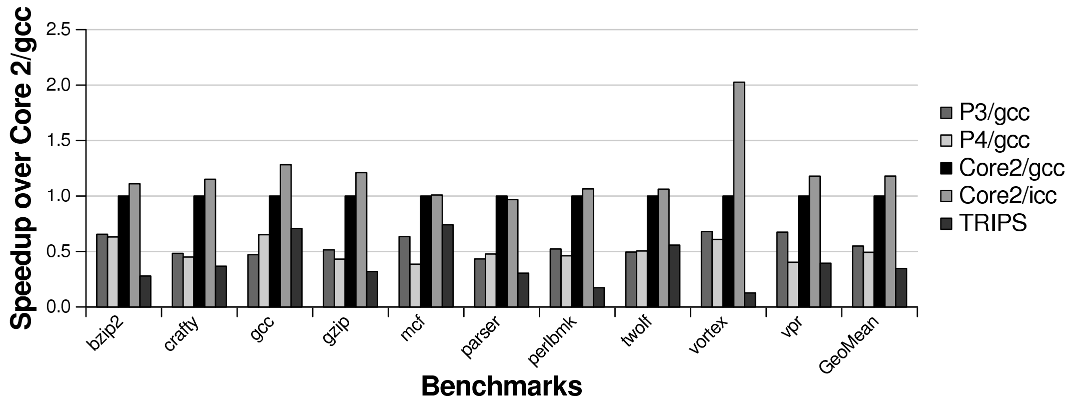


Figure 5.1: Speedup on SPECINT relative to Core 2/gcc.

reasonable optimized performance. To push the Core 2 to its extreme we also experimented with Intel's `icc` compiler at its highest optimization level.

We compare performance on the SPEC CPU2000 C and Fortran77 benchmarks. We omit C++ and Fortran90 as they are unsupported by the TRIPS compiler. Additionally, we omit gap and sixtrack for stability reasons—while these benchmarks run to completion in less-optimized configurations, but do not produce correct output at higher optimization levels. While prior work uses smaller benchmarks and hand-optimized code, this dissertation focuses on the SPEC benchmarks. While hand-optimized code is useful for pushing the limits of the microarchitecture, this dissertation is primarily concerned with the abilities of the compiler. As such, the more realistic SPEC benchmarks are a better target.

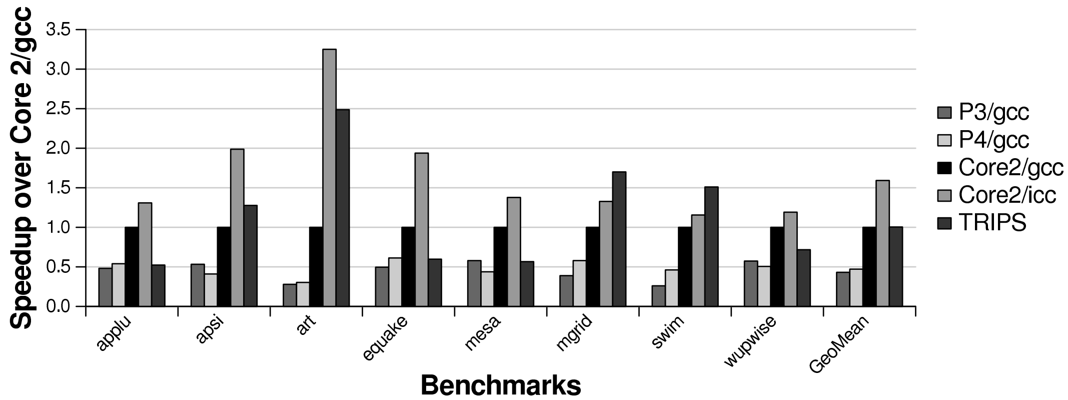


Figure 5.2: Speedup on SPECFP relative to Core 2/gcc.

5.1.2 Performance Results

Figure 5.1 compares SPECINT performance and Figure 5.2 compares SPECFP performance across the tested platforms. The graphs show performance as speedup over the Core 2 platform where the benchmarks are compiled with `gcc`. A few trends are notable. On SPECFP TRIPS achieves average equivalent performance to Core 2/`gcc`, although with `icc` the Core 2 outperforms TRIPS by a factor of 1.6. TRIPS fares noticeable worse on SPECINT, however, achieving only 40% of the performance of Core 2/`gcc`.

Compared to the Pentium III and Pentium 4, however, the TRIPS system fares much better: performance improves by 2.3x and 2.1x respectively on SPECFP, and achieves 63% and 70% of the Intel platform's performance on SPECINT. This result shows how quickly industry reacted as power became a limiting factor; per-clock performance dramatically improved from the NetBurst to Core 2 microarchitecture as predicted performance improvements due

to deep pipelining and frequency scaling proved too limited and too power-hungry [1, 33].

Several individual benchmarks results are notable. In SPECFP, TRIPS achieves a 2.5x speedup over Core 2/`gcc` on art. This speedup is due to a compiler transformation in an important loop which reduces the tree height of an arithmetic expression and allows the processor to exploit its wide execution bandwidth. This benefit is not limited to TRIPS alone, however, as Core 2/`icc` achieves slightly greater performance of 3.3x. On SPECINT, perlbmk and vortex stand out for their remarkably poor performance on TRIPS, 5.8x and 7.9x slower respectively than Core 2/`gcc`. Because perlbmk is an interpreter for Perl, it is characterized by extremely small blocks due to frequent function calls. This behavior is the worst case scenario for TRIPS, because the compiler is unable to expose significant parallelism to the hardware, and the overhead of block-atomic distributed execution overwhelms the work done in each block. The performance of vortex is somewhat suspect, as vortex heavily taxes the memory allocator. The TRIPS toolchain uses a simple `malloc` implementation suitable for embedded systems, while x86 platforms use a highly tuned allocator built into glibc based on dlmalloc [42]. The TRIPS allocator implements the free list as a long singly-linked list, which frequently requires long walks to find suitable allocation sites, where the x86 allocator uses a more efficient data structure.

To better understand the performance results, we collected detailed statistics from the hardware performance counters available in TRIPS and the

	Per 1000 useful instructions						
	Core 2 cond. br. misses	TRIPS cond. br. misses	TRIPS call/ret misses	Core 2 I-cache misses	TRIPS I-cache misses	TRIPS load flushes	Average useful insts in flight
bzip2	1.3	1.6	0.0	0.0	0.0	0.09	342.5
crafty	4.5	3.0	0.5	1.7	17.2	0.35	151.8
gcc	7.4	7.0	1.8	3.1	18.5	0.52	73.0
gzip	4.8	4.3	0.0	0.0	0.0	0.04	206.1
mcf	14.0	6.3	0.0	0.0	0.0	0.13	373.6
parser	2.0	3.2	0.1	0.0	0.6	0.04	—
perlbmk	2.5	0.4	8.3	0.0	13.0	0.19	106.9
twolf	8.5	4.8	0.1	0.0	8.2	0.36	275.2
vpr	0.5	1.4	0.5	0.0	3.2	0.40	221.8
ammp	0.2	1.5	0.1	0.0	1.0	0.05	—
applu	0.0	0.7	0.0	0.0	0.0	0.01	496.6
apsi	0.0	2.4	0.0	0.0	0.0	0.11	249.7
art	0.4	0.0	0.0	0.0	0.0	0.01	692.2
equake	0.2	0.6	0.0	0.0	0.9	0.08	337.9
mesa	1.4	1.6	0.0	0.0	3.5	0.04	199.4
mgrid	0.0	0.1	0.0	0.0	0.0	0.00	519.8
swim	0.0	1.0	0.0	0.0	0.0	0.00	416.1
wupwise	0.0	0.7	0.5	0.0	0.8	0.04	496.9

Table 5.2: Performance counter statistics for SPEC.

Core 2. Table 5.2 shows these results. We collect branch predictor statistics (conditional and call/return for TRIPS, conditional only for Core 2) and instruction cache miss rates. We correlate the TRIPS statistics with the effective instruction window size as an additional performance indicator.

The performance counter results indicate significant problems with the capacity of the TRIPS instruction cache. Three benchmarks (crafty, gcc, and perlbmk) show more than 12 instruction cache misses per 1,000 instructions. These miss rates severely limit the performance of the applications compared to the Core 2, which suffers a much lower rate of misses. Branch prediction is not as severe a problem, although perlbmk shows a relatively high rate of call/return misses, due to an undersized call target buffer. When these factors are considered, the TRIPS processor is able to use a much smaller fraction of its instruction window on SPECINT than on SPECFP; because a large window is key to performance of EDGE architectures, SPECINT suffers.

5.2 Block Size Efficiency

To understand the role of software in the previous section's performance results, we examine the compiler's ability to form full blocks in the SPEC benchmarks. Given the poor instruction cache performance on SPECINT—despite a larger L1 i-cache than the Core 2—it seems that the compiler's ability to fill blocks effectively is more limited on full benchmarks than in the microbenchmarks presented in Chapter 4.

This section analyzes the compiler's ability to form full, fixed-size blocks

and identifies structural characteristics of source code that limit the compiler's success. To compile effectively for a block-atomic ISA, the compiler must construct blocks that contain sufficient useful instructions at runtime. In prior work, the first EDGE microarchitectures, TRIPS and TFlex, demonstrate good performance on compute-intensive code with a fixed-size block, but had less success with more complex control flow [27, 37]. The TRIPS prototype evaluation revealed frequent misses to the L1 instruction cache and underutilization of the instruction window with fixed-size, 128-instruction blocks, thus motivating a more flexible approach.

We find that the compiler often cannot fill blocks effectively even when targeting a less ambitious, smaller block size. To form large dataflow blocks in control-intensive code the compiler must extensively predicate, thus converting control dependences into data dependences. As we show in Section 7.5, predication, which sacrifices control speculation and may increase dependence height, is not as effective on control-intensive codes as speculating down a single predicted branch path. Using the more flexible microarchitecture proposed in Chapter 6, the compiler reduces block size when needed to avoid excessive predication, while still making effective use of cache and issue resources.

5.2.1 Methodology

We can measure the average fullness of blocks experimentally by compiling and simulating the SPEC CPU2000 benchmarks. We use the SPEC benchmarks because they are fairly large programs that reflect realistic work-

loads and because they contain several control-intensive programs that pose a significant challenge for block formation. Block size depends on the compiler's optimizations of course, so for these numbers we use the most aggressive optimizations in terms of forming the largest blocks possible.

To compute an average block size we weight each block's size by its execution frequency. This weighted average is more meaningful than a static, unweighted average, because it reflects the fact that certain blocks are executed more frequently than others, and that some are never executed. The weighted measurement is sensitive to the data set used, which in these experiments is the MinneSPEC small reduced data set [38]. This reduced dataset is necessary because these numbers can only be acquired through simulation—the TRIPS prototype hardware lacks the logic necessary to distinguish fetched `NOP`s from real instructions. The data include all fetched instructions, and do not account for the fact that some of these instructions are predicated out or that others are overhead, such as store nulls and fanout.

We measure the efficiency of blocks for several possible upper bounds: 32, 64, and 128 instructions. By measuring efficiency across this range and comparing this data to energy efficiency and performance metrics, we hope to provide future designers with insight into how to select a desired block size.

5.2.2 Block Fullness

Figure 5.3 shows the dynamically-weighted average instructions per block with maximum fixed block sizes of 32, 64, and 128 instructions. *Useful*

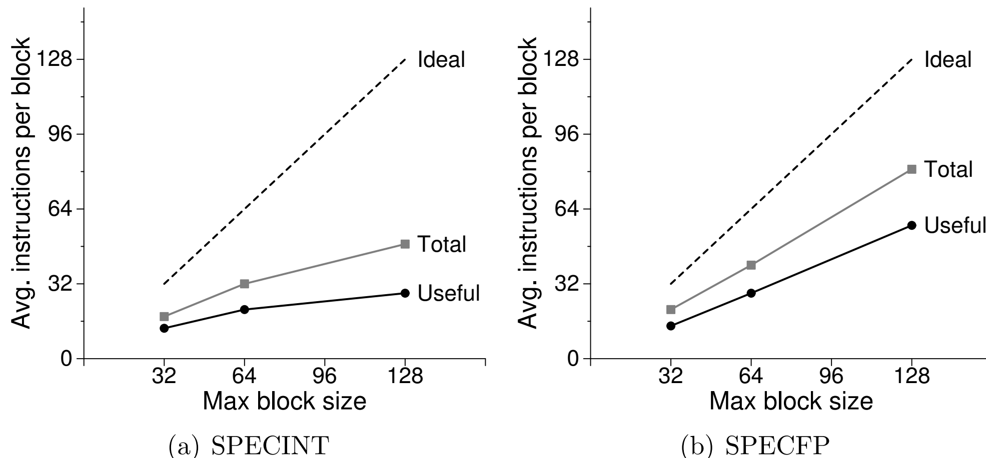


Figure 5.3: Dynamic average total and useful instructions per block with various maximum block sizes.

instructions include only those instructions that contribute to the block's outputs; instructions with non-matching predicates are excluded. The compiler's ability to fill blocks is limited, and reducing the granularity helps to only a limited extent. The SPECINT benchmarks' blocks are particularly difficult to fill as the maximum block size increases. Although efficiency improves as the upper bound decreases, the number of instructions per block decreases drastically as well, which suggests that for a fixed window in terms of blocks (such as on the TRIPS prototype) a larger block size will lead to a larger window, at the cost of additional energy spent fetching `NOP`s. Table 5.3 shows the efficiency as a percentage of maximum block size. As results in Chapter 6 show, implementing variable-size blocks improves efficiency without reducing the maximum block size.

As will be shown in the following section, efficiency does not improve

	Maximum Block Size		
Benchmark	**32**	**64**	**128**
bzip2	0.62	0.56	0.44
gzip	0.59	0.53	0.50
mcf	0.67	0.67	0.71
parser	0.52	0.51	0.40
vpr	0.53	0.48	0.43
applu	0.49	0.44	0.35
art	0.75	0.65	0.79
equake	0.48	0.39	0.29
mesa	0.50	0.45	0.39
mgrid	0.41	0.50	0.48
swim	0.50	0.41	0.30
wupwise	0.68	0.66	0.69
average	0.56	0.52	0.48

Table 5.3: Per-benchmark efficiency of blocks for fixed maximum sizes of 32, 64, and 128 instructions.

more dramatically for a variety of reasons. Many blocks that are small and medium-sized are unchanged, while larger blocks are divided up and run into limitations. Thus it is not surprising that the real number of instructions drops significantly as the maximum upper bound decreases.

5.2.3 Control Flow Limitations

We instrumented the compiler to produce a reason for why each pair of blocks were not merged. Thus for every branch between basic blocks in the control flow graph the compiler outputs a cause. The precise causes are described later in this section. We produce an execution trace consisting of all blocks, and create a histogram of cause frequency by correlating pairs of blocks

in the trace with pairs from the compiler's cause file. By relating the reasons for block cuts to the size of the blocks and their frequency of execution, we can discover the principle causes of under-full blocks.

There are some pairs of blocks that can occur that the compiler will not be able to analyze. For example, indirect function calls may have unknown targets, and function returns may go to any number of locations unknown to the function itself. These situations are detected using the TRIPS toolchain naming conventions for blocks, and thus do not present a problem for analysis. We did not compile libraries with instrumentation, so they are opaque to this analysis, so we separate all blocks occurring within libraries. We do, however separate calls into libraries from calls within user code, as they cannot be inlined without binary analysis, and would prevent dynamic linking.

Figure 5.4 provides the histogram of block sizes and cut reasons for SPEC CPU2000 benchmarks compiled with a fixed maximum block size of 128 instructions. This histogram shows the dynamic frequency of blocks in various size classes, and each bar of the histogram is divided according to which of the following factors prevented the compiler from merging a particular block with the next in execution order. We have grouped the blocks into size classes based on the first block of each pair (described above), which explains the unintuitive presence of "Full Blocks" with few instructions—the following block is quite large. This histogram shows a generally "U-shaped" distribution, where small blocks and large blocks are most common. To explain this distribution, we enumerate the most common reasons why the compiler stops enlarging a block.

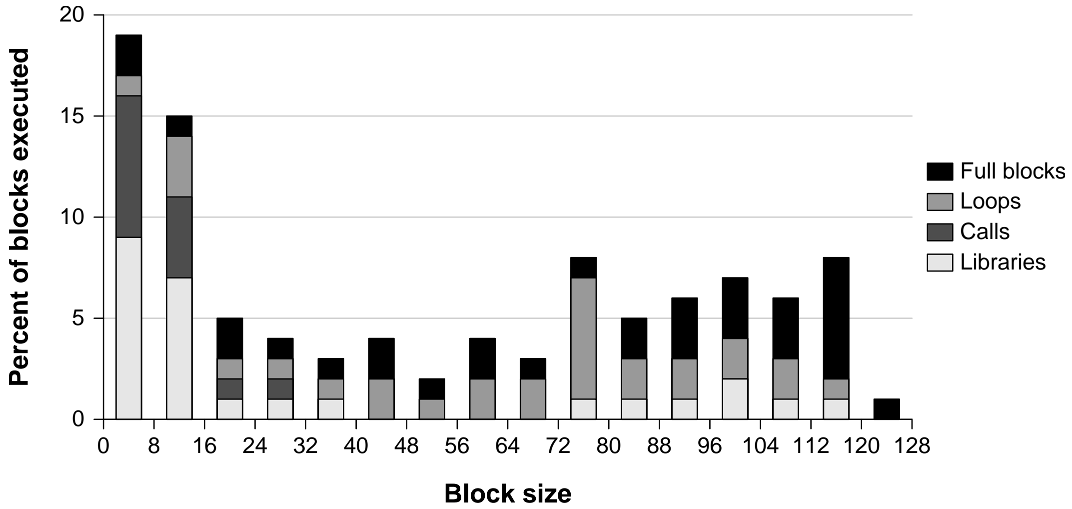


Figure 5.4: Distribution of block sizes in SPEC CPU2000 benchmarks, weighted by execution frequency. The categories indicate the reason for the compiler's inability to merge that block with the next in the execution trace.

- **Full Blocks:** In the simplest case, block formation stops when merging another basic block would violate size constraints. This constraint is most frequently encountered with large blocks, but occurs occasionally with small blocks when the *following* block is large. A limitation of our current compiler is that it merges basic blocks entirely or not at all, but we have explored splitting basic blocks and found that this limitation is a minor one. Commonly, we find that the limiting factor in merging an additional basic block is not the raw instruction count of the block itself, but the overhead instructions due to fanout (of data values or, more likely, predicates) or to write/store nullification. Nullification can be particularly problematic since it requires the addition of code along every mutually exclusive predicate path through the block.

- **Loops:** We classify separately the case where a loop cannot be additionally unrolled or peeled due to block size constraints. This condition is a subset of the more general block size limitations, but we separate it due to the particular importance of loops.

  Loops present several problems to the compiler for both peeling and unrolling, and depending on whether the loop body can be condensed into a single block. We enumerate the four possible cases. (1) A loop can be condensed into a single block, but cannot be additionally unrolled due to block size limitations. (2) A loop can be condensed, but cannot be peeled into surrounding code due to size limitations. (3) A loop cannot be condensed into a single block, and thus is not considered for back-end unrolling. (4) A loop cannot be condensed, and thus will not be considered for peeling into surrounding code.

  To overcome some of these limitations, loops could be unrolled past a single block to better align with the block size. For instance, an 80-instruction loop could—in theory—be unrolled three times, and carved into two 120-instruction blocks.

  Unrolling loops to fill multiple blocks has several problems, however. First, if the loop has any non-trivial control flow, it is not clear how to divide it neatly into multiple blocks. Second, filling blocks is not a heuristic that guarantees success; creating one less-full block may be better than three completely full blocks, because one block reduces instruction cache pressure and may align better with the dependence structure of the loop.

Third, it is simply rare that such a fortuitous division of iterations into blocks occurs, due to control flow, dataflow overheads, and the vagaries of actual loop sizes.

Due to these difficulties in implementing correct multi-block unrolling and achieving reasonable performance with it, we exclude it from our evaluation. This experience motivates the use of variable-size blocks because it is easier and less error-prone to rely on the hardware to deal with irregularly-sized loops by dynamically unrolling them to fill the available window.

- **Calls:** A function call must end a block, because blocks execute atomically and cannot be "re-entered" upon return from a function call. Function calls break up control flow in a way that the compiler cannot easily overcome during block formation. Aggressive inlining helps by inlining function calls, and the results in Figure 5.4 include up to 100% code growth from inlining, which is an aggressive value. Indirect function calls are particularly problematic because they cannot be inlined without speculation.

  We note two cases where our compiler is more conservative than necessary with calls. The first is that any basic block containing a call will not be predicated. The second is that function epilogues will never be tail-duplicated. These scenarios cause a relatively large number of small blocks, because calls and epilogues are themselves small, and frequent

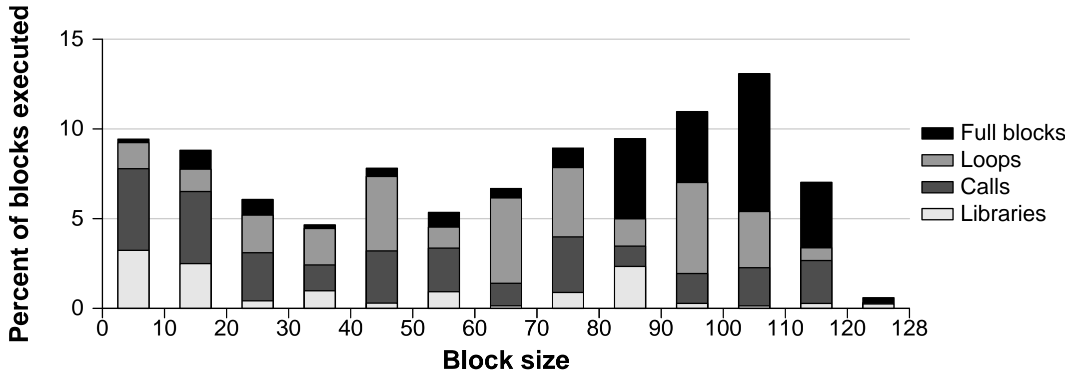


Figure 5.5: Distribution of block sizes in SPEC CPU2000 benchmarks, after implementing call merging and epilogue duplication. While block fullness is improved, performance suffers.

function calls will necessary entail frequent calls and epilogues. Figure 5.5 shows the block-cut histogram with these issues resolved. While the number of small blocks decreases, we noticed that performance is substantially worse with these transformations, particularly epilogue-duplication, up to 25% worse on average. This degradation is because calls and epilogues perform a great deal of memory and register operations, which need to be nullified or predicated down other paths; thus the overhead introduced is not worth the gain in block size.

- **Libraries:** Bars marked "Libraries" include both calls into libraries and blocks within libraries. Calls into libraries pose a particular problem because they cannot be resolved until link time. While post-link optimization is possible on some systems (though it is unsupported by Scale), libraries pose an additional problem, because they are frequently linked dynamically. Dynamic linking has a system level benefit for compati-

bility and security reasons, as well as for reducing the overall memory usage of the system. Unless the dynamic linking system can also perform binary-level inlining and optimization, calls into libraries will likely remain opaque. (The library functions themselves, however, can be almost arbitrarily optimized, as we have done with particular mathematical functions such as floating-point division and square root.)

- **Indirect branches:** Indirect branches are usually generated to handle switch statements. When the switch statement is sufficiently simple these indirect branches can be converted to predicated branches and thus merged into a single block. For more complex switch statements, however, this alternative is impractical and inefficient. In practice, since indirect branches occur rarely in the SPEC CPU2000 benchmarks, they are not considered here a significant impediment to block formation. In the future, however, that the popularity of dynamic and object-oriented languages with virtual dispatch may dramatically increase the frequency of indirect branches. Thus it will be important to either solve this problem using speculative compilation techniques—preferably using a dynamic compiler; or by architecting around the problem by supporting variable-size blocks.

To demonstrate the high variability in block size and cut rationale that occurs in practice, we provide the detailed analysis for the 19 SPEC benchmarks that are compatible with our infrastructure. Several benchmarks exhibit

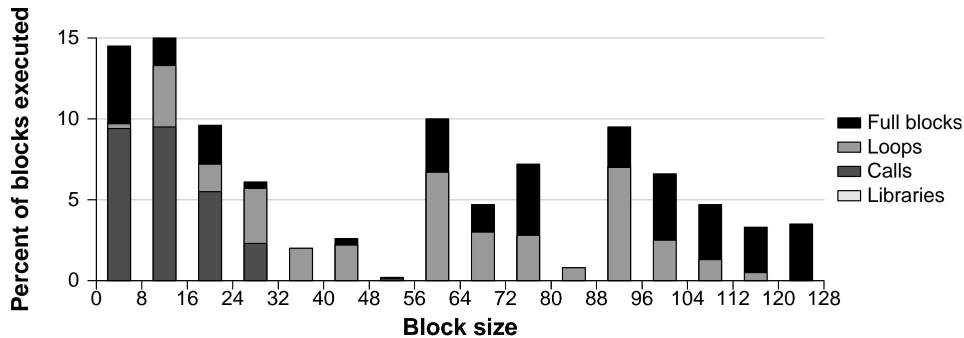


Figure 5.6: Cut analysis of 256.bzip2

roughly the U-shaped curve shown above. 176.gcc, which is arguably the most complex SPEC benchmark, has such a distribution as shown in Figure 5.9. 186.crafty and 254.gap, shown in Figures 5.7 and 5.8 respectively, have a similar distribution, but each benchmark contains a spike in the center of the size range. The spikes in crafty at 56 instructions and gap at 80 instructions provide evidence supporting an architecture with fine-grained variable-size blocks, rather than simply supporting a large and small size.

Two benchmarks with notable distributions are 164.gzip and 301.apsi, shown in Figures 5.10 and 5.18. These benchmarks are dominated by blocks of size 80. Inspecting the traces reveals a single dominant block for each of the two benchmarks, one from `longest_match` in gzip, and one from `apsi_` in apsi. The presence of such significant blocks in the center of the range of sizes argues further for variable-size blocks. If the maximum block size were simply reduced to 64, the compiler would be unable to form such a large block, but with the maximum fixed at 128, over 30% of the block space is wasted. These programs require a more flexible approach.

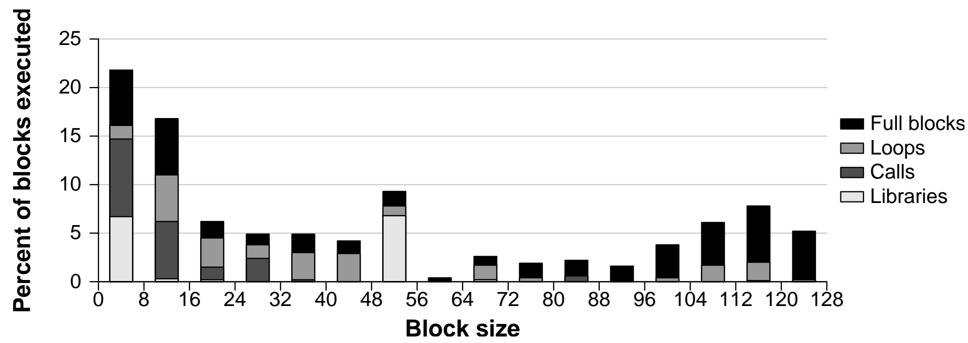


Figure 5.7: Cut analysis of 186.crafty

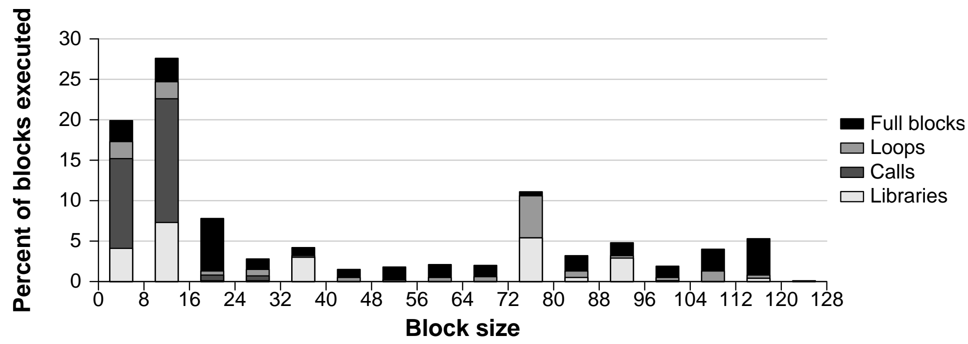


Figure 5.8: Cut analysis of 254.gap

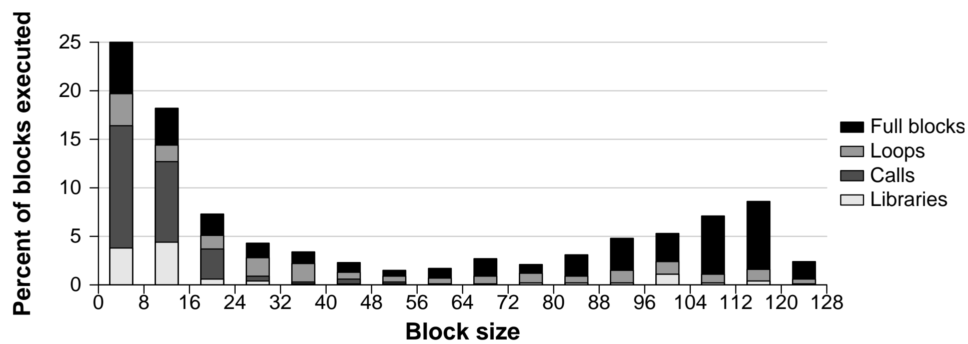


Figure 5.9: Cut analysis of 176.gcc

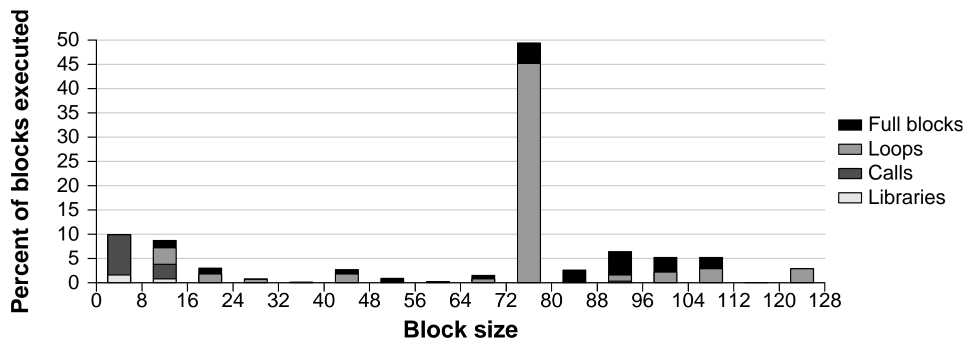


Figure 5.10: Cut analysis of 164.gzip

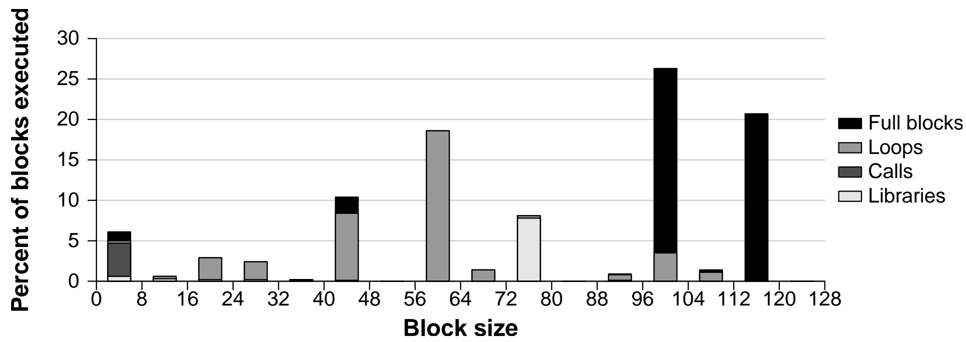


Figure 5.11: Cut analysis of 181.mcf

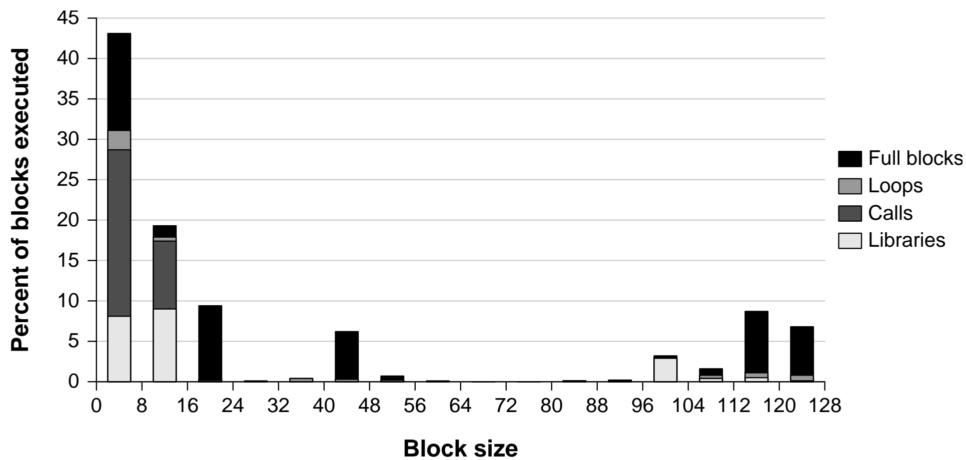


Figure 5.12: Cut analysis of 197.parser

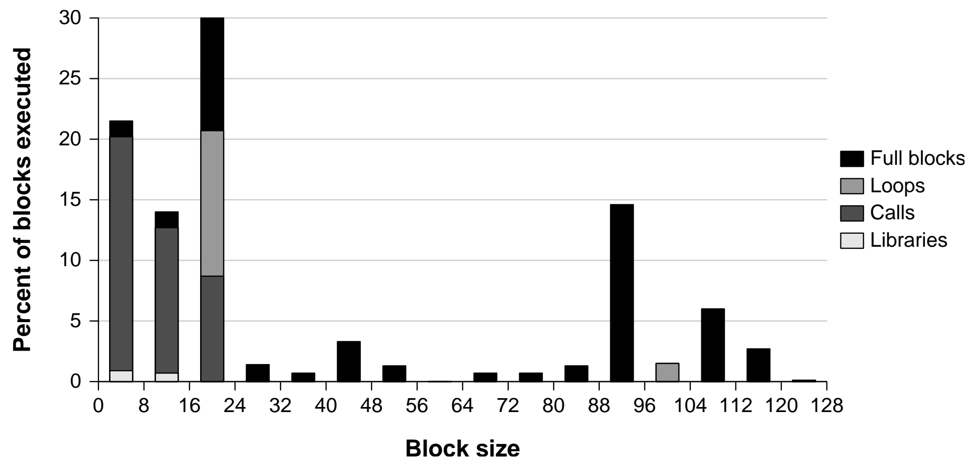


Figure 5.13: Cut analysis of 253.perlbmk

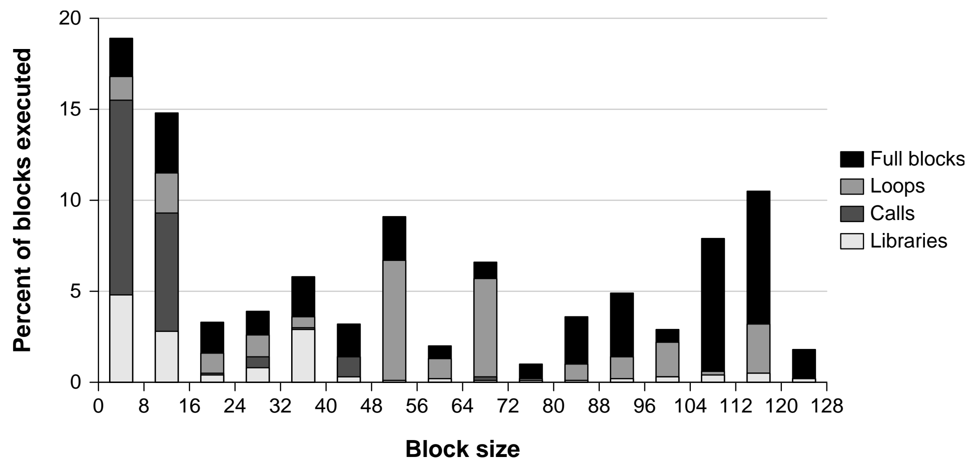


Figure 5.14: Cut analysis of 300.twolf

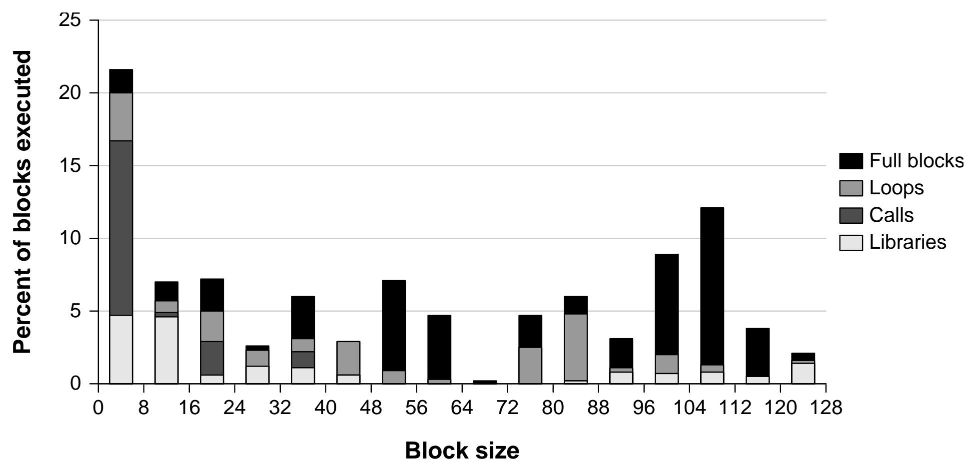


Figure 5.15: Cut analysis of 175.vpr

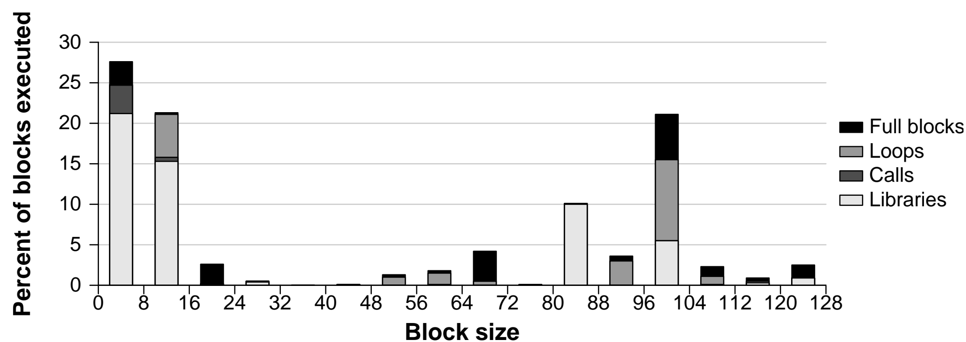


Figure 5.16: Cut analysis of 188.ammp

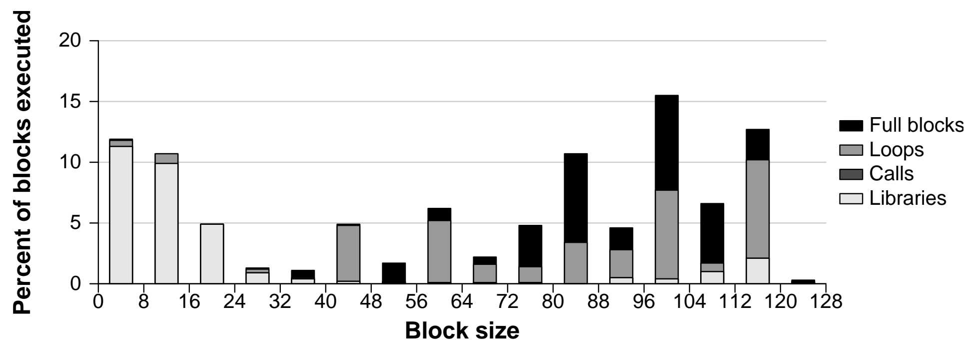


Figure 5.17: Cut analysis of 173.applu

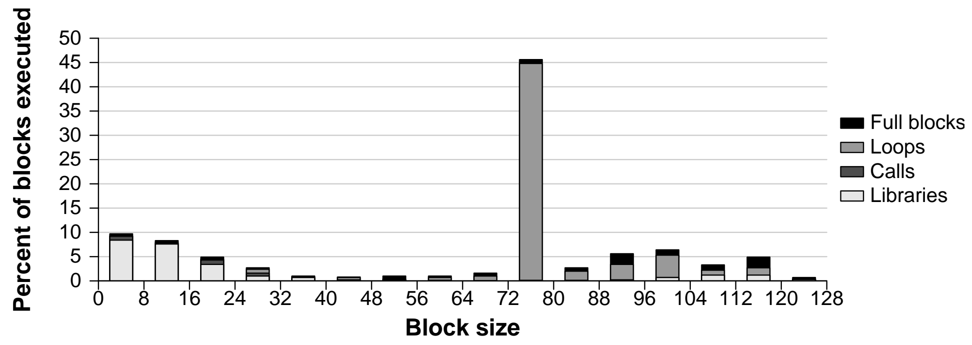


Figure 5.18: Cut analysis of 301.apsi

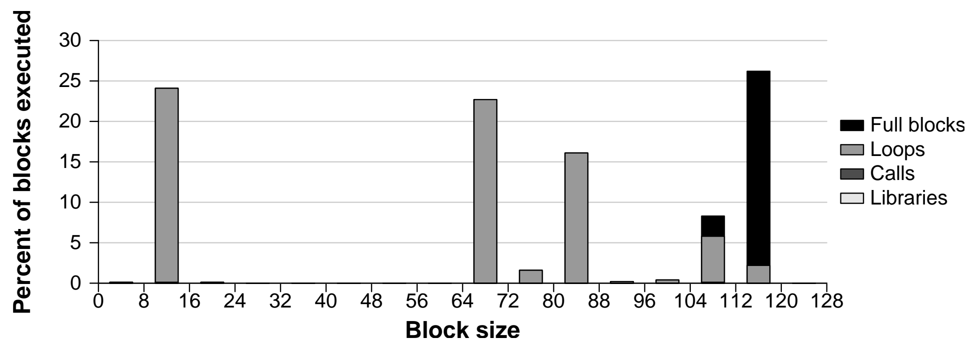


Figure 5.19: Cut analysis of 179.art

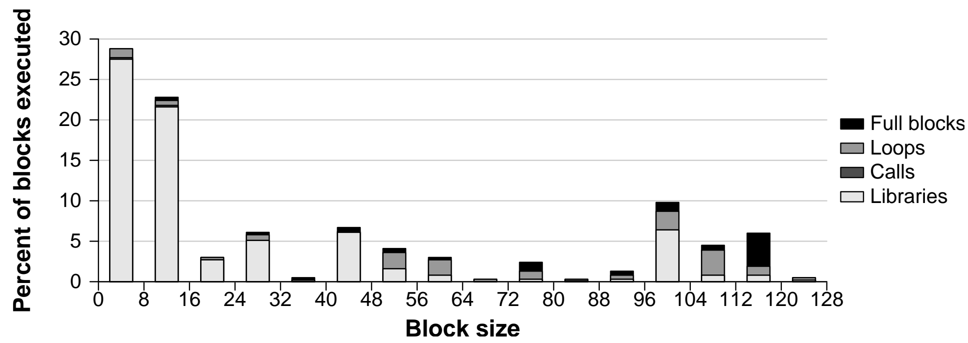


Figure 5.20: Cut analysis of 183.equake

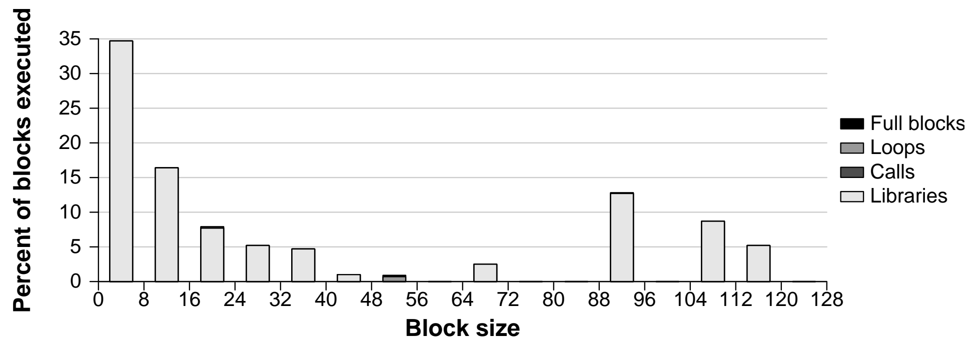


Figure 5.21: Cut analysis of 177.mesa

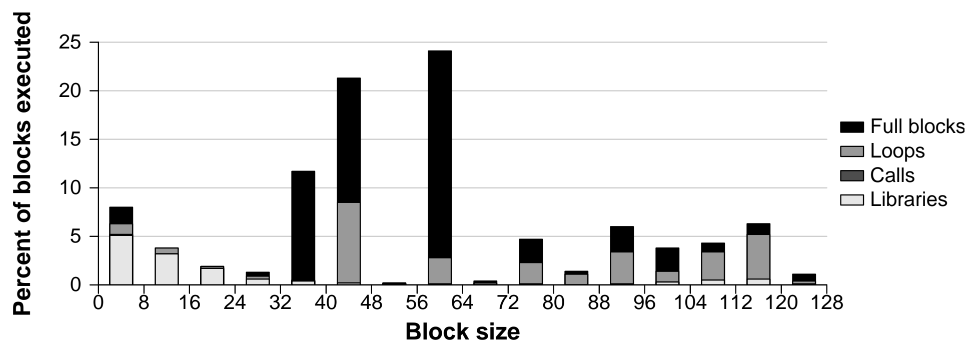


Figure 5.22: Cut analysis of 172.mgrid

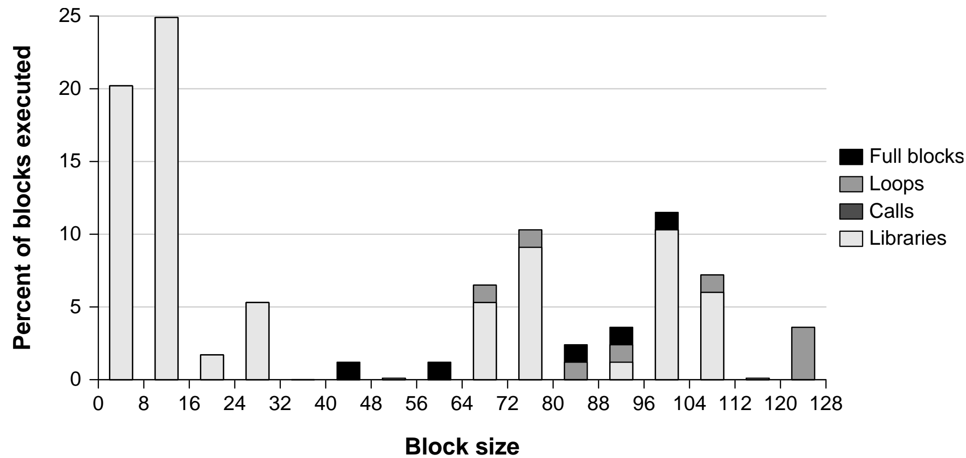


Figure 5.23: Cut analysis of 171.swim

The high proportion of small blocks that cannot be merged due to fundamental structural constraints in the code suggests that variable-size blocks may prove beneficial. The compiler *can* form large blocks, and in many cases it does. The benefits of these large blocks are offset, however, by scenarios in which the compiler fundamentally cannot form larger blocks due to structural constraints. Variable size blocks allow the compiler to exploit the advantages of large blocks when the compiler can form them, without wasting resources when the compiler cannot. Supporting variable-sized blocks complicates the microarchitecture, however, and the next chapter details the changes required.

5.3 Summary

This chapter analyzes the performance of the TRIPS system from the perspective of the compiler's capabilities. We show performance of the TRIPS system on compiled code and compare it to a leading superscalar processor as

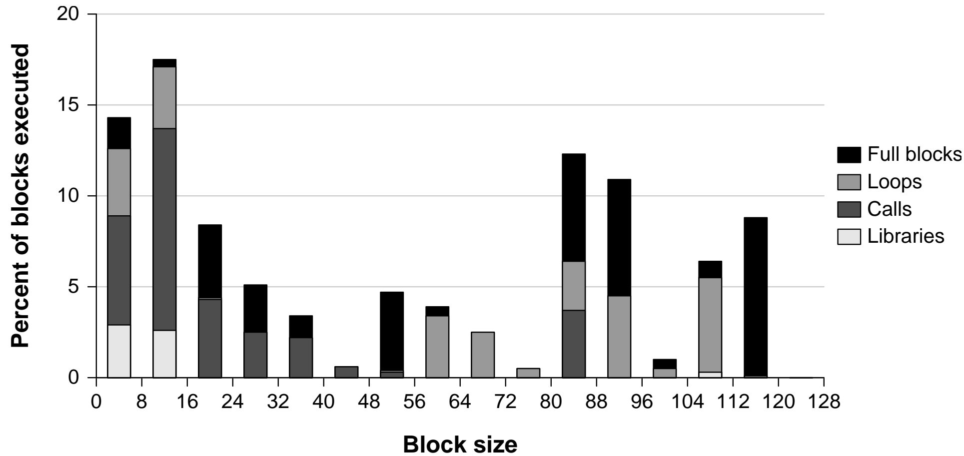


Figure 5.24: Cut analysis of 168.wupwise

of 2008, Intel's Core 2. We supply detailed performance counter statistics to explain the performance differential, and then provide an in-depth analysis of the compiler's block formation capabilities, and the causes of its shortcomings. These findings motivate significant microarchitectural and compiler optimizations that are explored in the rest of this dissertation.

The performance comparison between TRIPS and commercially available platforms yields mixed results. While prior work has shown that hand-coded TRIPS programs can achieve high performance relative to compiled code on other platforms, TRIPS performance on large, compiled bencmarks (SPEC CPU2000) does not generally compare favorably to commercial platforms. While floating-point performance is competitive, integer performance falls far behind. The evidence currently does not support transitioning to EDGE ISAs for performance reasons.

The evidence suggests that the cause of poor performance, particularly

on control-intensive integer benchmarks, is the small size of blocks found in these applications. EDGE ISAs, and TRIPS in particular, rely on the compiler to create very large blocks of instructions in order to amortize the overheads of block-atomic execution and expose sufficient parallelism within the window to tolerate the latencies of distributed execution. The average block size of SPEC INT benchmarks does not provide sufficient efficiency to the processor. Evidence that block formation is one of the causes of poor performance can be seen by inspecting performance counters: instruction cache misses are far higher on TRIPS than on commercial platforms.

By instrumenting the TRIPS compiler we categorize all possible blocks to determine what are the most frequent causes of very small blocks. This evaluation indicates that the causes of small blocks are fundamental and difficult to engineer around. The most significant cause of small blocks are function calls, which—short of massive inlining that could itself negatively effect program performance—cannot be removed. This discovery indicates that block-atomic execution has a serious shortcoming when faced with modern software development techniques in which small, frequent function calls are the norm.

At the same time, the distribution of block sizes gives hope for block-atomic execution. While small blocks are very common, the compiler is also frequently able to create very large, full blocks of instructions. This result indicates that a large block size is feasible, if the microarchitecture can also tolerate the multitudes of small blocks without a tremendous performance handicap. Effectively the microarchitecture should be able to hybridize block-

atomic execution with superscalar execution to achieve the best of both worlds. If performance is close to a superscalar when only small blocks are present, then large blocks can become a performance opportunity rather than a requirement. Such a hybrid design effectively compensates for the compiler's shortcomings while taking advantage of its capabilities.

The distribution of block sizes suggests that—rather than simply lowering the maximum block size—the architecture and microarchitecture should be able to use variable-size blocks to suit the compiler's capabilities. A variable block size with sufficient granularity is able to reduce waste in the processor's instruction cache and potentially support a large number of blocks in flight. Chapter 6 addresses the design of such a microarchitecture and evaluates its performance.

While this chapter has overall painted a pessimistic view of TRIPS performance, we believe there is significant long-term potential for these ideas. TRIPS is necessarily a research design which pushes certain technological decisions to an extreme level. The decided-upon partitioning of resources does not necessarily match the best partitioning for any particular technology node, nor does it tune many microarchitectural structures to their fullest. Comparing a research design to commercial designs running conventional, commercial workloads is this worst-case scenario for such a project, and TRIPS performs relatively well given its immaturity. With microarchitectural innovations, both proposed in this dissertation and elsewhere, as well as continued compiler optimizations, the performance of EDGE architectures will continue to increase.

Chapter 6

Variable-Size Blocks

As Chapter 5 shows, the compiler cannot consistently form large blocks of instructions due to control flow conditions such as function calls and loops. Furthermore, the frequency of function calls forces the creation of very small, frequently executed blocks. This condition implies that reducing the maximum block size will improve block fullness only slightly, as we have shown. To significantly improve the efficiency—in terms of useful instructions per block—the microarchitecture must support variable-size blocks. This chapter presents the design, implementation, and evaluation of an instruction set and microarchitecture that supports variable-size blocks.

Variable-size blocks are particularly helpful in the context of a composable dynamic multicore such as TFlex. Dynamic multicores promise the ability to adapt to a changing workload by allocating increased resources to the sequential portion of a program, thereby alleviating Amdahl's serial bottleneck [32]. We show that programs exhibit very different performance characteristics with respect to block size. While large blocks benefit data-intensive applications, control-intensive applications seem to perform best with limited block sizes. With variable-size blocks, a single microarchitecture can accom-

modate both extremes, and can adapt to the particular needs of the workload.

To support variable-size blocks effectively requires a slightly different instruction set than a fixed-size block architecture such as TRIPS. In particular, reducing the size of the block header is important to improving cache utilization. While the changes required are relatively minor, it is important to note them precisely to understand the design of the microarchitecture. We describe the ISA in Section 6.2.

We demonstrate a microarchitectural implementation of variable-size blocks. Variable-size has meanings at several levels of the processor: main memory, L2 cache, L1 instruction cache, and instruction window. We introducing variability at each level and describe the microarchitectural support necessary. Allowing variable-size blocks in the instruction window is particularly interesting, because it entails allowing multiple blocks in-flight per core, a significant design departure from earlier TFlex designs. While this capability entails an increase in microarchitectural complexity, we show that its performance benefits can be considerable, particularly when executing a program on a small number of cores.

Using a TFlex simulator modified to accept variable size blocks allows us to compare the performance impact of various block sizes, as well as the impact of variability. We experiment with several maximum block sizes to determine the impact of increasing block size. These trends are strongly program dependent. Large blocks are primarily helpful for data-intensive, floating-point applications; integer applications show much less improvement. Furthermore,

we show that by supporting more small blocks in flight, performance can be improved on integer benchmarks that favor small blocks.

With these trends in mind we argue that the microarchitecture should be flexible enough to take advantage of large blocks when it is fruitful to do so. Simply reducing the block size from 128 to 64 would improve performance on integer benchmarks, but sacrifice floating-point performance. Instead, the microarchitecture should provide support for smaller blocks as well as large blocks when that is beneficial. We experiment with these configurations over a full range of composed configurations. These results indicate that integer benchmarks are less scalable to high core counts, and may be run with greater efficiency by supporting more, smaller blocks in flight on fewer cores. In Chapter 7 we explore compiler heuristics for several core counts, to see if the compiler can automatically detect such scenarios.

We show in this section that the granularity of atomic execution units is an important machine parameter. The particular block size chosen for an application can have significantly varied impact on performance depending on the application. Because of the compiler's ability to deal with fixed-size blocks, there is no one-size-fits-all approach that works well for all applications. Thus some level of variability should be supported by the microarchitecture to achieve good performance.

6.1 Atomicity and Composability in EDGE Architectures

To explain the implementation of variable-size blocks (and multiple sizes of fixed blocks) we first review the baseline TFlex microarchitectural execution model and describe the aspects which are important to block structured execution. We first explain block-atomic execution as it is implemented in TFlex. We follow by discussing composability, since that is an important architectural parameter in this study. Composability dramatically affects the choices of block-size (and vice-versa) and as we shall see in Chapter 7 also affects the best heuristics for compilation.

6.1.1 Block-Atomic Execution

The smallest atomic execution unit in an EDGE architecture is the block, which contains a header that summarizes global communication and a body of individual instructions. The microarchitecture fetches, issues, and commits blocks atomically. At least one block is non-speculative. Next-block prediction may speculate subsequent blocks and thus more than one block is typically in flight. Regardless, the entire block commits, or it is flushed from the pipeline due to a misspeculation. The header indicates which global registers the block reads and writes. Summarizing the register information at the block level helps the architecture correctly rename registers when multiple blocks are in flight.

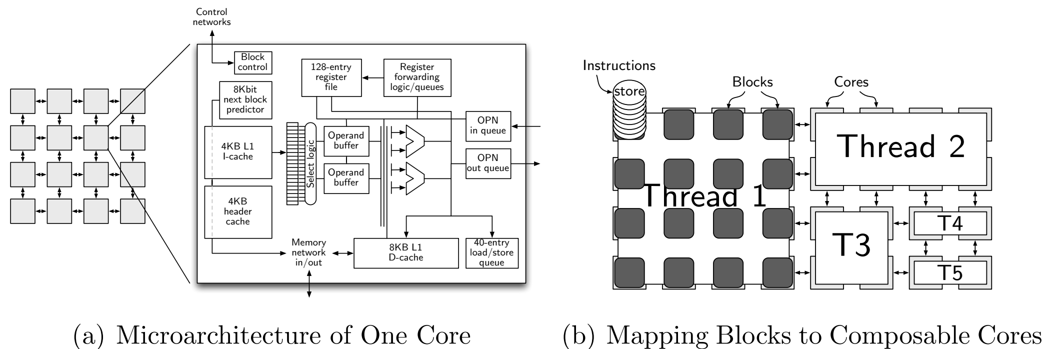
Within a block, instructions communicate directly in dataflow fashion.

This characteristic is key to the scalability of an EDGE architecture. Because the ISA uses a hierarchical namespace–global registers between blocks, and temporary dataflow names within blocks–the processor can efficiently support a large number of instructions in flight by putting several large blocks in flight. These reads and writes are the global communication mechanism between blocks. RISC and CISC ISAs also globally communicate with registers, but at the granularity of a single instruction and each access is through a shared register file and rename table. Point-to-point communication within a block eliminates these shared structure accesses and provides scalability to block-atomic architectures.

6.1.2 EDGE Support for Composability

Dynamic multicore processors, which adapt their parallel resources to the workload at hand, have been shown to provide the best performance trade-off given a mix of sequential and parallel work [32]. Block-atomic execution enables an EDGE processor to execute programs on few or many cores by executing blocks independently on cooperating cores, using a shared register file only for coarse grained communication [37, 60]. While composability can be achieved using a RISC or CISC ISA as in Core Fusion, fine-grained register communication and relatively frequent control decisions physically limit composability to a small number of cores [35].

In this chapter we evaluate performance on the TFlex microarchitecture simulator, which models a composable EDGE chip multiprocessor consisting



(a) Microarchitecture of One Core (b) Mapping Blocks to Composable Cores

Figure 6.1: TFlex: A Composable Chip Multiprocessor EDGE Architecture

of moderately powerful cores [37]. The base microarchitecture dynamically composes cores, as depicted in Figure 6.1(b). We vary these microarchitectural parameters in Section 6.4 to understand the effect of various block sizes on the instruction cache and issue queue.

TFlex can compose multiple processors to accelerate the execution of a single thread, as depicted in Figure 6.1(b). When multiple cores execute a single thread, TFlex dynamically maps each block to a single core [60] allowing multiple blocks execute in parallel on different cores. To execute variable-size blocks in this architecture requires a modified instruction cache and issue queue design, as well as scaling up the renaming logic and block control. The next subsection overviews the instruction queue and issue logic support required by a fixed-size design. Section 6.3.2 describes the variable-sized block design, and Section 6.4 evaluates the performance of fixed- and variable-size designs.

6.2 Instruction Set Support

Instruction set support for variable-size blocks is straightforward. The primary task is to reduce the size of the block header to a single cache-line to avoid wasting space with smaller blocks. Variable-size blocks will also be compressed in memory to 32-byte boundaries, which entails branches and calls must be appropriately offset. This section details the necessary modifications.

To take advantage of variable-size blocks at the 32-byte granularity, the block header should also be no more than 32 bytes, or else the advantage gained by fine-grained variability will be completely lost. The TRIPS block header already contains some metadata. The variable-size extensions encode the block size more precisely (four bits, to indicate each of 16 possible sizes). The remainder of the header is taken up by a store mask, to detect block completion, and read/write masks to allow register forwarding to be set up immediately upon fetch. We reduce the size of the register set to 64 to enable all possible reads and writes to be encoded in the 32-byte header.

The remainder of the TRIPS block header contains read and write instructions, which are often no-ops. To remove these instructions from the header, we encode read instructions as ordinary instructions that are scheduled in the body of a block. These special read instructions issue from the issue queue as ordinary instructions, and once issued retrieve a value from the register encoded in the instruction. The value is then forwarded to the targets of the read, in a manner similar to load instructions. Because reads are often on the critical path of a block it is important to get them in flight

early. We propose executing reads during decode, as the instructions are being transferred from the cache to the queue. Because these instructions require no functional units, scheduling should not be complicated by this optimization.

Writes are a simpler matter, because they are destinations only and do not have to encode targets. To eliminate writes from the header we simply encode the precise register number in the instruction itself, rather than a location within a write queue as was done in the TRIPS ISA. The reduction of registers from 128 to 64 is important here to ensure there are enough bits in the instruction to encode the register. By encoding the register number directly rather than relying on implicit bits, we also simplify the compiler's register allocation task, because bank restrictions are no longer necessary. We do not, however, evaluate the effect of this restriction, because spills are in practice extremely rare even with the reduced register set.

To simplify the interpretation of the results presented in this chapter, all benchmarks are compiled using this modified ISA, including the results for fixed size blocks. This decision allows for easy implementation of smaller block sizes as well as more direct comparison between fixed and variable-size performance measurements, while holding the compiler constant. The performance impact of the ISA change itself was negligible, usually $< 1\%$.

6.3 Microarchitecture

The chief advantage of fixed-size blocks is the simplicity of hardware implementation and ease of distributed execution. The advantage of variable-size

blocks is more flexibility and better resource utilization, at the cost of increased complexity. This section contrasts the microarchitectural support necessary for fixed-size blocks with the support necessary for variable-size blocks, and describes the advantages and disadvantages of each scheme. For more complete details on the implementation of variable size blocks, we refer to TRIPS design documents. We describe a few possible design alternatives for variable-size blocks to improve performance, as well as the configuration that is evaluated in Section 6.4.

6.3.1 Fixed-Size Blocks

The instruction cache design with fixed-size blocks is simple. Each block occupies a cache region, which is physically partitioned into lines. Particular lines are reserved for the start of a block. Because a single cache tag corresponds to an entire block, hit detection is fast, and there are no partial conflicts between blocks. If the cache tag matches, then the entire block is present. On a miss, the cache evicts the entire victim block and fetches the new block from a lower level of the memory hierarchy.

This design has two advantages. The first is simplicity. Because blocks are either present or not present, checking for a hit is a single, fast tag comparison. The second advantage is distribution. If the instruction cache is partitions the blocks across multiple tiles or cores, as in TRIPS and TFlex, the architecture detects the hit at a single predetermined location, and finds the remaining body chunks at predetermined locations. It makes no additional

tag comparisons.

The obvious disadvantage to caching fixed-size blocks is underutilization when blocks are not full. For large programs with complex control flow, this penalty can be significant. On the TRIPS microarchitecture, several SPECINT benchmarks suffered upwards of 10 instruction cache misses per 1,000 instructions, severely limiting performance [27].

Issue queue design is also simple for fixed-size blocks. Each core is given enough instruction window slots to accomodate a fixed number N of blocks in flight, and instructions within the queue are located by concatenating their location within the block with the block's offset into the physical queue. Another consideration is register renaming: since renaming is done on at the block granularity, supporting a single large block size reduces the total renaming hardware necessary.

6.3.2 Variable-Size Blocks

This section describes the changes to the TFlex microarchitecture described in Section 6.1 that are needed to support variable-sized blocks. We discuss changes to the instruction cache and issue logic below. In addition to these changes, supporting variable-size blocks requires support for potentially more total blocks in flight, which requires additional register renaming resources and additional block control logic.

An instruction cache supporting variable-size blocks operates similarly to a conventional instruction cache. Cache lines are tagged by address and hits

are detected on a per-line granularity. With this organization, an instruction block occupies only the number of lines required by its actual size. Because a block can start at any cache line, each line must contain storage for tags. With these additional tags the cache can store more total blocks, but must manage the tags for each line within a block, rather than a single tag per block. Furthermore, when a logical cache is distributed to multiple cores, it is possible that lines corresponding to the same block reside on different cores. In TFlex, memory addresses are distributed across all participating cores using a hash of the address. We model remote instruction caches as in the baseline microarchitecture. The additional latency we model for fetching from remote cores assumes the same distributed fetch protocol as the baseline TFlex microarchitecture.

Variable-size blocks require more complex conflict detection in the cache because of the possibility that blocks can partially conflict. There are essentially two methods of dealing with this complexity. The first method simply mimics a conventional processor by fetching lines on demand and stalling while servicing misses. This method does not attempt to optimize miss detection, rather waiting until a particular cache line is needed to report the miss. The second method detects misses earlier by marking the first line of a block as invalid whenever *any* line is evicted. This method requires additional bits per line that identify the block header. We model the first method, which is more conservative both in terms of performance and complexity.

Variable-size blocks in the instruction cache provide better cache uti-

lization because they do not need NOPs to pad each block to the required length. When cache capacity is small compared to the size of the program, this property is very important to good performance. The complexity of a variable-size design is similar to that of superscalar caches, but more complex than a fixed-size design.

A variable-size issue queue requires additional resources for renaming. Renaming requires expensive content-addressable memories for wakeup and logic for detecting dependences. It is not scalable to extremely large instruction windows. Power and complexity limitations have held superscalar processors to around 100 in-flight instructions, and similar limitations constrain the number of in-flight blocks. A large number of renaming registers are needed, as are large issue queues and a highly accurate branch predictor. Increasing the effective size of the instruction window by allowing more blocks in flight when blocks are smaller is a critical performance optimization, however. In Section 6.4 we evaluate variable-size blocks assuming maximal renaming resources, up to 256 blocks in flight. This limitation is not incomparable to the size of the instruction window in instructions of a modern superscalar processor.

Distributing a window of variable-size blocks is more complex than with fixed-size blocks. To support a variable number of blocks in the window requires an additional table of indices to map each block into the window, since a static slice cannot be known. For this work, we assume that blocks are mapped to a single core. This mapping simplifies the design of a window for

variable-size blocks, since all the instructions in each block are local to a particular core. Distributing smaller blocks across several tiles as in TRIPS would increase the overhead of executing smaller blocks, reducing the performance advantage of implementing variable-size blocks.

6.4 Architectural Results

To evaluate the importance of block size and granularity, we simulate a range of sizes on all composed topologies. We measure the effects of block size in isolation; block size in conjunction with mapping multiple blocks per core; and finally full variability. We describe the methodology and results in the following subsections.

6.4.1 Methodology

We evaluate the effects of block size and variability using a cycle-level simulator based on the TFlex microarchitecture [37]. We extend this simulator with ISA support for variable-size blocks, the ability to vary the maximum block size, and the ability to vary the number of in-flight blocks supported by each core. The simulator supports configurations of up to 32 cores, where each core has the microarchitectural parameters described in Table 6.1. As described in Section 6.1, the issue window size at each core depends on the architectural block size and the number of blocks per core supported by the microarchitecture.

We evaluate the effects of block granularity on composability by con-

Parameter	Configuration
Instruction cache	32 KB; 1-cycle hit; 4-way set associative
Branch predictor	Local/Gshare tournament predictor (8K+256 bits, 3-cycle latency) with speculative updates Entries: Local: 64(L1) + 128(L2), Global: 512, Choice: 512, RAS: 16, CTB: 16, BTB: 128, Btype: 256
Data cache	32 KB; 2-cycle hit; 2-way set associative; 1 read/1 write port; 44-entry LSQ
Issue width	Limited dual issue (up to 2 integer and 1 floating-point)
Issue window	Depends on block size
L2 cache	4 MB S-NUCA [36]; 5–27 cycle hit latency depending on address
Main memory	Average unloaded latency: 150 cycles

Table 6.1: Microarchitectural parameters of each TFlex core

ducting each experiment on configurations containing 1 through 32 cores, and measure performance at a combination of block sizes and core counts. The block size given is the maximum architecturally defined size, and the core count is the number of composed processors participating in the execution of each single thread.

The compiler produces blocks up to the specified maximum size, and the processor uses those blocks as the unit of work. The results use the default compiler block formation heuristic from Section 7.2, parameterized by the block size, unless otherwise specified.

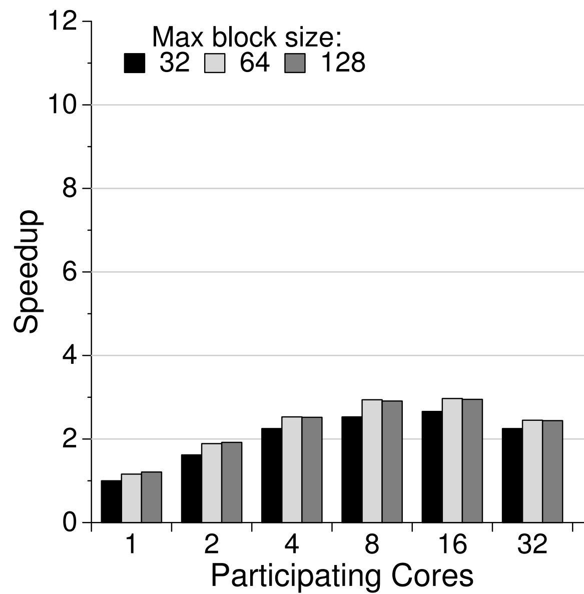
We measure performance on the SPEC CPU2000 benchmarks using reference datasets. To keep simulation time tractable, we use the early SimPoint methodology [64]. When providing average results we separate integer from floating-point benchmarks, as these suites show trends that differ significantly from one another, but are internally similar. We use single-threaded workloads, rather than multiprogrammed or multithreaded workloads, in order to

more fully explore the design space of an individual core, the composition of cores, and their interactions for a single thread.

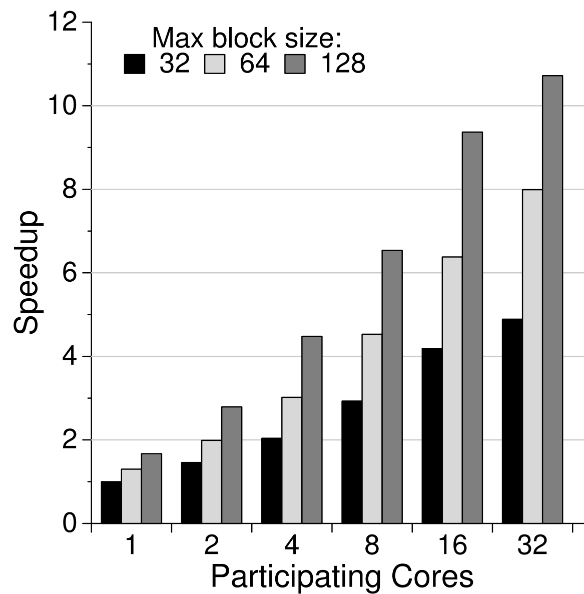
6.4.2 One Block Per Core

First we evaluate the effect of block size on performance in a system with one block executing on each composed core. Each core thus executes a block and the number of blocks in flight is a function of the number of participating cores. We measure performance for block sizes of 32, 64, and 128 instructions, and for core counts from 1 to 32. In this system, the maximum possible instruction window size—assuming completely full blocks—is equal to the block size multiplied by the number of cores. These cores are not microarchitecturally equivalent, of course, since supporting smaller blocks allows building smaller issue queues. However, these configurations separate the performance effect of the block size from other microarchitectural issues, such as the speculation depth, and they explore scalability of a single thread as a function of the number of participating cores.

Figure 6.2 shows the geometric mean speedup of the SPECINT and SPECFP benchmarks, normalized to the cycle count of a single TFlex core with support for 32-instruction blocks. The performance trends differ markedly between SPECINT and SPECFP. At all block sizes, floating-point performance scales well with core count, while integer performance does not. With 16 participating cores and 32-instruction blocks floating-point performance improves by a factor of 4, while integer performance improves by a factor of 2.5.



(a) SPECINT



(b) SPECFP

Figure 6.2: Performance with one fixed-size block per core and maximum fixed block size of 32, 64, and 128 instructions. Thus, fewer hardware resources are required with smaller maximum block sizes.

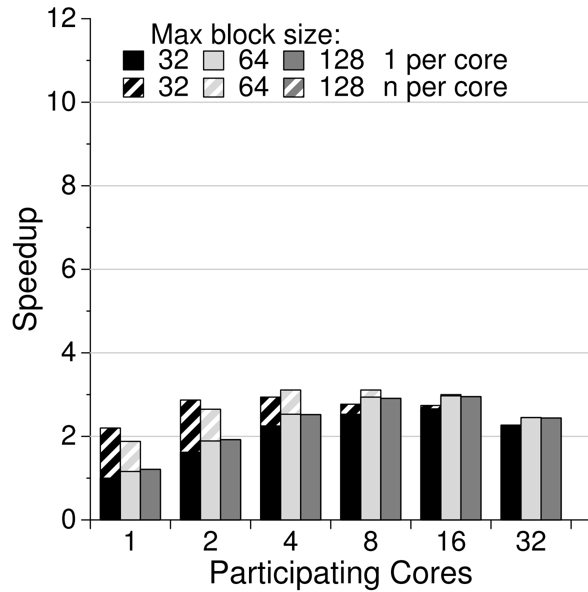
Furthermore, performance continues to improve up to 32 cores on floating-point programs while integer performance changes little from 8 to 16 cores, and actually reduces at 32.

The effect of fixed-size blocks on performance is also markedly different between SPECINT and SPECFP. On SPECFP, the difference between 32- and 128-instruction blocks ranges from a factor of 1.6 at one core to a factor of 2.2 at 32 cores. While 32-instruction blocks achieve a speedup of 4.4 at 32-cores on SPECFP, 128-instruction blocks achieve an 11x speedup.

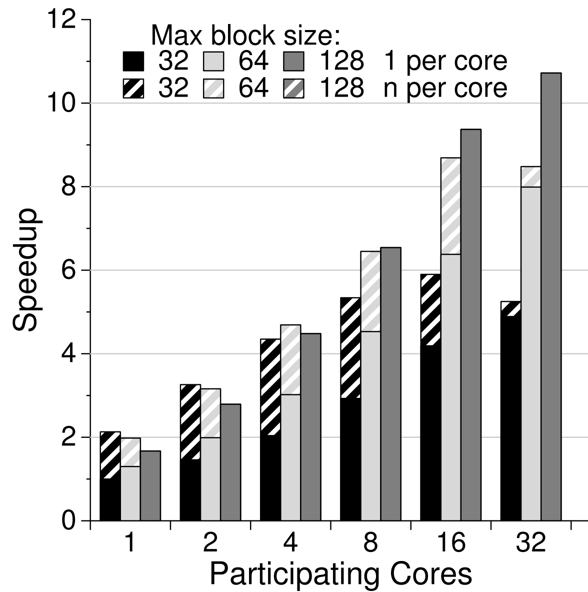
On SPECINT benchmarks, 32-instruction blocks perform only slightly worse than 64, which in turn matches or exceeds the performance of 128. The best-performing block size and core combination is 64 instructions blocks with eight participating cores. The performance characteristics of integer benchmarks are explained by the code characteristics described in Chapter 5. Blocks in integer benchmarks are frequently small, thus yielding no benefit for large architectural block sizes. When integer benchmark blocks are large, they are typically deeply predicated, which leads to the inclusion of many useless instructions. Also, many control dependences are converted to data dependences, replacing narrow speculation from branch prediction with wide speculation from predication.

6.4.3 Multiple Blocks Per Core

Because integer programs lend themselves naturally to smaller blocks, allowing more small blocks in flight per core may yield better performance



(a) SPECINT



(b) SPECFP

Figure 6.3: Performance with 128 instructions per core and fixed-size blocks. Thus, with maximum block sizes of 32, 64, and 128 instructions, there are 4, 2, and 1 blocks per core, respectively, and all configurations require the same number of issue queue slots.

than fewer large blocks. For example, four 32-instruction blocks in flight per core requires the same issue queue space as a single 128-instruction block, but may lead to better resource utilization. While supporting more blocks in flight increases complexity, especially with a large number of composed cores, the additional performance may be worth that cost.

Figure 6.3 shows performance with multiple blocks per core for smaller maximum sizes. The increased window size generally translates to significant performance improvements, particularly at low core counts where the speculation depth is low. Interestingly, the maximum performance on the SPECINT benchmarks always occurs when a maximum of 16 blocks are in flight: four cores with 32-instruction blocks, two cores with 64, and one core with 128.

SPECINT performance in this configuration saturates quickly. The global best speedup of 3x over the baseline occurs at four cores with 64-instruction blocks. Even one core with 32-instruction blocks, however, yields a 2.2x speedup. Most of the performance comes from the increased window size rather than the improved issue width. The smaller blocks have an advantage at lower core counts because they are less deeply predicated and thus do not waste the machine's limited resources on misspredicated instructions; instead they use limited predication and take advantage of more accurate dynamic branch prediction. Only once the machine has a large issue width does it make sense to use larger blocks. The larger window and increased issue width enable the overheads of predication, and speculating further via branch prediction does little good due to the geometric increase in branch missprediction

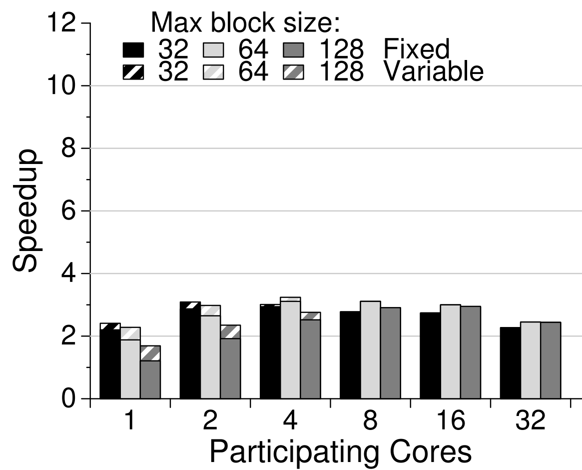
frequency.

SPECFP benchmarks do not perform nearly as well with smaller blocks. Even though all hardware configurations provide a 128-instruction window per core, larger block sizes still outperform smaller blocks. The reason for this is that intra-block communication is much more efficient than inter-block communication. Since SPECFP blocks are generally full of useful instructions with very little predication, there is no downside to using large blocks, and the efficiency of communication produces a significant performance win.

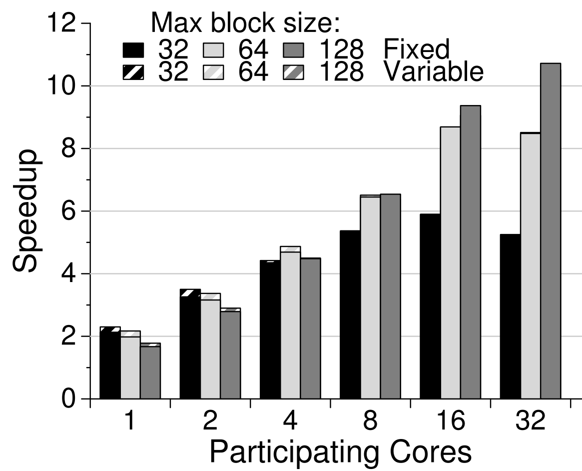
These results motivate the flexibility of the variable-size blocks architecture. Ideally, a system should be able to achieve the high floating-point performance of 128-instruction blocks, while still achieving efficient performance on integer benchmarks with smaller block sizes.

6.4.4 Variable-Size Blocks

We measure performance with variable-size blocks at an eight-instruction granularity as described in Section 6.3.2. As in the previous sections, we use 1–32 cores to show the effect at each configuration. We use maximum block sizes from 32 to 128 instructions, as in previous sections. Note that these maximum block sizes do not necessarily reduce the efficiency of the machine as they do in the fixed-size block experiments. Because each configuration supports variable-size blocks with eight-instruction granularity, the maximum block size is primarily a compiler target. Using larger or smaller maximum block sizes may allow for various microarchitectural simplifications, but we do



(a) SPECINT



(b) SPECFP

Figure 6.4: Performance with 8-instruction granularity variability in the instruction window with various maximum block sizes using 128 instructions per core.

	Granularity				
Benchmark	**8**	**16**	**32**	**64**	**128**
bzip2	0.94	0.87	0.79	0.60	0.44
gzip	0.95	0.90	0.77	0.64	0.50
mcf	0.96	0.91	0.87	0.77	0.71
parser	0.94	0.88	0.75	0.57	0.40
vpr	0.94	0.89	0.79	0.61	0.43
applu	0.94	0.84	0.70	0.52	0.35
art	0.97	0.93	0.89	0.83	0.79
equake	0.93	0.81	0.64	0.46	0.29
mesa	0.94	0.86	0.73	0.56	0.39
mgrid	0.96	0.86	0.78	0.61	0.48
swim	0.94	0.83	0.66	0.48	0.30
wupwise	0.95	0.92	0.85	0.77	0.69
average	0.95	0.88	0.77	0.62	0.48

Table 6.2: Efficiency, in terms of real instructions to total words fetched, of variable-size blocks with different granularities.

not address that issue.

Table 6.2 shows the efficiency of several granularities of variable-size blocks, from eight instructions up to 128 instructions . These efficiencies are much higher than those achieved by reducing the maximum block size, as shown in Table 6.3, which we repeat here from Chapter 5 for convenience. At the eight-granularity, 95% of the instruction words fetched are real instructions rather than `NOP`s, much higher than the 56% efficiency achieved by fixed-size blocks when the maximum size is reduced to 32 instructions. While there are several possible granularities, this section evaluates the performance of eight-instruction granularity only. The complexity of implementing variable-size blocks is roughly equivalent at any granularity down to the cache line level, so

	Maximum Block Size		
Benchmark	**32**	**64**	**128**
bzip2	0.62	0.56	0.44
gzip	0.59	0.53	0.50
mcf	0.67	0.67	0.71
parser	0.52	0.51	0.40
vpr	0.53	0.48	0.43
applu	0.49	0.44	0.35
art	0.75	0.65	0.79
equake	0.48	0.39	0.29
mesa	0.50	0.45	0.39
mgrid	0.41	0.50	0.48
swim	0.50	0.41	0.30
wupwise	0.68	0.66	0.69
average	0.56	0.52	0.48

Table 6.3: Per-benchmark efficiency of blocks for fixed maximum sizes of 32, 64, and 128 instructions.

there is no compelling reason to evaluate the interior points.

Figure 6.4 shows the performance of this microarchitecture. Variable-size blocks improve performance at small core counts, particularly on SPECINT benchmarks, which frequently contain very small blocks. However, overall the performance improvement is limited because the baseline compiler is not aware of variable-size blocks, and thus does not optimize the SPECINT benchmarks by using smaller, more tightly-packed, less-predicated blocks. This result motivates an investigation of compiler support for such architectures, which Chapter 7 explores. Ideally, the compiler would be able to identify programs that exhibit behavior similar to SPECINT or SPECFP and optimize accordingly.

One particularly important data point to notice is the performance im-

provement of variable-size blocks on SPECINT with one TFlex core and a maximum block size of 128 instructions. This scenario is the best case for variable-size blocks and succeeds in improving performance by 40%. This performance increase is due to several factors that variable-size blocks explicitly target. (1) Integer benchmarks have particularly small blocks and generally have large executables, both of which increase instruction cache pressure. (2) A single TFlex core has only 32 KB of instruction cache, so optimizing for cache efficiency is more effective. The TRIPS prototype had only 64 KB of body cache, so TFlex configurations larger than two cores have much larger caches than TRIPS. This design decision was influenced by the performance evaluation described in Chapter 5. (3) Because a single-core, 128-instruction block design can only support a single block in flight, small blocks extremely limit control speculation. Supporting variable-size blocks allows the processor to regain the benefits of speculation in this constrained environment.

In this experiment, the maximum upper bounds on block size do not necessarily have any microarchitectural meaning. If one were to build a microarchitecture that supports 128-instruction blocks with eight-instruction variability and execute binaries compiled with a maximum block size of 32, one would achieve precisely the performance indicated by the 32-instruction block data set in Figure 6.4. This processor is much more flexible than any fixed design. Two reference points where this flexibility stands out are: integer benchmarks with 32-instruction blocks on two cores; and floating-point benchmarks with 128-instruction blocks on 32-cores. On the integer programs, this

configuration gives performance close to the global maximum with high power efficiency by using few cores. On the floating-point programs, this configuration gives extremely good performance, much higher than any other configuration. Furthermore, if one were to execute these workloads simultaneously, as will likely be the case in future CMP systems, the ability to execute integer code on a small number of cores with high performance will allow the system to allocate more cores for data-parallel floating-point tasks, thus maximizing total system performance.

While variable-size blocks yield performance improvements over any of the three fixed block sizes evaluated in the previous section, this performance improvement is secondary to the increase in flexibility. By supporting variable-size blocks, a single microarchitecture can achieve both high performance on integer benchmarks at low core counts by using smaller blocks, and high floating-point performance at large counts by using smaller blocks. However, since the maximum block size is 128, the compiler must use heuristics to determine the appropriate block size for a benchmark. The ability of the compiler to detect appropriate situations in which to create large blocks is therefore the focus of the following chapter.

6.5 Summary

As Chapter 5 demonstrated, support for variable-size blocks is necessary to efficiently use processor resources. While TFlex is better-provisioned than TRIPS in terms of instruction cache space and the number of blocks

supported in flight, we still see improvements in performance by implementing variable-size blocks, particularly when executing in small configurations. Variable-size blocks are effective at reducing the frequency of instruction cache misses, and allow a much more effective utilization of the available instruction queue.

The most striking result is the extent to which integer benchmark performance changes little as the block size is increased from 32 to 128 instructions. Correlating this result with the number of useful instructions in the window indicates that the larger blocks are proportionally less full, that they contain fewer useful instructions due to predication, and they are likely to be mispredicted. This indicates, and our experimental data confirms, that executing more small blocks per core is beneficial to integer performance. This situation occurs both for reasons of microarchitectural efficiency, and for compiler reasons. The smaller upper bound on block size acts as a de facto heuristic for the compiler, limiting the extent to which it can apply predication. As we shall see in the following chapter, limiting predication is a useful heuristic when resources are limited.

While integer benchmarks are shown to benefit little from large blocks or multiple-cores, the opposite holds true for floating-point benchmarks. Because these programs have ample data-parallelism and a high ratio of computation instructions to control transfers, adding more cores allows the machine to improve performance. Similarly, creating larger blocks requires less predication than with integer benchmarks, so the machine can take advantage of

efficient dataflow execution without introducing the overhead of predication. To achieve the best performance in these benchmarks, the processor should therefore support large block sizes and allow composition of many cores.

To automatically take advantage of the differences in these classes of benchmarks, the compiler must be able to identify where it is useful to form large blocks and where it is not. While it is possible that programmers with application knowledge could select an appropriate block size for their application, the common case is that such optimizations are not used. An automatic system would be much more powerful and accessible. As the following chapter describes, such characteristics are difficult to extract from the program source. One possible angle is to consider the fraction of useful instructions: as shown in Section 5.2.2, integer programs generally have fewer useful instructions per block, and the fullness decreases as the block size grows. While it may not be possible to statically identify these characteristics, a dynamic optimization system may be able to take advantage of them.

As workloads become increasingly diverse, the ability of a dynamic multicore to adapt to its workloads will become an ever-more valuable optimization. Future processors will increasingly be called upon to execute a mix of largely serial, control-intensive operations and data-intensive operations. Even within the same program, serial bottlenecks can greatly hinder the speedup of an overall application, a situation that dynamic multicores are better equipped to handle than static alternatives. By introducing the additional parameter of variable-size blocks, the microarchitecture becomes increasingly adaptable

to the workload. Such control-intensive programs can be run more efficiently on fewer cores, freeing more parallel resources to improve the performance of parallel applications.

To achieve this ultimate goal in a manner transparent to the application programmer will require a synthesis of architectural/microarchitectural and compiler techniques. This chapter has shown the architecture and microarchitecture necessary to provide flexibility. Furthermore we have shown that said flexibility can improve performance if the applications can be matched to the appropriate compiler and microarchitectural configuration. In the following chapter we explore the compiler side to show whether and how the compiler can take advantage of microarchitectural flexibility by identifying situations where smaller or larger blocks are beneficial.

Chapter 7

Block Formation Heuristics

To form effective blocks, the compiler must employ a block formation policy. Prior work on block formation considers both if-conversion policy, typically handled by hyperblock formation heuristics, and loop unrolling/peeling, which are usually handled separately. With iterative hyperblock formation, the compiler must combine these heuristics into a single priority function. The appropriate heuristics should indicate when to apply predication, merging, and unrolling, by balancing a variety of factors that interact in complex ways. This chapter describes the factors at work in determining policy, how the compiler measures and applies these factors, and describes an approach using machine learning to rapidly search the policy space [72]. The particular machine learning algorithm is not a contribution of this dissertation, as it has been described in prior work; however, we use the mechanism to discover novel characteristics of an EDGE machine.

Iterative, incremental block formation as described in Chapter 4 provides a straightforward framework for implementing heuristics. At each step of block enlargement, the compiler considers the outgoing edges of a block, ranks them according to some metric, and selects the best (or none). Chap-

ter 4 demonstrated a few possible policies to show the generality of incremental block formation; this chapter considers what those policies should be.

Chapter 6 showed that—with variable-size blocks in place—significant performance improvements can be realized by tailoring the compiler's heuristics to the benchmarks. The integer benchmarks, for example, tend to favor smaller, less deeply predicated blocks, while the floating-point benchmarks benefit from extremely large blocks. The results from the prior chapter use a maximum upper bound on block size as an "inadvertent" heuristic: we discovered the above performance rules simply through experimentation. Ideally the compiler should be able to make good block formation decisions simply by inspecting the code.

Programs have numerous features that the compiler could use to make its decisions. Each feature should describe a characteristic of the code, and have some definable relationship to performance. While there is some recent research on crafting compilers that can automatically construct features, we take a more traditional approach in which the compiler writer lists all possible code features that seem useful [43]. We then use these features to construct policy functions, which combine features using common arithmetic operators, that the compiler uses to decide which blocks are most profitable. In Section 7.4 we list the features we implemented in Scale.

Because policy functions can be arbitrary arithmetic expressions, the policy space is large, complex and nonlinear. To explore this large space, we use machine learning. While we have some intuition about what policies

might be effective and produced a few handcrafted policies, we desired a more systematic approach to searching the policy space. We use a standard genetic programming approach to learning policies [39, 72]. This approach has been demonstrated on conventional hyperblock formation to give speedups in excess of 25% over a highly-hand tuned heuristic. We use the technique as a method for gaining insight about the architecture and compiler involved in this system.

Because benchmarks behave very differently when given varying numbers of cores, we apply learning techniques to create specialized policies for one, eight, and 32 cores. This heuristic exploration reveals that the best policy differs strikingly at low cores counts. With large topologies predication is desirable, because it can reduce the number of branch mispredictions and associated pipeline flushes. With small topologies, however, predication is a performance liability because it over-saturates the limited resources of the processor. The best heuristic discovered for one or two cores uses no predication at all. Interestingly, it does use tail duplication to merge unconditional branches.

We show some improvements using learned policies overall. We train on a reduced set of microbenchmarks to keep simulation time tractable, and show good speedups over that suite. The results, however, do not appear to generalize well to SPEC. While the one-core heuristic still performs exceptionally well, the other heuristics appear to be over-specialized to the benchmarks they were trained on, and do not significantly outperform the baseline heuristic. This behavior is a common problem with machine learning techniques, but

it does demonstrate that there is room for significant compiler-driven speedup, even if the correct heuristics remain elusive.

7.1 Compiler Structure

We implement block formation heuristics in Scale (described in Section 3.3) using iterative, incremental block formation (described in Chapter 4. This iterative approach allows the compiler to solve phase ordering problems between block formation, loop transformations, and scalar optimizations. We use all available back-end optimizations, including global value numbering [65], predicate minimization [67], and various peephole optimizations.

The compiler's block formation policy consists of two nested traversals. The outer traversal of the EDGE block flow graph selects which blocks to enlarge. The inner traversal starts from a block to enlarge, and selects successors to merge. For the outer traversal, we perform a postorder traversal of the loop structure tree. The compiler merges blocks in innermost loops first, then traverses outer loops. While operating on the outer loops the compiler can perform peeling and/or unrolling on the inner loops. Within each loop body, the compiler chooses which block to enlarge next using a topological sort such that a block will not be enlarged until all of its predecessor blocks have been enlarged. This order ensures that when a block is enlarged, its successor blocks are still basic blocks and are thus more amenable to merging. This organization allows the compiler to easily combine unrolling and peeling with the rest of block formation, without needing to rely on subtle heuristics to

guide the compiler towards unrolling or peeling.

After selecting the next block to enlarge, the compiler makes one critical decision repeatedly: given the existing block, which (if any) of its successors should be merged into it. The compiler makes this decision based on a heuristic function that selects the most desirable next block, described in more detail in the following sections. To merge blocks the compiler performs if-conversion where necessary, which replaces branches with predication. If the merged block has a side entrance, the compiler applies head or tail duplication to ensure that the block has a single entry point. The compiler performs scalar optimizations on the resulting block and ensures it meets the architectural constraints on block size, register reads and writes, loads, and stores. If the block exceeds these constraints, the compiler discards the merge and chooses another next block, if one exists.

To determine the next block to merge the compiler considers all immediate successors of the block to be enlarged. For each block the compiler computes the value of a set of features, where features are properties of the successor block, the predecessor block, or the combination of the two. Examples of features include branch probability, block size, and dependence height; a full list is given in Section 7.4. Feature values are combined using an arithmetic heuristic function to produce a priority value, and the block with the highest priority value is merged first. Priority functions consist of the operators defined in Table 7.1.

The compiler uses hierarchical heuristic functions to express priority

<table>
<tr><th colspan="2">Hierarchy</th></tr>
<tr><td>(R_1, R_2)</td><td>R1 _ R2</td></tr>
<tr><th colspan="2">Real Operators</th></tr>
<tr><td>Function</td><td>Representation</td></tr>
<tr><td>$-R_1$</td><td>-R1</td></tr>
<tr><td>$R_1 + R_2$</td><td>R1 + R2</td></tr>
<tr><td>$R_1 - R_2$</td><td>R1 - R2</td></tr>
<tr><td>$R_1 \cdot R_2$</td><td>R1 * R2</td></tr>
<tr><td>R_1/R_2</td><td>R1 / R2</td></tr>
<tr><td>$R_1^{R_2}$</td><td>R1 ** R2</td></tr>
<tr><td>$\begin{cases} R_1 & \text{if } B_1 \\ R_2 & \text{if } \neg B_1 \end{cases}$</td><td>B1? R1 : R2</td></tr>
<tr><td>$\begin{cases} R_1 \cdot R2 & \text{if } B_1 \\ R_2 & \text{if } \neg B_1 \end{cases}$</td><td>B1? R1 @ R2</td></tr>
<tr><th colspan="2">Boolean Operators</th></tr>
<tr><td>$R_1 < R_2$</td><td>R1 < R2</td></tr>
<tr><td>$R_1 < R_2$</td><td>R1 > R2</td></tr>
<tr><td>$R_1 = R_2$</td><td>R1 == R2</td></tr>
<tr><td>$\neg B_1$</td><td>~R1</td></tr>
<tr><td>$B_1 \wedge B_2$</td><td>R1 && R2</td></tr>
<tr><td>$B_1 \vee B_2$</td><td>R1 || R2</td></tr>
</table>

Table 7.1: Operators used to construct cost functions.

```
prob _ 1/size
```

Figure 7.1: An example cost function that selects first by branch probability, then by the inverse of block size.

within a heuristic. A hierarchical heuristic consists of a sequence of arithmetic heuristics. To evaluate the priority of two blocks, the compiler begins with the highest-level arithmetic heuristic; if this value is equal for two blocks, it looks to the next arithmetic heuristic. This approach allows the heuristic writer to define tie breakers. For example, the policy in Figure 7.1 first selects based on branch probability and then based on the inverse of block size. In certain circumstances the compiler may decide not to continue merging blocks. To express this condition we define a zero cost element. Blocks with priority below zero will not be merged.

7.2 Default Block Formation Heuristic

To compare the various heuristics, we implement a baseline heuristic that corresponds to the "breadth-first" heuristic described in Chapter 4. The baseline heuristic merges blocks according to their order in a topological sort. This ordering ensures that no basic block will be merged before the compiler attempts to merge its predecessors. By imposing that condition the compiler limits unnecessary tail duplication. A further optimization performed by the baseline heuristic is to always perform loop unrolling or peeling first, before merging non-loop code. Because loops tend to be frequently executed this heuristic generally improves performance.

Because the heuristic function is critical to performance, we explore other possible functions to find better strategies. The following sections detail the considerations of block formation, the features implemented to make block

formation decisions, and the machine learning approach we used to explore the policy space. We also examine the performance of various policies with respect to the number of participating cores to determine if the configuration significantly changes the best policy.

7.3 Block Formation Heuristics

To determine which merges are most profitable, the compiler must consider several factors. This section overviews the tradeoffs and performance considerations that the compiler must weigh when determining the best merge. To appropriately balance these concerns, the compiler writer must identify features that measure each attribute, then develop a policy that uses those features to select the best blocks. Because TFlex can be configured with many core counts, we discuss the effect of core count on each consideration where applicable.

Register communication vs. Fanout: Values produced in one block and consumed in a successor block are transmitted through a global register file, whereas values produced and consumed within the same block use more efficient direct instruction communication. Thus, merging a producer and consumer block may benefit both latency and power consumption. If a value has many consumers within a given block, however, it requires fanout instructions, which may offset the benefits of merging. Composing cores on large substrates changes this tradeoff because it increases the communication overhead between blocks, making it more desirable for the compiler to merge producers

and consumers.

Prediction vs. Predication: By converting hard-to-predict control dependences into data dependences, predication can improve prediction accuracy. Merging a block may therefore be beneficial if the branch is hard to predict at runtime. This capability is a double-edged sword, however, because converting *predictable* control flow to data flow may delay resolution of the control decision, sacrifice branch correlation information, and introduce a data dependence where a branch predictor would be free to speculate. Unfortunately, while profile information suggests the frequency with which a branch will be taken, it does not necessarily indicate how *predictable* that branch will be. On larger configurations, branch predictability is generally more important than on smaller configurations, because the opportunity cost of branch misprediction is greater. A larger processor's instruction window may go largely unused unless the branch predictor is highly accurate.

Code bloat: The compiler helps determine how effectively the instruction cache is used. If the compiler does not fill a block then the assembler pads it with NOPs, reducing the effective instruction cache capacity. Likewise, instructions with non-matching predicates may needlessly consume space in the instruction cache. Optimizations such as loop unrolling, function inlining, and tail duplication increase instruction cache pressure via code duplication. On a composable processor, these considerations may be particularly important for small configurations with a more limited instruction cache capacity.

Useful instructions: The compiler is instrumental in forming a large effective instruction window. The effective instruction window includes only *useful* in-flight instructions. Useful instructions exclude NOPs, mis-speculated instructions, and instructions with non-matching predicates. The compiler may increase the effective window size by filling blocks with instructions, including predicated instructions, but the majority of these instructions must be useful at runtime for the compiler to be effective.

With accurate and consistent profiles the compiler can use a “narrow” strategy that merges blocks on hot paths to maximize useful instructions. In exchange, however, the compiler increases code bloat due to tail duplication to avoid side entries, and the compiler sacrifices opportunities for control speculation at runtime. Alternatively, the compiler can choose a “wide” strategy that includes multiple control paths, but ensures that some of those instructions will be useless.

Composability affects this tradeoff because a small configuration limits its execution resources, such as functional units and instruction window slots, and therefore the compiler must ensure that they only perform the most critical work. Small configurations will not tolerate useless instructions. Larger configurations, however, have ample functional units and instruction window space, so these resources can more readily be used to speculate. The additional resources tolerate some useless work if it does not interfere with useful and correctly speculated work. With ample resources, this strategy may improve performance, though it may decrease power efficiency.

Parallelism: On a substrate with two or more cores, multiple blocks can execute in parallel on different cores. If dependences exist between these blocks, however, then these parallel resources may be wasted. Heuristics that discourage merging independent work into a block may therefore benefit larger configurations. This criteria varies noticeably with the configuration because very limited parallelism is available on a single core.

Code Structure: When applying incremental block formation, the order in which the compiler considers blocks for merging can change the outcome of the merge, even if all of the same blocks are eventually selected. Because the compiler applies tail duplication immediately upon finding a side entrance, it preemptively changes the code, even if the side entrance is merged. Thus, if the compiler selects a merge point before selecting all paths leading to the merge point, then the merge is tail-duplicated, *even* if the other paths are eventually selected. The default topological sort naturally prevents this occurrence. Heuristic approaches, however, must account for this condition by down-ranking merge points, if there are other paths leading to that merge.

7.4 Implemented Features

We implemented 63 features, each of which provides information regarding one or more of the tradeoffs described above. During block formation, the compiler computes the value of each feature used in a heuristic function. The implemented features are a combination of boolean and real-valued measurements. Table 7.2 contains the boolean features and Table 7.3 and Table 7.4

Abbreviation	Description
cd	S is control dependent on P
dom	P dominates S
looppreheader	P precedes a loop header
pd	S post-dominates P
pil	P is in an innermost loop
ploopheader	P is a loop header
sameiloop	P and S belong to the same innermost loop
scd	S is control dependent on another successor of P
sibe	S contains a loop back edge
sil	S is in an innermost loop
sl	P and S belong to the same loop nest
slh	P and S belong to the same loop
sloopheader	S is a loop header
spd	S post-dominates another successor of P
sswrs	S writes a value read by its successor
std	S descends from another successor of P
td	Merging S into P requires tail duplication
wrblh	S writes registers read by its loop header

Table 7.2: Boolean-valued features

Abbreviation	Description
cdh	Maximum dependence height of the combined block
cfgheight	Height of S in the control flow graph
crr	Reads common to P and S
crw	Number of registers read by P also written by S
cwr	Number of registers written by P also read by S
cww	Writes common to predecessor and successor
dhr	Ratio of P dependence height to combined
loopsize	Size of loop containing P, in blocks
lsi	Size of loop containing P, in instructions
npb	Number of branches in P
npl	Number of loads in P
nppreds	Number of P's predecessors
npreads	Number of register reads in P
npreddoms	Number of blocks dominated by P
nps	Number of stores in P
npsuccs	Number of P's successors
npwrites	Number of register writes in P
nsam	Number of successors if P and S are combined
nsb	Number of branches in S
nsdomsinloop	Number of blocks dominated by S within the same loop
nsl	Number of loads in S
nspreds	Number of S's predecessors
nsreads	Number of register reads in S
nss	Number of stores in S
nssuccs	Number of S's successors
nsuccdoms	Number of blocks dominated by S
nswrites	Number of register writes in S
nw	New writes introduced by merging

Table 7.3: Real-valued features

Abbreviation	**Description**
overhead	Estimate of overhead instructions: null stores and writes
pdh	Maximum dependence height of P
pfan	Number of fanout instructions in P
phd	Maximum dependence height in P of any predicate leading to S
phsb	Size in blocks of the region dominated by P
phsi	Size in instructions of the region dominated by P
prob	Probability of taking branch from P to S
psize	Size in instructions of P
ptag	Rank of P in a topological sort of the control flow graph
sdh	Maximum dependence height of S
sfan	Number of fanout instructions in S
shps	Number of blocks descending from S that are dominated by P
shsb	Size in blocks of the region dominated by S
shsi	Size in instructions of the region dominated by S
size	Size in instructions of S
stag	Rank of S in a topological sort of the control flow graph
usize	Estimate of instructions in P that would be useless if S executes

Table 7.4: Real-valued features, continued

contain the real-valued features.

Each feature is computed for a pair of blocks: the predecessor P, which is the block currently being enlarged, and the successor S, which is the target being considered for merging. The features of S differ for each successor, while the features of P remain constant until an additional block is merged. Some features consider the combination of P and S. For example, the combined dependence height, `cdh`, represents the longest data dependence path through P and S. While the compiler could merge the blocks in scratch space to determine a precise value for such features, that approach would likely be too time-consuming. The compiler therefore estimates such feature values without actually merging blocks. Still other features analyze blocks other than P and S. For example, `looppreheader` examines the other successors of P to determine if P precedes a loop. This feature can be used to determine if the compiler should delay merging non-loop blocks in favor of peeling a succeeding loop.

7.5 Machine Learning Results

The compiler can take advantage of the flexibility provided by variable-size blocks to improve performance. We use machine learning to find good block formation policies. In addition to learning good policies, we also try to understand why these policies are effective. Furthermore, we describe the role of composability: can the compiler do better if it knows what the hardware will look like at run time? We employ machine learning to rapidly search the space

of heuristics. We select a technique based on genetic programming because the results are human-interpretable, unlike many other learning methods.

7.5.1 Learning Methodology

We use meta-optimization, proposed by Stephenson et al. [72], to learn EDGE block formation heuristics. The machine learning system represents heuristic functions as N-ary trees of operators, with a mix of code features and constants at the leaves. Each generation consists of 300 heuristics, with the first generation initialized with a combination of hand-written and randomly generated heuristics. We evaluate the performance of each heuristic in a generation and use these scores to determine which organisms will survive to the next generation. We additionally crossover and mutate organisms randomly to introduce variation into the population and probabilistically discover higher performance heuristics [39].

Because simulated execution of entire benchmarks—particularly a large and complex suite such as SPEC CPU—is extremely time intensive, we instead use a group of microbenchmarks for training and later apply the learned heuristics to the full benchmarks. We draw a suite of microbenchmarks from SPEC kernels, signal processing applications, and high-performance computing kernels. The training runs target a particular configuration of composed cores, thus learning heuristics specialized for 1-core, 8-core, and 32-core configurations. We train specifically for these configurations to isolate differences in compiler policy needed for different configurations.

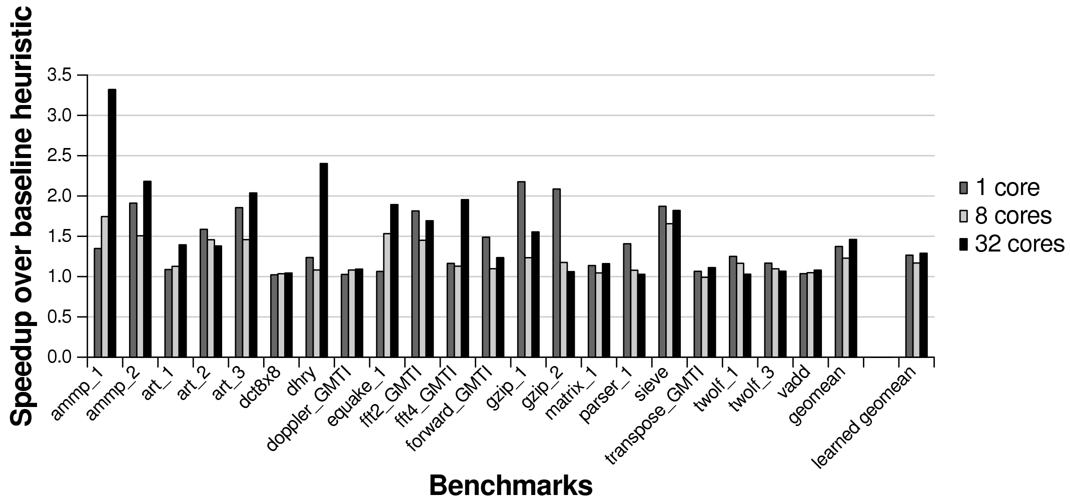


Figure 7.2: Speedup of learned heuristics on microbenchmarks. Each bar is normalized to the performance of the baseline heuristic running on the same number of cores as the learned heuristic.

We use a two-phase training methodology. First, we specialize heuristics for each microbenchmark. We seed the intial population with our best hand-written heuristic for that benchmark, and fill out the population with 299 randomly-generated heuristics. We then evolve this population over 50 generations, using performance on that microbenchmark as the fitness function.

Figure 7.2 shows the speedup compared to the baseline heuristic for each benchmark using this technique. Each bar in the figure represents a different heuristic, which has been specialized for that particular benchmark and that particular core count. Each bar is normalized to the performance of the baseline heuristic running on the trained number of cores. For example, ammp_1/32-cores is the ratio of cycles to complete ammp_1 using the baseline

heuristic and and 32 cores to the cycles to complete ammp_1 using the heuristic trained on ammp_1 with a 32-core configuration. Thus, the bars show only the speedup of learning over the baseline, rather than the absolute performance gained by increasing the core count.

To describe the nature of these optimizations, two particular points are worth discussing. The gzip_1 benchmark trained for 1-core execution achieves a 2.2x speedup over baseline, the highest of any 1-core speedup. This heuristic is particularly interesting because it effectively disables predication: the only merges the compiler performs are unconditional branches. For a single core, this makes sense, because predication overly saturates the machine resources. With variable-size blocks implemented, the processor can speculate several small blocks in advance, which negates the performance cost of small blocks.

The ammp_1 benchmark trained for 32-core execution shows a much different trend. This benchmark achieves a striking 3.3x speedup over baseline. The learned heuristic optimizes the benchmarks such that key inner loops are unrolled and peeled optimally. At the end of the main loop, however, the heuristic leaves a single basic block containing an induction variable update unpredicated. While this decision costs block size, it dramatically improves performance by allowing multiple speculative induction variable updates in-flight at once, without waiting on predicates within the loop.

Once each benchmark has been trained individually, we next train for a general heuristic across all benchmarks. We initialize the population with the best learned heuristic for each benchmark as well as the best overall

hand-written heuristic. We augment this population with an additional 277 randomly-generated heuristics to complete a 300 member population. During general learning, the system uses each heuristic to compile all the benchmarks. It computes the geometric mean speedup over the baseline heuristic across the benchmark suite, and uses that value as the fitness for the heuristic. Once all heuristics are evaluated, the population evolves and the procedure repeats.

We evolve this population for an additional 12 generations to arrive at an optimized heuristic for a particular core count. At this point the best heuristic had largely stabilized, yielding few improvements from one generation to the next. The "learned geomean" bar of Figure 7.2 shows the speedup using this process. Somewhat unexpectedly, the heuristics for 1 and 8-core configurations was only mildly changed by the learning process. This result stands in contrast to previous work using this method, which found that frequently the initial, random population saw speedups over the built-in heuristic [72]. In our case, however, we hand-tuned the heuristic on the same set of benchmarks used for machine-learning, which undoubtedly provides a human advantage not shared by the previous work.

An avenue we did not explore was to separately train benchmarks with integer versus floating-point characteristics. Training based on this classification would likely result in greater overall speedups, as the two categories of programs have significantly different characteristics. Despite this potential, the goal of this experiment was to learn heuristics that could automatically handle code of any type. We therefore train over the entire set of microbench-

<table>
<tr><th>Config</th><th>Speedup</th><th>Heuristic</th></tr>
<tr><td>Hand-1</td><td>1.22</td><td>(npsuccs<2)</td></tr>
<tr><td>Learn-1</td><td>1.27</td><td>((sswrs)?((2.0)/(shsi))@(npsuccs))==(1.0)</td></tr>
<tr><td>Hand-8</td><td>1.06</td><td>sloopheader*std*prob
_std*(prob+0.01)*sl
_prob*sl</td></tr>
<tr><td>Learn-8</td><td>1.16</td><td>((sloopheader)*(std))*(((sloopheader)*(std))*(prob))
_((std)*((prob)+(0.01)))*(sl)
_(npwrites)*(cd)</td></tr>
<tr><td>Hand-32</td><td>1.11</td><td>sloopheader*std*prob
_std*(prob+0.01)*sl
_prob*sl</td></tr>
<tr><td>Learn-32</td><td>1.29</td><td>(((((1.0)-(84.317566))+(phd))<((nsreads)+(nss)))&&(((sl)||(sibe))
&&((nsuccdoms)<(cfgheight))))?((~(spd))?(((~(spd))?((shsb)
+((pdh)**(prob))):(((49.410812)+((pdh)**((~(spd))*((prob)+(loopsize)))))
-((~(spd))?((shsb)+((sswrs)?(16.669044)@(nsreads))):((loopsize)-(shsi)))))
+((sswrs)?(16.669044)@(loopsize))):
((loopsize)-(shsi))):(((std)&&(prob))?(prob):(0.0))_(cww)/(lsi)</td></tr>
</table>

Table 7.5: Heuristics with the best geometric mean speedup over baseline, both hand-written and machine-learned.

marks rather than hand-picked subsets.

7.5.2 Learned Heuristics Discussion

Table 7.5 shows the best heuristics discovered, in terms of geometric mean speedup, both by hand and by machine learning. This table shows a few interesting outcomes of the machine learning process. The learned heuristics for 1 and 8 cores are, as mentioned above, quite similar to the hand-learned heuristic. In fact, the hand heuristic remained near the top of these populations for the entire training process. The speedups shown indicate that the learning system was able to improve performance only slightly over the hand heuristic. The 32-core learned heuristic performs significantly better than the hand heuristic, but demonstrates a fundamental problem with machine learning as a tool for gaining insight: the resulting function is almost completely

uninterpretable.

We found that the best overall heuristic function for a 1-core configuration differed significantly from the best heuristics for 8- and 32-core configurations. The 1-core heuristic essentially never uses predication, as it only merges a block when the branch is unconditional. The 8-core heuristic behaves very similarly to the baseline heuristic described in Section 7.2, in that it generally favors "wide" predication and limited tail duplication. The 32-core heuristic demonstrates one of the weaknesses of machine learning as a tool for gaining insight: the function generated by the learner is too complex for us to extract a meaningful policy.

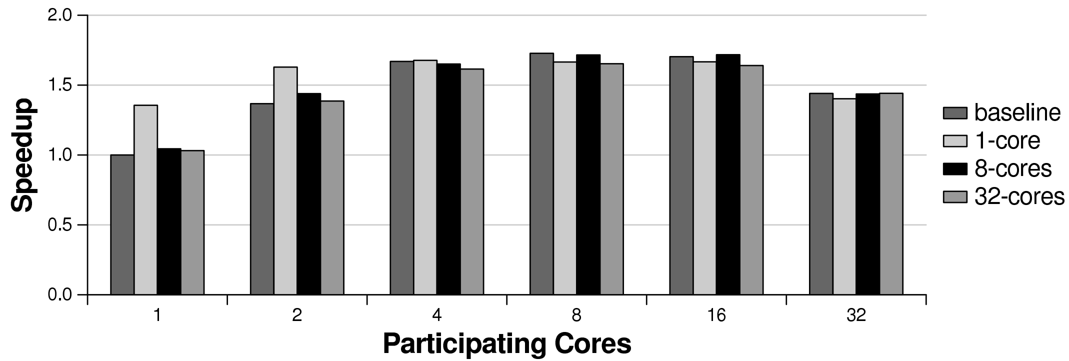
The 1-core heuristic is significant because it avoids predication, which is an important feature of an EDGE ISA, and differs in this respect from the best heuristics for larger topologies. A one-core configuration lacks the resources to tolerate more widely predicated code as mispredicated instructions occupy scarce issue and execution slots that could be put to better use. At the same time, the 128-instruction window provided by a single core is small enough to be effectively used by the branch predictor. Larger configurations, by contrast, suffer from geometrically increasing misprediction rates without the use of predication.

Furthermore, the 1-core heuristic is unlikely to be useful without support for variable-size blocks. By avoiding predication, the compiler creates many small blocks, because it is rare to chain more than a few unconditional branches together. Variable-size blocks make such a heuristic possible by re-

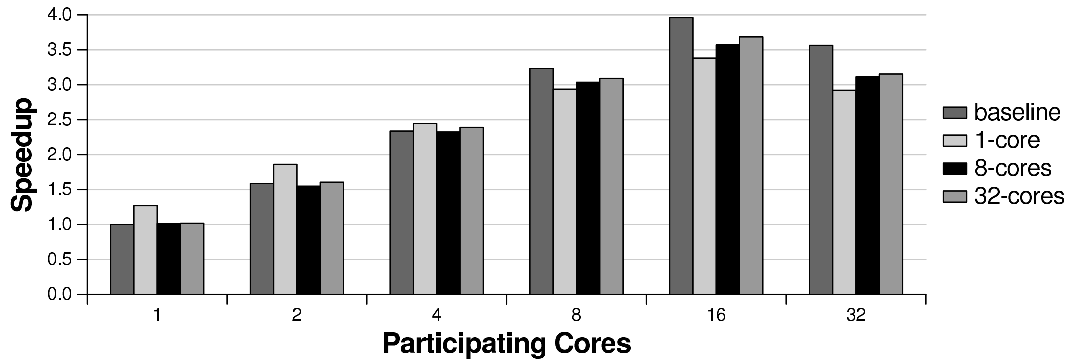
ducing the penalty for executing small blocks. As the core count increases, however, the likelihood of misprediction reduces the effectiveness of variable-size blocks in this scenario. Therefore the benefits of this heuristic are likely confined to relatively small configurations.

While composability remains an effective technique for improving performance of the same binary executable by aggregating cores, these results suggest that the performance of smaller configurations can be significantly improved. Chapter 6 showed that by providing variable-size block support, integer benchmarks immediately benefit from executing smaller blocks on smaller configurations. This combination could be quite effective in certain domains with area and power constraints. This chapter has shown how to extend that solution with compiler support, by developing heuristics tailored to particular configurations.

Because the performance and heuristics for each benchmark vary widely with the core count, a compiler that is aware of the configuration could realize significant gains in performance. Such a capability could be useful for constrained systems where area or power are limited, where the compiler knows that it must generate code for a small system. On more complex systems, one could envision a dynamic optimizing compiler, which could receive feedback from the processor about the composed topology. The compiler could then adjust its heuristics to provide better-suited code to the underlying microarchitecture.



(a) SPECINT



(b) SPECFP

Figure 7.3: Speedup of heuristics learned for 1-, 8-, and 32-cores over baseline heuristic on one core.

7.5.3 SPEC CPU Results

We use the learned heuristics for 1-, 8-, and 32-core configurations to compile the SPEC benchmarks and measure their performance on all core counts. Figure 7.3 shows the performance of these heuristics at all core counts, normalized to the performance of a single core with the compiler's baseline heuristic.

The most striking success of learning is the 1-core learned heuristic,

which outperforms the baseline by 35% on SPECINT and 27% on SPECFP when running on a single core. This heuristic performs well in 2-core configurations as well, but tapers off at four or more cores, performing roughly equivalently on INT, and as much as 22% below the baseline on FP.

Heuristics for 8- and 32-core topologies are mixed: they achieve approximately the same performance as the baseline on SPECINT, but fall short on SPECFP, particularly with eight or more cores. Compared to the eight-core heuristic, the baseline achieves 6% higher performance at eight cores and 14% at 32-cores. This result highlights the performance fragility of learned heuristics. While performance gains were significant on microbenchmarks, the results do not scale as well to full programs.

7.6 Summary

By using machine learning techniques we have learned a fundamental property of the architecture: that predication limits performance when machine resources are limited. While this insight by itself has been noted for other architectures, such as VLIW, it is novel in the context of block-atomic EDGE architectures. Variable-size blocks are of course important to this result, because without variable-size blocks the opportunity cost of not using predication to fill blocks is simply too high. By changing the relative costs of executing blocks, we have changed the compiler's best heuristics as well.

While we have successfully found interesting heuristics for a set of benchmarks on which the compiler trained, more general heuristics remain

elusive. While the small core-count policy performs generally on SPECINT, no policy significantly outperforms the baseline on SPECFP, or in most configurations of SPECINT. While this perhaps validates the baseline as a reasonable policy, it gives little insight into whether generally better policies could be found. However, overspecialization is a general problem with machine learning and in retrospect it is not surprising that we find that problem as well in this work.

Since we did not appear to learn a heuristic that generally achieves the best performing points for SPECINT and SPECFP, it may be worthwhile to question the limitations of this system. It is possible that a dynamic system that can monitor performance counters could perform much better. While such a system is beyond the scope of this work, there are many other advantages to a dynamic system that would make it worthwhile for this class of architectures.

Chapter 8

Conclusions

The computing industry is at an inflection point. The primary driver of single-CPU performance—increasing frequency by building deeper pipelines—came to an end with Intel's cancellation of the NetBurst microarchitecture. Conventional techniques for improving single-threaded performance using out-of-order superscalar techniques have reached a point of diminishing returns due to complexity. Even improvements in frequency and power consumption due to Dennard scaling appear to have dramatically slowed or stopped. Demand for increased performance, however, continues unabated. As a community we need innovative systems solutions to meet this continued demand even as conventional approaches come to an end.

EDGE architectures attempt to address this problem by organizing instructions in large blocks, which use dataflow internally. This organization enables power-efficient out-of-order execution using a distributed microarchitecture. Furthermore, the design allows extremely high levels of parallelism to be exploited, by increasing the ratio of functional and arithmetic units to control logic. The TRIPS prototype processor puts these hypotheses to the test with a fully-realized implementation of an EDGE ISA. To achieve these the-

oretical advances however, EDGE architectures require compiler support for new optimizations. Key among these compiler optimizations are block formation, which has been the focus of this dissertation, and instruction scheduling, which has been research elsewhere. We believe that these compiler tasks are more tractable than the approaches needed for alternative designs. Existing chip multiprocessors, for example, seem to require automatic parallelization or a programming model that makes manual parallelization tractable. It is still an open question what combination of hardware and software techniques, if any, will succeeded in delivering performance improvements through parallelism.

As this dissertation has shown, the performance of EDGE architectures rests on the ability of the compiler to form blocks that contain ample amounts of parallel work. We have presented an algorithm that achieves this goal under many circumstances. Inspecting the performance of the compiler on realistic benchmarks, however, reveals that we are a long way from a general ability to construct full blocks that take advantage of an EDGE machine's immense parallel resources. The complexity of control flow in modern programs frequently makes forming large blocks difficult or impossible. Future EDGE microarchitectures should support such modifications as we propose for variable-size blocks to avoid unnecessary performance penalties for block-atomic execution.

To conclude this dissertation, we review the contributions of the dissertation towards understanding block-atomic compilation, and suggest future avenues of research. While many of the early, fundamental issues regarding EDGE compilation have been addressed by this dissertation (and others), there

are still significant problems to be solved to make EDGE architectures achieve extremely high performance.

8.1 Dissertation Contributions

This dissertation has considered the roles of the compiler, architecture, and microarchitecture in enabling efficient block-atomic execution. We began with the question "Can a compiler form large blocks for an EDGE architecture?" While frequently the answer this question can be answered affirmatively, we have dissected the scenarios in which it is not possible. This dissertation has presented compiler techniques for achieving large blocks, analyzed the performance of those blocks, and suggested a combination of architectural optimizations and compiler heuristics to further improve performance. Together these techniques make block-atomic architectures a more compelling platform for future research and development.

The first contribution of the dissertation is an iterative, incremental algorithm for EDGE block formation that incorporates scalar optimizations, loop unrolling and loop peeling into a single-pass framework. This algorithm is significant for several reasons. Iterative hyperblock formation enables robust compilation for a structurally-constrained block atomic architecture. Before this iterative approach, we had considered approaches based on ideal block formation followed by block-splitting. For the reasons described in prior chapters, this approach was not tractable. Furthermore, iterative hyperblock formation enables improved filling of EDGE blocks even when optimization opportunities

are available. Because merging blocks enables new optimization opportunities, it is difficult to say statically which blocks should be merged. By iterating, we reach the best fixed point. Finally, we unify loop unrolling and peeling under the umbrella of head duplication, which allows it to fold cleanly into a unified block formation algorithm.

Using this block formation algorithm we compare performance of compiled code against Intel's Core 2 processor to determine the relative strength of the TRIPS processor. This performance comparison demonstrated that TRIPS could compete with a commercial processor on certain applications that were highly parallel, data-intensive algorithms, particularly when we wrote assembly code by hand. However we could not confirm the hypothesis that TRIPS could generally outperform superscalar processors. We discovered that particular design decisions and compiler difficulties particularly impacted the performance of TRIPS. Strikingly, the instruction cache miss rate was significantly higher on TRIPS than that of the Intel processor, indicating a combination of insufficient capacity (although the structure was 2.5 times larger than the Intel cache), or an inability to form large blocks.

A detailed analysis of the compiler's capabilities show that block formation is a fundamental problem for a compiler due to control flow constraints. Function calls were the most significant source of very small (1–16 instruction) blocks, despite extremely aggressive inlining. While we attempted to implement techniques that would mitigate this problem by predicating call sites or epilogues, the predication overhead severely reduced performance despite a

modest improvement in block size. Furthermore, reducing the maximum block size only improves average efficiency by a modest amount, due to the dual-peaked nature of the block-size distribution. We concluded that to truly solve the performance problems solved by under-full blocks, the microarchitecture should support variable-size blocks.

To solve this problem we propose an instruction set revision that, without changing the fundamentals of an EDGE architecture, enables efficient implementation of variable size blocks. With minor instruction set modifications we are able to support eight-instruction variability, which achieves over 95% code size efficiency. We implement this capability in a composable microarchitecture, TFlex, that shares similar characteristics to TRIPS but allows flexible allocation of core resources to threads. The variable-size blocks capability translates into significantly increased performance in resource-constrained environments such as a single-core TFlex configuration.

With this increased from block size constraints we consider a range of policy options for block formation. While there are several criteria which make sense for block formation it is not immediately clear how to combine them for the best effect, so we explore the space of policy options using a genetic algorithm. We find that for small configurations the policy options are markedly different from large configurations. Small configurations appear to benefit from reduced use of predication. With variable-size blocks implemented, the smaller blocks that result from not predicating are not a handicap, and with limited machine resources it is more important not to do wasted, predicated-out work.

For large topologies it is more important to construct a larger window and resource constraints are less severe, thus predication is more beneficial.

With this combination of compiler techniques and architectural optimizations we have improved the ability of an EDGE system to efficiently execute real-world code. We have combined compiler optimizations—both mechanisms and policies—with a careful evaluation of the compiler's capabilities, and what capabilities it could potentially have. By redesigning an aspect of the hardware (fixed-size blocks) to allow the compiler slack we have not only improved the performance of the baseline system, but discovered a novel opportunity for optimization by removing a significant constraint (rigid adherence to block size) from the compiler.

8.2 Future Directions

While this dissertation has contributed a good deal towards high performance compilation for block-atomic architectures, there is more research to be done. Additional engineering could improve the compiler's block formation abilities (although there may be unforeseen problems along the way). Evidence that the best block formation policies differ based on the microarchitecture indicates that dynamic techniques could be a promising research area in the future. We outline these areas of future work here.

Because function calls, and by extension inlining, significantly reduce the compiler's ability to form large blocks, improving the compiler's ability to remove function call boundaries could significantly improve block sizes. While

Scale uses a fairly typical overall compiler structure where inlining is performed early on, a more flexible, whole-program optimization approach could be applied that could allow traditional optimizations across call boundaries, as well as block formation. While whole-program optimization techniques have existed for some time, they have been applied infrequently in common compilers because of the time overhead. However, some limited version of whole-program compilation, perhaps restricted to block formation only, could prove helpful in improving performance without unnecessarily increasing compile time.

Because calls into libraries and cross-module function calls account for a sizable fraction of the problematic calls, post-link optimization may be worthwhile for a block-atomic architecture. Post-link optimization has become increasingly common in production compilers, so it is clearly feasible in some contexts. For EDGE block formation to be feasible post-link, the linked bytecodes must be at a sufficiently high level of abstraction for block splitting and register allocation to operate in a relatively natural manner. It would be rather difficult to extract a useful representation from an already optimized TRIPS binary, for example, due to the extensive use of block merging and predicate optimizations.

Dynamic compilation using a just-in-time compiler or virtual machine could significantly improve performance, both by providing the ability to optimize across boundaries and by providing more extensive profiling information for speculative optimizations. Dynamic optimization of an intermediate representation immediately gives the system the benefits of a restructured static

compiler or of post-link optimization, as well as offering new opportunities for speculative optimization, re-optimization, and profile-guided optimization. Profiling could be particularly useful; while our heuristic experiments indicated only mild biases in favor of frequently executed paths, the compiler could combine block formation with speculative optimizations frequently used by dynamic optimizers to achieve increased performance.

As presented in Chapter 7 different programs have markedly different behavior. While this dissertation has presented some techniques for automatically discovering and optimizing for this behavior, more could be done. In a dynamic system, the processor could detect programs or phases of programs that require more or fewer parallel resources and allocate a slice of the processor appropriately. Combined with the variable-size blocks capability of Chapter 6, such a processor could achieve high efficiency and high performance by executing control-intensive code on fewer cores, while allocating more cores to data-intensive, highly parallel code.

A dynamic system with greater awareness of the specialized needs of programs should be capable of taking advantage of these diverse characteristics. As we have shown, smaller topologies benefit from different heuristics than larger topologies, with predication being less desirable as the number of participating cores decreases. A cooperative hardware/software system could notify a dynamic optimizer when the core allocation of a program changes. That optimizer could either re-optimize the program to execute more quickly on the new topology, or could invoke a different version of the binary that was

precompiled. While this system would have increased complexity, the benefits in terms of performance and power efficiency could be significant.

8.3 Final Thoughts

As parallelism becomes increasingly important to computer performance, unconventional architectures will increasingly gain acceptance in the broader community. EDGE architectures, and block-atomic processors more generally, may find favor due to their ability to achieve out-of-order execution with a large instruction window in a relatively efficient manner. Combined with the ability to relatively easily compose large numbers of EDGE cores together to improve performance, these characteristics may make EDGE particularly attractive going forward. However, compiler support for such architectures is still in its infancy, and it remains to be seen whether such a design can exceed superscalar capabilities and scale to future technologies.

Because EDGE requires a radically new ISA, its widespread acceptance dependends on social as well as technical factors. Binary compatibility has historically hindered market acceptance of new ISAs. The mass market for desktop computers running the x86 instruction set entrenched that ISA in many domains. Despite this history, several trends may ease the path to market for new ISAs. The rise of mobile phones, tablets, and cloud computing reduces the dependence of consumers on applications pre-compiled for a particular architecture. Furthermore, the performance of compiler technology for binary translation has improved and may be used to provide compatibility for

legacy software. Even if binary compatibility remains important, however, the rise of general-purpose computation on graphics processors shows that certain application domains that demand high performance may be willing to shift to a new architecture, if the performance gains are sufficient.

This dissertation has provided evaluation, insight, and algorithms to improve the performance and efficiency of such architectures. By advancing the state of the art in this area, the dissertation perhaps moves a step closer to more widespread adoption of some of the ideas of block-atomic processing. Regardless of whether such processors are ever taken up by the market, there is scientific value in understanding their capabilities and limitations. This work provides such an evaluation, and enables future compilers to build on what has been done to create better solutions. By providing a detailed exploration of the challenges faced in compiling for a block atomic architecture and solutions to at least a few of those challenges, we show that such designs have significant potential for power-efficient performance.

Bibliography

[1] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus IPC: the end of the road for conventional microarchitectures. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 248–259, New York, NY, USA, 2000. ACM.

[2] A. Aiken and A. Nicolau. A realistic resource-constrained software pipelining algorithm. In *Workshop on Languages and Compilers for Parallel Computing*, Irvine, CA, Aug. 1990.

[3] A. Aiken, A. Nicolau, and S. Novack. Resource-constrained software pipelining. *IEEE Transactions on Parallel and Distributed Systems*, 6(12):1248–1270, 1995.

[4] J. R. Allen, K. Kennedy, C. Porterfield, and J. D. Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th Annual Symposium on Principles of Programming Languages*, pages 177–189, 1983.

[5] D. I. August. Hyperblock performance optimizations for ILP processors. Master's thesis, University of Illinois at Urbana-Champaign, 1993.

[6] D. I. August, W.-m. W. Hwu, and S. A. Mahlke. A framework for balancing control flow and predication. In *Proceedings of the 30th annual*

ACM/IEEE international symposium on Microarchitecture (MICRO '97), pages 92–103. IEEE Computer Society, 1997.

[7] D. I. August, W.-M. W. Hwu, and S. A. Mahlke. The partial reverse if-conversion framework for balancing control flow and predication. *International Journal of Parallel Programming*, 27(5):381–423, 1999.

[8] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt. The FORTRAN automatic coding system. In *IRE-AIEE-ACM '57 (Western): Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability*, pages 188–198, 1957.

[9] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood. Performance pathologies in hardware transactional memory. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 81–91, New York, NY, USA, 2007. ACM.

[10] D. G. Bradlee, S. J. Eggers, and R. R. Henry. Integrating register allocation and instruction scheduling for RISCs. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems (ASPLOS '91)*, pages 122–131. ACM, 1991.

[11] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, W. Yoder, and the

TRIPS team. Scaling to the end of silicon with EDGE architectures. *Computer*, 37(7):44–55, 2004.

[12] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 227–238, 2006.

[13] P. P. Chang and W. W. Hwu. Trace selection for compiling large C application programs to microcode. In *MICRO 21: Proceedings of the 21st annual workshop on Microprogramming and microarchitecture*, pages 21–29, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[14] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W.-m. W. Hwu. IMPACT: an architectural framework for multiple-instruction-issue processors. In *ISCA '91: Proceedings of the 18th annual international symposium on Computer architecture*, pages 266–275, New York, NY, USA, 1991. ACM.

[15] P. P. Chang, S. A. Mahlke, and W.-m. W. Hwu. Using profile information to assist classic code optimizations. *Software Practice and Experience*, 21(12):1301–1321, 1991.

[16] K. E. Coons, X. Chen, D. Burger, K. S. McKinley, and S. K. Kushwaha. A spatial path scheduling algorithm for EDGE architectures. In *Proceedings of the 12th international conference on Architectural support*

for programming languages and operating systems (ASPLOS '06), pages 129–140. ACM, 2006.

[17] K. E. Coons, B. Robatmili, M. E. Taylor, B. A. Maher, D. Burger, and K. S. McKinley. Feature selection and policy optimization for distributed instruction placement using reinforcement learning. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*, pages 32–42, 2008.

[18] K. Crozier. Structural and static analysis techniques for enhancing compiler support of predicated execution. Master's thesis, University of Illinois at Urbana-Champaign, 1999.

[19] J. Davidson and S. Jinturkar. An aggressive approach to loop unrolling. Technical Report CS-95-26, University of Virginia, 1995.

[20] J. W. Davidson and S. Jinturkar. Aggressive loop unrolling in a retargetable optimizing compiler. In *CC '96: Proceedings of the 6th International Conference on Compiler Construction*, pages 59–73, London, UK, 1996. Springer-Verlag.

[21] B. L. Deitrich and W.-M. W. Hwu. Speculative hedge: regulating compile-time speculation against profile variations. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 70–79, 1996.

[22] J. R. Diamond, B. Robatmili, S. W. Keckler, R. van de Geijn, K. Goto, and D. Burger. High performance dense linear algebra on a spatially distributed processor. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '08)*, pages 63–72, 2008.

[23] H. Esmaeilzadeh and D. Burger. Hierarchical control prediction: Support for aggressive predication. In *PESPMA '09: Proceedings of the 2nd Workshop on Parallel Execution of Sequential Programs on Multi-core Architectures*, pages 71–80, 2009.

[24] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 30(7):478–490, 1981.

[25] J. A. Fisher. Very long instruction word architectures and the ELI-512. In *ISCA '83: Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 140–150, 1983.

[26] J. A. Fisher, J. R. Ellis, J. C. Ruttenberg, and A. Nicolau. Parallel processing: A smart compiler and a dumb machine. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction*, pages 37–47, 1984.

[27] M. Gebhart, B. A. Maher, K. E. Coons, J. Diamond, P. Gratz, M. Marino, N. Ranganathan, B. Robatmili, A. Smith, J. Burrill, S. W. Keckler, D. Burger, and K. S. McKinley. An evaluation of the TRIPS computer

system. In *ASPLOS '09: Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1–12, 2009.

[28] J. R. Goodman and W.-C. Hsu. Code scheduling and register allocation in large basic blocks. In *ICS '88: Proceedings of the 2nd international conference on Supercomputing*, pages 442–452. ACM, 1988.

[29] E. Hao, P.-Y. Chang, M. Evers, and Y. N. Patt. Increasing the instruction fetch rate via block-structured instruction set architectures. In *MICRO-29: Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 191–200, 1996.

[30] W. Havanki, S. Banerjia, and T. Conte. Treegion scheduling for wide issue processors. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, page 266, 1998.

[31] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on Computer architecture (ISCA '93)*, pages 289–300. ACM, 1993.

[32] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.

[33] M. S. Hrishikesh, D. Burger, N. P. Jouppi, S. W. Keckler, K. I. Farkas, and P. Shivakumar. The optimal logic depth per pipeline stage is 6 to 8

fo4 inverter delays. In *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*, pages 14–24, Washington, DC, USA, 2002. IEEE Computer Society.

[34] W.-M. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: an effective technique for VLIW and superscalar compilation. *Journal of Supercomputing*, 7(1–2):229–248, 1993.

[35] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez. Core fusion: accommodating software diversity in chip multiprocessors. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 186–197, 2007.

[36] C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *ASPLOS '02: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 211–222, 2002.

[37] C. Kim, S. Sethumadhavan, M. S. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler. Composable lightweight processors. In *MICRO-40: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 381–394, 2007.

[38] A. J. KleinOsowski and D. J. Lilja. MinneSPEC: A new SPEC bench-

mark workload for simulation-based computer architecture research. *Computer Architecture Letters*, 1:7–11, 2002.

[39] J. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* The MIT Press, 1992.

[40] J. Lah and D. E. Atkins. Tree compaction of microprograms. *SIGMICRO Newsletter*, 14(4):23–33, 1983.

[41] M. Lam. Software pipelining: An effective scheduing technique for VLIW machines. In *ACM Conference on Programming Language Design and Implementation*, Atlanta, GA, June 1988.

[42] D. Lea. A memory allocator, 1996. http://gee.cs.oswego.edu/dl/html/malloc.html.

[43] H. Leather, E. Bonilla, and M. O'Boyle. Automatic feature generation for machine learning based optimizing compilation. In *CGO '09: Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 81–91, Washington, DC, USA, 2009. IEEE Computer Society.

[44] D. Li, B. Robatmili, S. Govindan, D. Burger, and S. Keckler. Hybrid operand communication for dataflow processors. In *PESPMA '09: Proceedings of the 2nd Workshop on Parallel Execution of Sequential Programs on Multi-core Architectures*, pages 61–70, 2009.

[45] B. A. Maher, A. Smith, D. Burger, and K. S. McKinley. Merging head and tail duplication for convergent hyperblock formation. In *MICRO-39:*

Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, pages 65–76, 2006.

[46] S. A. Mahlke. *Exploiting instruction level parallelism in the presence of conditional branches.* PhD thesis, University of Illinois at Urbana-Champaign, 1997.

[47] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *MICRO 25: Proceedings of the 25th annual international symposium on Microarchitecture*, pages 45–54, 1992.

[48] K. S. McKinley, J. Burrill, M. Bond, D. Burger, B. Maher, B. Robatmili, and A. Smith. The Scale compiler, 2007. http://ali-www.cs.umass.edu/-~scale/.

[49] S. W. Melvin and Y. N. Patt. Exploiting fine-grained parallelism through a combination of hardware and software techniques. In *ISCA '91: Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 287–296, 1991.

[50] S. W. Melvin and Y. N. Patt. Enhancing instruction scheduling with a block-structured ISA. *International Journal of Parallel Programming*, 23(3):221–243, 1995.

[51] S. W. Melvin, M. C. Shebanow, and Y. N. Patt. Hardware support for large atomic units in dynamically scheduled machines. In *MICRO-*

21: Proceedings of the 21st Annual Workshop on Microprogramming and Microarchitecture, pages 60–63, 1988.

[52] R. Nagarajan, S. K. Kushwaha, D. Burger, K. S. McKinley, C. Lin, and S. W. Keckler. Static placement, dynamic issue (SPDI) scheduling for EDGE architectures. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 74–84, Antibes Juan-les-Pins, France, Oct. 2004.

[53] R. Nagarajan, K. Sankaralingam, D. Burger, and S. W. Keckler. A design space evaluation of grid processor architectures. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture (MICRO '01)*, pages 40–51. IEEE Computer Society Press, 2001.

[54] N. Neelakantam, D. R. Ditzel, and C. Zilles. A real system evaluation of hardware atomicity for software speculation. In *ASPLOS '10: Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 29–38, 2010.

[55] N. Neelakantam, R. Rajwar, S. Srinivas, U. Srinivasan, and C. Zilles. Hardware atomicity for reliable software speculation. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 174–185, 2007.

[56] N. Nethercote, D. Burger, and K. S. Mckinley. Convergent compilation applied to loop unrolling. *Transactions on High-Performance Embedded Architectures and Compilers I*, pages 140–158, 2007.

[57] J. C. H. Park and M. Schlansker. On predicated execution. Technical Report HPL-91-58, HP Laboratories, 1991.

[58] B. R. Rau. Iterative modulo scheduling: an algorithm for software pipelining loops. In *Proceedings of the 27th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 63–74, 1994.

[59] B. Robatmili, K. Coons, D. Burger, and K. S. McKinley. Register bank assignment for spatially partitioned processors. pages 64–79, 2008.

[60] B. Robatmili, K. E. Coons, D. Burger, and K. S. McKinley. Strategies for mapping dataflow blocks to distributed hardware. In *MICRO-41: Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, pages 23–34, 2008.

[61] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *ISCA '03: Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 422–433, 2003.

[62] K. Sankaralingam, R. Nagarajan, R. McDonald, R. Desikan, S. Drolia, M. S. Govindan, P. Gratz, D. Gulati, H. Hanson, C. Kim, H. Liu, N. Ranganathan, S. Sethumadhavan, S. Sharif, P. Shivakumar, S. W. Keckler, and D. Burger. Distributed microarchitectural protocols in the trips prototype processor. In *MICRO 39: Proceedings of the 39th Annual*

IEEE/ACM International Symposium on Microarchitecture, pages 480–491, Washington, DC, USA, 2006. IEEE Computer Society.

[63] P. B. Schneck. Automatic recognition of vector and parallel operations in a higher level language. In *ACM '72: Proceedings of the ACM annual conference*, pages 772–779. ACM, 1972.

[64] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *PACT '01: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 3–14, 2001.

[65] L. T. Simpson. *Value-driven Redundancy Elimination.* PhD thesis, Rice University, 1996.

[66] A. Smith, J. Burrill, J. Gibson, B. Maher, N. Nethercote, B. Yoder, D. Burger, and K. S. McKinley. Compiling for EDGE architectures. In *International Symposium on Code Generation and Optimization*, 2006.

[67] A. Smith, R. Nagarajan, K. Sankaralingam, R. McDonald, D. Burger, S. W. Keckler, and K. S. McKinley. Dataflow predication. In *MICRO-39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 89–102, 2006.

[68] A. L. Smith. *Explicit Data Graph Compilation.* PhD thesis, The University of Texas at Austin, 2009.

[69] J. E. Smith and A. R. Pleszkun. Implementation of precise interrupts in pipelined processors. In *Proceedings of the 12th annual international symposium on Computer architecture (ISCA '85)*, pages 36–44. IEEE Computer Society Press, 1985.

[70] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 414–425, 1995.

[71] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. *SIGARCH Computer Architecture News*, 28(2):1–12, 2000.

[72] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly. Meta optimization: improving compiler heuristics with machine learning. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 77–90, 2003.

[73] R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, 58(1):267–288, 1996.

[74] S. Triantafyllis, M. J. Bridges, E. Raman, G. Ottoni, and D. I. August. A framework for unrestricted whole-program optimization. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 61–71, 2006.

[75] N. J. Warter, S. A. Mahlke, W.-M. W. Hwu, and B. R. Rau. Reverse if-conversion. In *Proceedings of the 1993 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '93)*, pages 290–299. ACM, 1993.

[76] B. Yoder, J. Burrill, R. McDonald, K. Bush, K. Coons, M. Gebhart, S. Govindan, B. Maher, R. Nagarajan, B. Robatmili, K. Sankaralingam, S. Sharif, A. Smith, D. Burger, S. W. Keckler, and K. S. McKinley. Software infrastructure and tools for the TRIPS prototype. In *MoBS '07: The Third Annual Workshop on Modeling, Benchmarking and Simulation.*

Vita

Bertrand Allen Maher was born in Skokie, Illinois on May 3rd, 1982, the son of Howard and Rita Maher. He received the Bachelor of Science in Electrical Engineering and Computer Science from Yale University in 2004, and entered the Ph.D. program at the University of Texas at Austin in the same year. He earned the Master of Science degree in Computer Science from the University of Texas at Austin in 2007.

Permanent address: 1813 Clemson Dr.
Richardson, TX 75081

This dissertation was typeset with LaTeX† by the author.

†LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.