ABSTRACT

Block Enlargement Optimizations for Increasing the Instruction Fetch Rate in Block-Structured Instruction Set Architectures

by
Eric Hao

Chair: Yale N. Patt

To exploit larger amounts of instruction level parallelism, processors are being built with wider issue widths and larger numbers of functional units. Instruction fetch rate must also be increased in order to effectively exploit the performance potential of such processors. Block-structured ISAs are a new class of instruction set architectures that were designed to address the performance obstacles faced by processors attempting to exploit high levels of instruction level parallelism. The major distinguishing feature of a block-structured ISA is that it defines the architectural atomic unit (i.e. the instruction) to be a group of operations which is called an atomic block. This dissertation defines an optimization, block enlargement, that can be applied to a block-structured ISA to increase the instruction fetch rate of a processor that implements that ISA. A compiler that generates block-structured ISA code and a simulator that models the execution of that code on a block-structured ISA processor were constructed to evaluate the performance benefit of block-structured ISAs. This dissertation shows that for the SPECint95 benchmarks, the block-structured ISA processor executing enlarged atomic blocks and using simpler microarchitectural mechanisms to support wide-issue and dynamic scheduling outperforms a conventional ISA processor that also supports wide-issue and dynamic scheduling by 28% when assuming perfect branch prediction and by 15% when using real branch prediction.

Block Enlargement Optimizations for Increasing the Instruction Fetch Rate in Block-Structured Instruction Set Architectures

by

Eric Hao

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
1997

Doctoral Committee:
Professor Yale N. Patt, Chair
Professor John T. Coffey
Professor Kevin J. Compton
Professor Edward S. Davidson
Pohua P. Chang, Senior Researcher, Intel Corporation

ACKNOWLEDGEMENTS

I give special thanks to my advisor, Professor Yale Patt, for all that he has taught me in the past four years. In particular, I will continue to value the intellectual and personal lessons I have learned from him long after the technical knowledge I have learned here has grown obsolete.

I also thank the members of my dissertation committee, Professor Edward Davidson, Professor Kevin Compton, Professor John Coffey, and Pohua Chang. I am deeply appreciative for the time and effort they devoted towards guiding the work of my dissertation. In addition, I thank Pohua Chang for serving as my Intel mentor and providing valuable input to the development of the compiler used in this dissertation.

I am grateful for the opportunity to have worked with five generations of the HPS group: Steve Melvin; Mike Butler, Tse-Yu Yeh, and Robert Hou; Bruce Worthington and Greg Ganger; Po-Yung Chang, Carlos Fuentes, Jared Stark, Chris Eberly, Eric Sprangle and Lea-Hwang Lee; and Sanjay Patel, Mark Evers, Dan Friendly, Paul Racunas, Peter Kim, and Rob Chappell. Their help was invaluable and their company will be missed. Special thanks go to Steve Melvin for conceiving the original ideas behind block-structured ISAs, Po-Yung Chang for helping me write the compiler, Mark Evers for developing the simulator, and Jared Stark for answering all my questions about hardware.

I am indebted to our industrial partners: Intel, NCR, Hewlett-Packard, and Motorola, for their technical and financial support. In particular, I thank Intel for the gift of the Proton compiler and the Intel Fellowship.

Finally, I thank the Hwangs for their company and support, without which my years here at Michigan would not have been nearly as enjoyable.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure

LIST OF APPENDICES

APPENDIX

CHAPTER 1

Introduction

To achieve higher levels of performance, processors are being built with wider issue widths and larger numbers of functional units. In the past ten years, instruction issue width has grown from one (MIPS R2000, Sun MicroSparc, Motorola 68020), to two (Intel Pentium, Alpha 21064) to four (MIPS R10000, Sun UltraSparc, Alpha 21164, PowerPC 604). This increase in issue width will continue as processors attempt to exploit even higher levels of instruction level parallelism. To effectively exploit the performance potential of such processors, instruction fetch rate must also increase. Because the average basic block size for integer programs is approximately five instructions, processors that aim to exploit higher levels of instruction level parallelism must be able to fetch multiple basic blocks each cycle.

1.1 The Basic Block Fetch Bottleneck

Figure 1.1 illustrates the importance of fetching multiple basic blocks per cycle for a wide issue machine. It plots the performance of the gcc benchmark from the SPECint95 benchmark suite for a sixteen wide issue, dynamically scheduled processor with perfect branch prediction. The number of blocks that the processor could fetch each cycle was varied from one to four. Increasing the number of blocks that could be fetched each cycle from one to two reduced the execution time by 35%. Increasing the number of fetched blocks from one to three reduced execution time by 45%. The reduction in execution time begins to tail off at three blocks per cycle because the processor was constrained to fetch at most sixteen instructions per cycle. Figure 1.2 shows the average number of instructions fetched per cycle (packet size) as the number of blocks that could be fetched was increased from one to four. Appendix A contains the corresponding figures for the seven other SPECint95 benchmarks.

Various approaches have been proposed for increasing instruction fetch rate from that of a single basic block per cycle. Some approaches [54, 10, 11, 45] extend the branch predictor and icache so that multiple branch predictions can be made each cycle and multiple, non-consecutive cache lines can be fetched each cycle. However, this extra hardware requires extra stages in the pipeline which will increase the branch misprediction penalty, decreasing performance. Other approaches [14, 22] statically predict the direction to be taken by a program's branches and then based on those predictions, use the compiler to arrange the blocks so that the multiple blocks to be fetched are always placed in consecutive memory

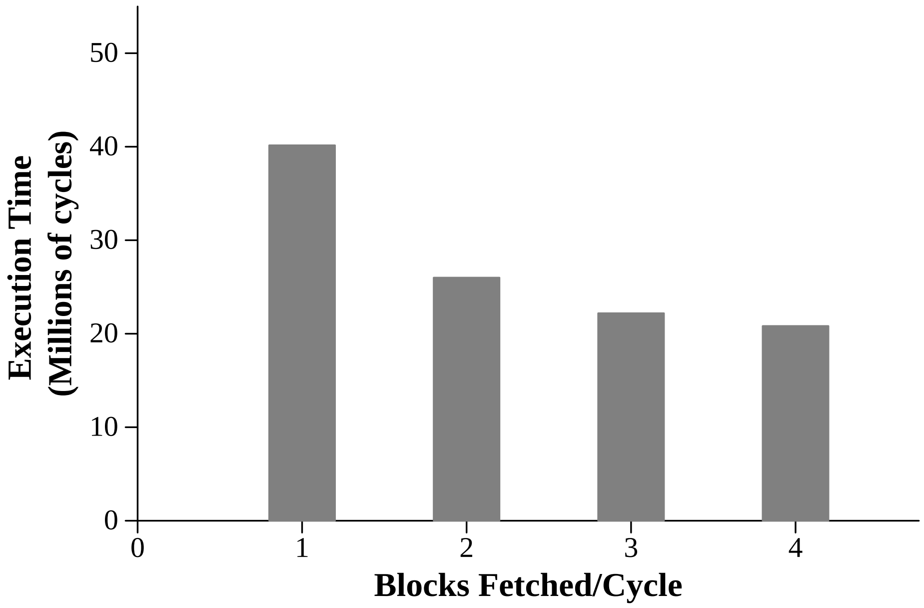

Figure 1.1: The performance of a sixteen wide issue processor executing the gcc benchmark as the number of blocks fetched per cycle is increased from one to four.

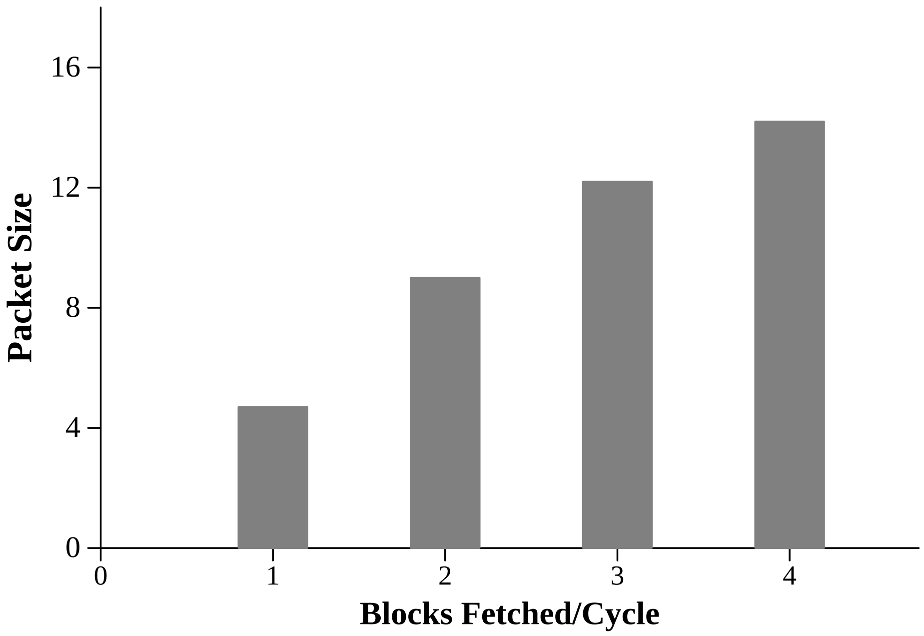

Figure 1.2: The average number of instructions fetched by a sixteen wide issue processor executing the gcc benchmark as the number of blocks fetched per cycle is increased from one to four.

locations. Although they eliminate the need for extra hardware, these approaches must rely on the branch predictions made by a static branch predictor which is usually significantly less accurate than those made by a dynamic branch predictor.

1.2 The Block-Structured ISA Solution

To overcome the basic block fetch bottleneck, this dissertation presents a solution that uses block-structured ISAs. Block-structured ISAs exploit the advantages of both compiler-based and hardware-based solutions by merging basic blocks together at compile-time and providing support for dynamic branch prediction. Block-structured ISAs [31, 33, 32, 48, 19] are a new class of instruction set architectures that were designed to address the performance obstacles faced by processors attempting to exploit high levels of instruction level parallelism. The major distinguishing feature of a block-structured ISA is that it defines the architectural atomic unit (i.e. the instruction) to be a group of operations. These groups of operations are called atomic blocks. Each operation within the atomic block corresponds roughly to an instruction in a conventional ISA. This definition of the atomic block enables the block-structured ISA to simplify many implementation issues for wide-issue processors.

1.3 Thesis Statement

Block-structured ISAs increase the instruction fetch rate of a processor (and as a result, its performance) by enabling the compiler to combine separate basic blocks into a single, enlarged atomic block. By increasing the sizes of the blocks, the instruction fetch rate of the processor is increased without having to fetch multiple blocks each cycle. Furthermore, the semantics of the block-structured ISA enable the processor to use a dynamic branch predictor to predict the successor for each atomic block fetched. As a result, block-structured ISAs increase the instruction fetch rate without relying on extra hardware to fetch non-consecutive blocks out of the icache or foregoing the use of dynamic branch prediction.

1.4 Contributions

This dissertation makes two key contributions:

1. It demonstrates that a processor implementing a block-structured ISA can achieve a significantly higher instruction fetch rate than that achieved by a processor implementing a conventional ISA. This increase in instruction fetch rate translates to a 28% performance improvement when perfect branch prediction is assumed and a 15% performance improvement when real branch prediction is used.

2. It presents the design of a dynamic branch predictor for block-structured ISAs. It demonstrates how this predictor is able to achieve a prediction accuracy that is comparable to that of an aggressive dynamic branch predictor for a conventional ISA.

1.5 Dissertation Organization

This dissertation is organized into eight chapters. Chapter 2 discusses other approaches to increasing the instruction fetch rate. Chapter 3 describes the simulation methodology used throughout this dissertation. Chapter 4 gives an overview of block-structured ISAs, explaining how the block enlargement optimization works and how it increases the instruction fetch rate. Chapter 4 also defines the specific block-structured ISA studied in this dissertation. Chapter 5 evaluates the performance benefit of the block enlargement optimization, focusing on its ability to increase the average block size. It presents experimental results comparing the performance of the block-structured ISA to that of a conventional ISA. Chapter 6 describes the implementation of a dynamic branch predictor for block-structured ISAs. Chapter 7 evaluates the performance of this branch predictor as compared to an aggressive branch predictor for a conventional ISA. Chapter 7 also examines the block enlargement optimization's impact on branch prediction accuracy. Chapter 8 evaluates the performance benefit of the block enlargement optimization for scientific code. Chapter 9 presents concluding remarks and directions for future work.

CHAPTER 2

Related Work

The majority of the approaches previously proposed for increasing the instruction fetch rate can be divided into two categories, hardware-based and compiler-based. The hardware-based schemes extend the branch predictor and icache so that multiple branch predictions can be made each cycle and multiple non-consecutive cache lines can be fetched each cycle. They include the branch address cache [54], the collapsing buffer [10], the subgraph-level predictor [11], the multiple-block ahead branch predictor [45], and the trace cache [35, 44, 39]. The compiler-based schemes reorganize the program to increase the amount of work that can be fetched with a single icache access, eliminating the need for extra hardware. These schemes include trace and superblock scheduling [14, 4, 22], predicated execution [20, 42], hyperblock scheduling [29], the VLIW multi-way jump mechanism [13, 26, 12, 36], and multiscalar processors [15, 47].

2.1 Hardware-based Approaches

2.1.1 Predicting Multiple Branches Per Cycle

The branch address cache [54], the collapsing buffer [10], the subgraph-level predictor [11], and the multiple-block ahead branch predictor [45] are hardware schemes that propose different ways to extend the dynamic branch predictor so that it can make multiple branch predictions each cycle. Because some of the branches may be predicted to be taken, these schemes all require the ability to fetch multiple non-consecutive lines from the icache each cycle. They all propose to meet this requirement by interleaving the icache. This general approach has three disadvantages. First, bank conflicts will arise in the icache when fetching multiple lines from the same bank. To handle this conflict, the fetch for all but one of the conflicting lines must be delayed. This first disadvantage can be minimized if the icache is interleaved with a large enough number of banks. Second, because it is fetching multiple non-consecutive blocks from the icache, the processor must determine which instructions from the fetched cache lines correspond to the desired basic blocks and reorder the instructions so that they correspond to the order of those basic blocks. The processor will require at least one additional stage in the pipeline in order to accomplish these tasks. This additional stage will increase the branch misprediction penalty, decreasing overall performance. Third, these approaches do not provide solutions to the problem of increased hardware complexity required to support wide issue widths. In particular, the hardware required to perform dependency checking, register file access, and architectural state maintenance increases as the issue width increases. If this problem is not addressed, then this increase in hardware complexity will result in increased cycle times or additional pipeline stages for an aggressive wide-issue machine, which in turn, will decrease performance.

2.1.2 The Trace Cache

The trace cache [35, 44, 39] is a hardware-based scheme that does not require fetching non-consecutive blocks from the icache. Its fetch unit consists of three parts: a core fetch unit, a fill unit, and a trace cache. The core fetch unit fetches one basic block per cycle from the icache. The fill unit records sequences of basic blocks fetched by the core fetch unit, combining each sequence into a trace. These traces are then stored in the trace cache. If the branch predictor indicates that the sequence of basic blocks to be fetched matches a trace stored in the trace cache, then the processor is able to fetch multiple blocks that cycle by fetching the specified trace from the trace cache. Because the blocks have been combined into a contiguous trace, the processor need only fetch a single trace cache line to get all the blocks. In addition, because the blocks' instructions are stored in decoded form with their dependencies specified, the instructions fetched from the trace cache can bypass the pipeline stages that generate this information for instructions fetched from the icache. If no matching trace is found, the processor is able to fetch only one basic block that cycle via the core fetch unit. As long as the processor is fetching its instructions from the trace cache, the trace cache is an effective means for fetching multiple basic blocks each cycle without incurring the costs associated with the other hardware-based approaches.

The trace cache idea was originally proposed by Melvin, Shebanow, and Patt [35]. They proposed using the trace cache (or decoded instruction cache) to ease the instruction decoding bottleneck for HPS implementations of complex instruction sets such as the VAX. The core fetch unit fetched one instruction per cycle. The fill unit recorded the sequences of microoperations that were generated for each instruction. Each decoded instruction cache entry held the microoperations that corresponded to the set of instructions within a given basic block. Thus, a hit in the decoded instruction cache would result in the fetch and issue of an entire basic block's worth of instructions. Later, Melvin and Patt [34] suggested using the fill unit to combine instructions from different basic blocks. Rotenberg et al. [44] and Patel et al. [39] independently proposed implementations of the trace cache that combined basic blocks as suggested by Melvin and Patt. These two trace cache implementations showed significant performance improvements over previously proposed fetch mechanisms including the branch address cache and the collapsing buffer. The implementation proposed by Rotenberg et al. used the trace cache as the secondary fetch mechanism. The majority of the hardware was devoted to the icache. The implementation proposed by Patel et al. [39] used the trace cache as the primary fetch mechanism. By devoting more hardware to the trace cache, they showed that further performance improvements could be achieved. In addition, they showed that the trace cache works well even when compared to a single block fetch mechanism that uses an aggressive branch predictor.

2.2 Software-based Approaches

2.2.1 Trace and Superblock Scheduling

Trace scheduling [14] and superblock scheduling [22, 4, 9] are compiler optimizations that enlarge the scope in which the compiler can schedule instructions. They use static branch prediction to determine the frequently executed program paths and place the basic blocks along these paths into consecutive locations, forming a superblock. The instructions within the superblock can then by optimized as if they were in a single basic block. Although the focus of superblock scheduling is to enlarge the scope in which the compiler can schedule instructions, it can also be used to increase the instruction fetch rate. Each superblock can be fetched in a single cycle because the basic blocks which form the superblock are contiguous.

Figure 2.1 shows an application of the superblock optimization. The control flow graph on the left consists of basic blocks A–E. The control flow graph on the right is the result of applying superblock scheduling. The static branch predictor has predicted that block C is

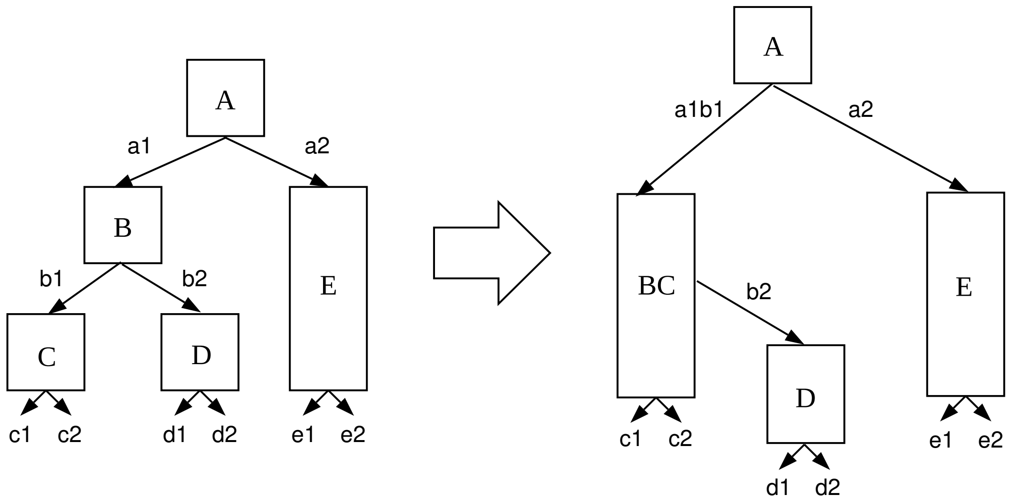

Figure 2.1: Using superblock scheduling to combine basic blocks.

the most likely successor to block B so blocks B and C are combined to form a superblock. As a result, blocks B and C can be fetched in the same cycle. The disadvantage of this approach is that by optimizing blocks B and C as a single unit, block B cannot be fetched with any other block in a single cycle. Even if the processor knew that block D was the correct control flow successor to block B, the processor will still need two cycles to fetch blocks B and D. In effect, to fetch multiple basic blocks in a cycle, the processor must follow the static branch predictions used to form the superblocks. Because dynamic branch prediction is significantly more accurate than static branch prediction [53], using superblock scheduling to increase the instruction fetch rate will not be as effective as an approach that uses dynamic branch prediction to select the set of blocks to be fetched each cycle.

2.2.2 Predicated Execution

A processor that supports predicated execution [2, 20, 43, 42] associates a predicate register with each instruction issued. The execution of an instruction proceeds normally if the predicate register resolves to a true value. The execution is suppressed if that predicate value resolves to a false value. Using predicated instructions, a compiler can eliminate branches in a program by converting the program control dependencies into data dependencies. By eliminating branches, predicated execution can improve performance because the processor will not have to suffer the branch execution penalty associated with those branches. In addition, predicated execution can increase the instruction fetch rate. Once a basic block's branch has been eliminated, it can be combined with its control flow successors to form one larger block. As a result, the processor can now fetch the original basic block along with its control flow successors in a single cycle. Figure 2.2 gives an example of this process. The branch at the end of block A is eliminated via predicated execution, resulting in the combination of blocks A, B, C, and D into a single block.

In contrast to its benefits, predicated execution has two disadvantages. First, it wastes fetch and issue bandwidth fetching and issuing instructions that are suppressed because their predicates evaluate to false. Second, by converting an instruction's control dependency into a data dependency, the program's critical paths may be lengthened. The processor must now wait for the new data dependency to be resolved instead of speculatively resolving the control dependency at fetch time. While predicated execution by itself may not be an effective mechanism for increasing fetch rate, it can provide a significant performance benefit when used in conjunction with speculative execution [28, 7] and other schemes for increasing fetch rate.

2.2.3 Hyperblock Scheduling

Hyperblock scheduling [29, 28] is an extension of superblock scheduling that incorporates predicated execution. As discussed earlier, the processor must always follow the static branch predictions used to form the superblock. If the branches inside a superblock are not highly biased, then using a static branch prediction will result in a low prediction accuracy for that branch. To avoid these mispredictions, hyperblock scheduling will use predicated execution to eliminate such branches. The highly biased branches are not eliminated because they can be accurately predicted by a static branch predictor. Figure 2.3 illustrates the difference between hyperblock and superblock scheduling. The control flow graph at the top shows basic blocks A–G where the branch in block B is slightly biased in the b1 direction and the branch in block D is highly biased in the d1 direction. The control flow graph on the bottom left is the result of applying superblock scheduling to the original control flow graph. Because the branch in block B is not highly biased, it will be frequently mispredicted by the static branch predictor. As a result, the processor will frequently fetch and issue blocks BCE and DE. The control flow graph on the bottom right is the result of applying hyperblock scheduling to the original control flow graph. Because the two frequently executed paths, BCE and BDE, are both included in the hyperblock BCDE, the processor will frequently fetch and issue only block BCDE. Block F will rarely have to be fetched after block BCDE because the branch in block D is highly biased in the d1 direction. By using predicated execution,

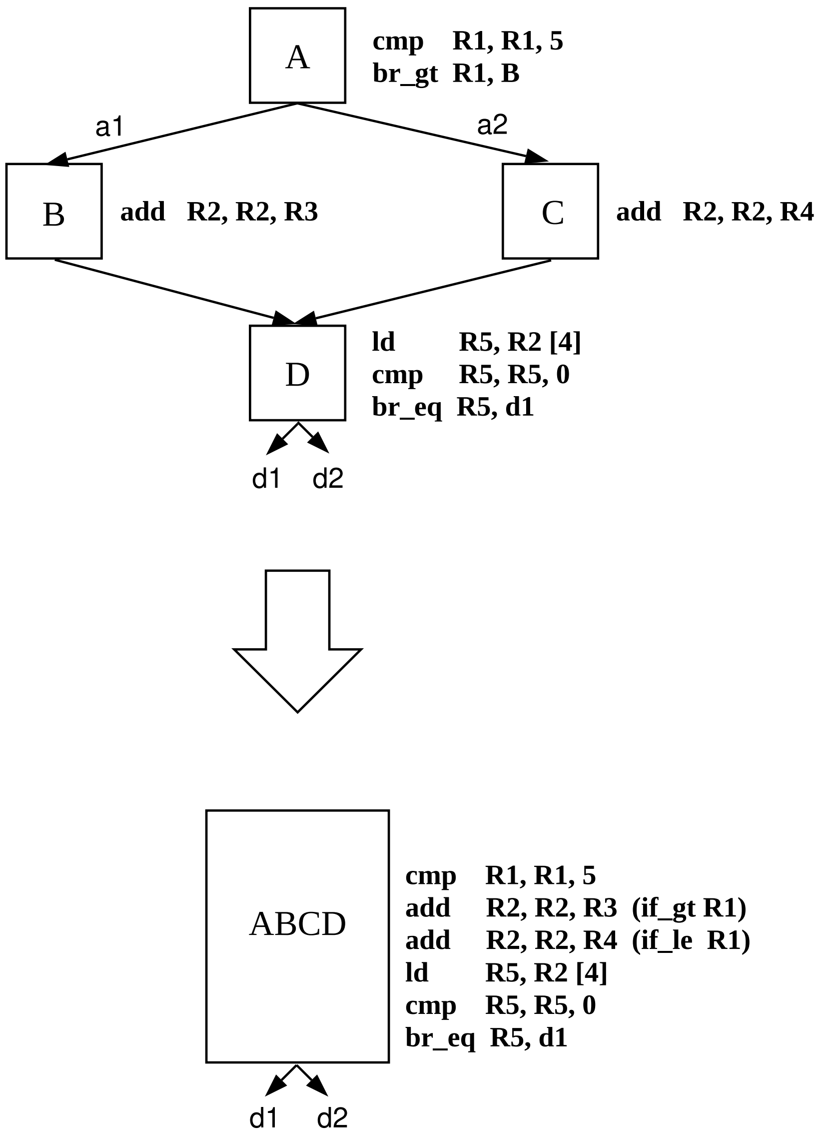

Figure 2.2: Using predicated execution to combine basic blocks.

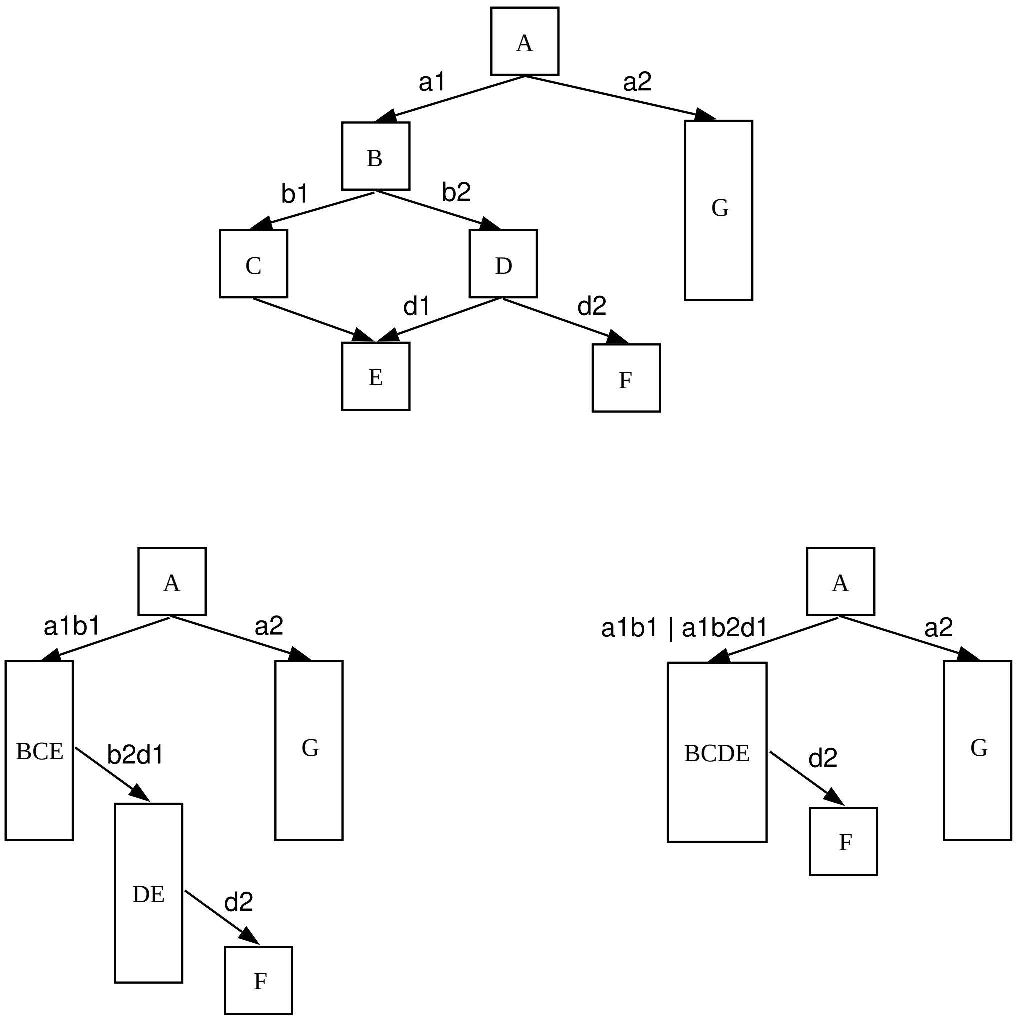

Figure 2.3: Comparing superblock and hyperblock scheduling.

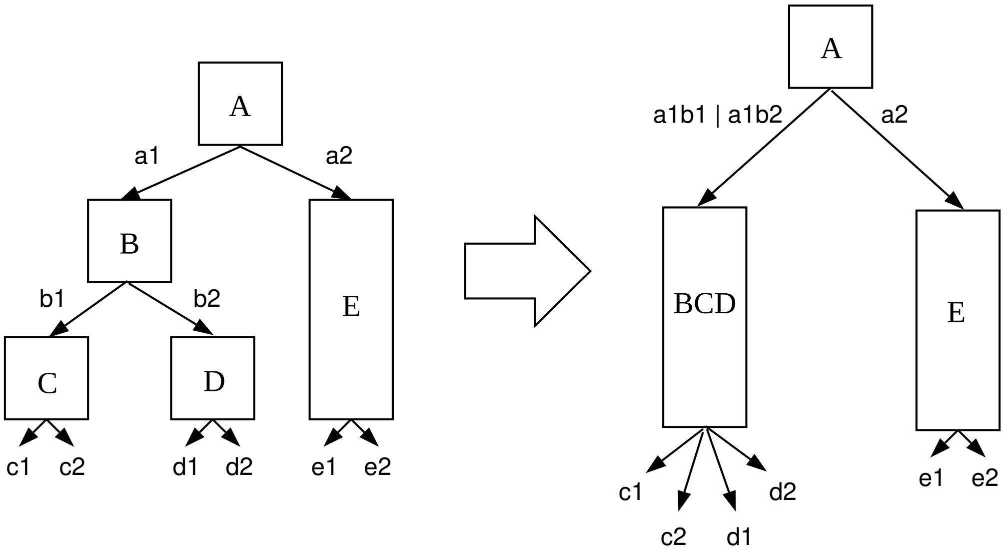

Figure 2.4: Building a VLIW tree instruction.

hyperblock scheduling is able to avoid many of the mispredictions incurred by superblock scheduling. However, because hyperblock scheduling uses predicated execution, it suffers the disadvantages of wasted fetch and issue bandwidth and increased critical path lengths that are associated with predicated execution. Hyperblock scheduling must be carefully controlled to ensure that this tradeoff results in a performance improvement.

2.2.4 VLIW Multi-Way Jumps

The VLIW multi-way jump mechanism [13, 26, 12, 36] combines multiple branches from multiple paths in the control flow graph into a single branch. Using this mechanism along with predicated execution, basic blocks which form a rooted subgraph in the control flow graph can be combined into a single VLIW instruction, which is called a tree instruction. The branches for these basic blocks are combined into a single multi-way branch operation. The processor's condition code registers will specify which path through the rooted subgraph is to be executed (predicated execution is used to suppress the execution of operations from basic blocks not on this path) and which target of the multi-way branch operation is to be the next instruction. Figure 2.4 gives an example of how a tree instruction is formed. Blocks B, C, and D are combined into a single tree instruction. The figure assumes that all the instructions within B, C, and D are independent. When block BCD is issued into the processor, the direction taken by the branches in the original blocks B, C, and D will already be known. If block B's branch has taken the b1 direction and block C's branch has taken the c1 direction, then the execution of block D in BCD will be suppressed and the successor block that corresponds to block C's c1 direction will be the next block fetched. In this example, the direction taken by block D's branch is irrelevant because block D is not on the path of execution.

Because a single tree instruction can include operations from many basic blocks, this approach gives VLIW processors the means to fetch the equivalent work of multiple basic blocks each cycle. However, because the operations within a VLIW instruction must be independent, it is critical that the compiler be able to find enough independent instructions to fill each VLIW instruction and be able to schedule the operations which evaluate the condition codes for the multi-way jump early enough so that the condition codes are available when the multi-way jump is issued. The compiler may have to delay the scheduling of certain operations in order to meet these requirements, lowering the instruction fetch rate of the processor.

2.2.5 Multiscalar Processors

The multiscalar processing paradigm [15, 47] applies multiprocessor concepts to the design of microprocessors. A multiscalar processor consists of a set of processing elements connected in a ring. The multiscalar processor's compiler is responsible for dividing the program up into units of work, called tasks. These tasks are dynamically assigned to the processing elements for execution. The connecting logic among the processing elements detects and enforces the data dependencies among the tasks. This logic is the multiscalar counterpart for the cache coherency logic for shared memory multiprocessors except that it enforces not only dependencies among memory references from different processing elements, but dependencies among register references from different processing elements as well. Values are communicated among the processing elements via the ring network.

Multiscalar processors eliminate the problem of fetching multiple cache lines from the icache each cycle by associating a separate level one icache with each processing element. Each cycle, each processing element accesses its own level one icache for a single basic block. As long as each icache achieves a sufficient hit rate, the multiscalar processor as a whole is able to fetch the equivalent work of multiple basic blocks each cycle. However, the multiscalar model raises new performance issues not found in traditional wide-issue processors. To achieve high performance, it is important that the multiscalar compiler create tasks so that the workload is evenly balanced among the processing elements and that the communication among the tasks does not exceed the ring bandwidth

CHAPTER 3

Simulation Methodology

To evaluate the performance advantages of block-structured ISAs, the performance of an HPS implementation of a block-structured ISA is compared to an identically configured HPS implementation of a conventional (or base) ISA. The conventional ISA considered is the same load/store ISA that formed the basis of the block-structured ISA (see chapter 4.2). This ensures that the block-structured ISA has no architectural advantages over the conventional ISA with the exception of those due to block-structuring. Figure 3.1 gives an overview of how this comparison is performed. A compiler generates the block-structured and conventional ISA executables. These executables are passed to the simulators which model the corresponding HPS implementations and generate the performance statistics. This chapter details each component of the experimental setup.

3.1 The Compiler

I implemented a compiler that was targeted for both the block-structured and the conventional ISAs. This compiler is based on the Intel Reference C Compiler [24] with the back end appropriately retargeted. The Intel Reference C Compiler generates an intermediate representation of the program being compiled and applies the standard set of optimizations to that representation. The compiler back end takes this representation and applies a set of target-specific optimizations and allocates registers. By using the same core compiler for both ISAs, any unfair compiler advantages one ISA may have had over the other was eliminated.

3.2 The HPS Processor Model

The HPS paradigm [40, 41] is used to model the processor implementation in this study because it encompasses a set of microarchitectural techniques that are designed to achieve high performance for single instruction stream execution. These techniques include aggressive branch prediction, wide instruction issue, out-of-order execution, and precise exception handling and have been adopted by all currently introduced microprocessors [16, 17, 18].

The HPS implementations modeled issue sixteen instructions per cycle. This issue width was chosen for two reasons. First, because the current generation of microprocessors are

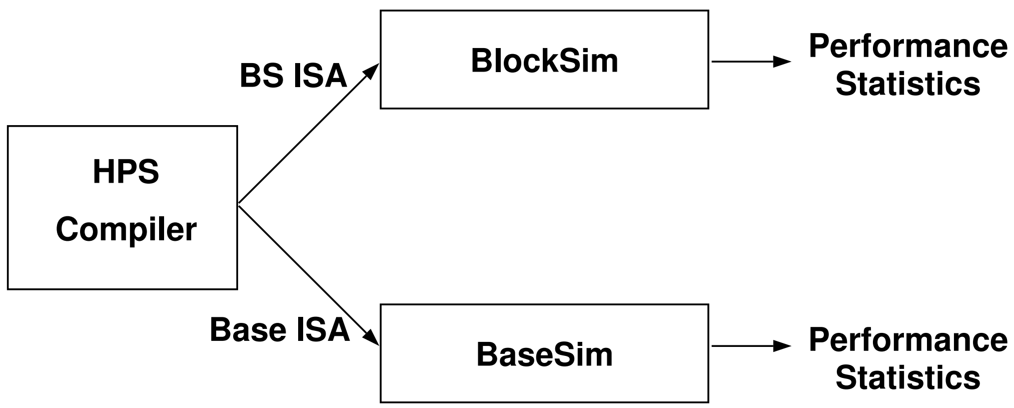

Figure 3.1: Experimental setup for comparing block-structured ISA performance to conventional ISA performance.

already issuing four instructions per cycle, sixteen wide issue is the minimum issue width that is aggressive enough to represent a design point far enough in the future to be considered research. Second, the block enlargement optimizations are designed to improve performance for wide-issue machines. They have little benefit for narrow issue widths. Using a generalized version of the Tomasulo algorithm [51], the instructions' registers are dynamically renamed in parallel and sent to an appropriate node table entry to await execution. An instruction is executed as soon as its source operands are ready and a functional unit is free. The HPS implementations modeled have 16 functional units with each functional unit capable of executing instructions from any instruction class. Table 3.1 lists the simulated latencies for each instruction class. After execution, an instruction's result is forwarded to the register file and to the other instructions awaiting execution in the node tables. An instruction is retired when its associated checkpoint [23] is retired. This occurs after all the instructions in the checkpoint have successfully executed and all previously issued checkpoints have been retired. The HPS implementations modeled can hold up to 32 checkpoints for a total of 256 instructions. The processor will stop fetching and issuing instructions if all 32 checkpoints are in use. This break in instruction fetch and issue is called a full-window stall. A full-window stall will continue until a checkpoint is retired (i.e. freed up for use). Unless otherwise noted, the level one icache size modeled was 128KB [1]. For the SPECint95 experiments, the level one dcache modeled was 16KB. The data requirements for the SPECint95 benchmarks are low enough so that using a small dcache has little effect on performance. For the SPECfp95 experiments, the level one dcache modeled was 128KB to handle the larger data requirements for the SPECfp benchmarks. The level two caches had six cycle access times and were modeled as perfect caches.

The same, aggressive HPS configuration was used for both the block-structured and conventional ISA processors. While, this guaranteed that neither ISA had a microarchitectural

[1]A 128KB icache with a single cycle access time may appear aggressive compared to currently available microprocessor designs. However, advances in process technology will make such icaches possible in the future. Digital has already announced the 21264 Alpha processor which has a 64KB, two-way set associative icache with a single cycle access time [18].

Class	Latency	Description
Integer	1	INT add, sub and logic OPs
FP Add	3	FP add, sub, and convert
FP/INT Mul	3	FP mul and INT mul
FP/INT Div	8	FP div and INT div
Load	2	Memory loads
Store	-	Memory stores
Bit Field	1	Shift and bit testing
Branch	1	Trap, fault, and other control instructions

Table 3.1: Instruction classes and latencies.

advantage over the other, this constraint ignored the hardware advantages of block-structured ISAs. As mentioned in chapter 4, block-structured ISAs simplify the hardware required to implement wide-issue processors. If this advantage were to be exploited, then for a fixed hardware cost, the block-structured ISA processor would have either a faster cycle time or a wider issue width than that of the conventional ISA processor.

3.3 The Simulators

Two trace-driven simulators were implemented to measure the performance of the conventional and block-structured ISAs, BaseSim and BlockSim. Both simulators were divided into two parts, a front end and a back end. The front ends were responsible for actually executing the specified executable and generating an instruction trace that corresponded to the correct execution of that executable. The back ends were responsible for modeling the microarchitectural components of the processor in order to determine the performance achieved by the specified HPS implementation while executing the instruction trace. The microarchitectural components modeled included the branch predictor, the node tables, the functional units, the distribution buses, the register files and renaming logic, and the caches.

The instruction trace generated by BaseSim's front end included only instructions that were on the correct path of execution. By excluding instructions from incorrect speculative (or wrong) paths, BaseSim was unable to model the effects of issuing such instructions into the machine [3, 25]. However, the execution penalty associated with the mispredicted branch that created the incorrect speculative path is still correctly modeled. When a branch misprediction occurs in a real machine, the processor continues to fetch instructions from the wrong path until the mispredicted branch is resolved. At that time, the architectural state of the processor is restored to its state at the point of the mispredicted branch, the instructions issued from the wrong path are removed from the node tables, and the processor begins fetching instructions from the correct path. When a branch misprediction occurs in the simulator, the simulator simply stalls the instruction fetch mechanism until the branch is resolved. By stalling, the simulator mimics the effect of not fetching and issuing useful work during this time without having to actually fetch and issue instructions from the incorrect path. After the mispredicted branch is resolved, the simulator restarts instruction fetch from the correct path, which is the next instruction in the instruction trace.

To accurately model branch misprediction penalties for a processor implementing a block-structured ISA, the simulator must take into account the effect of wrong path instructions for certain branch mispredictions (see chapter 4.1.4 for more details). As a result, the instruction trace generated by BlockSim's front end must include the wrong path instructions that occur after these specific branch mispredictions. To accomplish this, BlockSim's back end communicates its branch predictions to the front end so that the front end knows when a branch misprediction will occur and includes the appropriate instructions in the instruction trace.

3.4 The SPEC95 Benchmarks

The SPEC benchmarks [49] were used to evaluate the performance of the block-structured and conventional ISAs. Tables 3.2 lists the eight SPECint95 used along with their test and training data sets. The test data sets were used to generate the performance numbers reported. The training sets were used to generate the benchmark profiles for the experiments that required profiling. The test and training data sets were all either the reference data sets provided with the SPEC benchmarks or modified versions of those data sets. Table 3.3 lists the ten SPECfp95 benchmarks used along with their test data sets. The table does not include a listing of training data sets because profiling was not used for the SPECfp benchmarks. Modified data sets were used for both the SPECint95 and SPECfp95 benchmarks whenever the running time for the reference data set was too long. Appendices B and C give descriptions of the modified data sets used.

Benchmark	Description	Test Set	Training Set
gcc	GNU C compiler	jump.i	stmt.i
compress	Data compression program	30KB.in*	300B.in*
go	Go-playing program	2stone9.in	null.in
ijpeg	Image compression program	specmun.ppm	vigo.ppm
li	Xlisp interpreter	boyer.lsp	queens.lsp
m88ksim	Simulator for 88100 processor	dcrand	dhry
perl	Perl interpreter	scrabbl.pl	primes.pl
vortex	Object-oriented database	test.in*	profile.in*

Table 3.2: The SPECint95 benchmarks and their test and training data sets. * indicates the input set is an abbreviated version of the SPECint95 reference input set.

Benchmark	Description	Test Set
tomcatv	Mesh generation	tomcatv.in
swim	Shallow water equation solver	swim.in
su2cor	Monte Carlo method for particle mass computation	su2cor.in
hydro2d	Navier Stokes equation solver	hydro2d.in
mgrid	3D multigrid solver	mgrid.in
applu	Partial differential equation solver	applu.in
turb3d	Turbulence modeling	turb3d.in
apsi	Weather modeling	apsi.in
fpppp	Quantum chemistry problem	fpppp.in
wave5	Maxwell's equations solver	wave5.in

Table 3.3: The SPECfp95 benchmarks and their test data sets. All the test data sets used were abbreviated versions of the SPECfp95 reference input sets.

CHAPTER 4

Block-Structured ISAs and the Block Enlargement Optimization

Block-structured ISAs [31, 33, 32, 48] were designed to help solve performance obstacles faced by wide issue processors. Their major distinguishing feature is that the architectural atomic unit is defined to be a group of operations, where an operation typically corresponds to an instruction in a load/store architecture. These groups, known as atomic blocks, are specified by the compiler. When an atomic block is issued into the machine, either every operation in the block is executed or none of the operations in the block are executed. The semantics of the atomic block enable the block-structured ISA to explicitly represent the dependencies among the operations within a block and to list the operations within the block in any order without affecting the semantics of the block. These features simplify the implementation of a wide issue processor by simplifying the logic required for recording architectural state, checking dependencies, accessing the register file, and routing operations to the appropriate reservation stations. By reducing hardware complexity, wide issue implementations of a block-structured ISA will require fewer hardware resources than that of a wide issue implementation of a conventional ISA, resulting in a faster cycle time or a shallower pipeline. While these benefits are critical to building wide issue processors, this dissertation focuses on the ability of block-structured ISAs to increase the instruction fetch rate. This chapter will describe the block enlargement optimization, a compiler optimization used by block-structured ISAs to increase the instruction fetch rate and the details of the specific block-structured ISA that will be studied throughout this dissertation.

4.1 The Block Enlargement Optimization

4.1.1 Overview

The block enlargement optimization increases the size of an atomic block by combining the block with its control flow successors. Figure 4.1 illustrates how block enlargement works. The control flow graph on the left consists of the atomic blocks A–E, each one ending with a branch that specifies its two successor blocks. These branches are called trap operations to differentiate them from fault operations which will be described below. These blocks are analogous to the basic blocks in a control flow graph for a conventional ISA. The control

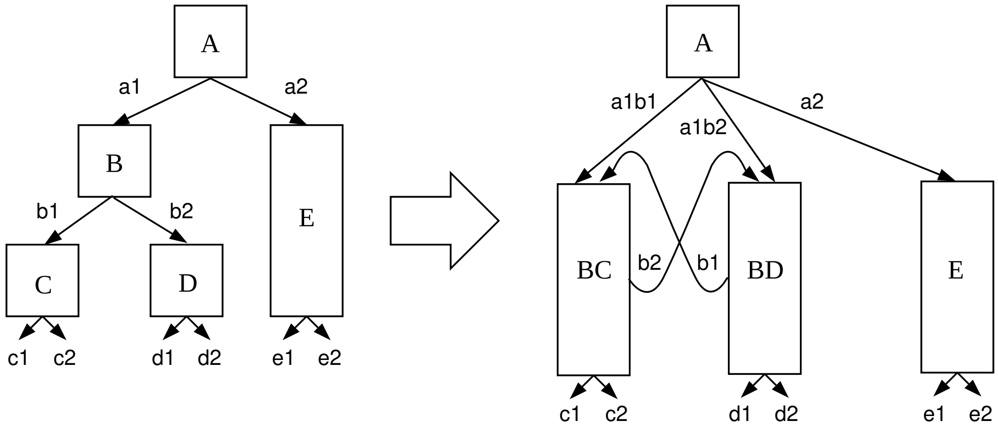

Figure 4.1: Combining atomic blocks into an enlarged atomic block.

flow graph on the right shows the result of combining atomic block B with its control flow successors C and D to form the enlarged atomic blocks BC and BD. Both blocks BC and BD are now control flow successors to block A.

To support the block enlargement optimization, a new class of branch operations, the fault operation, is included in block-structured ISAs. The fault operation takes a condition and a target. If the condition evaluates to false, the fault operation has no effect. If the condition evaluates to true, the instructions from the atomic block to which it belongs are discarded and the instruction stream is redirected to its target. When two blocks are combined, the trap operation at the end of the first block is converted into a fault operation. If a block is combined with its fall-through successor, then the condition of the resulting fault operation is the same as the original trap operation's condition. If a block is combined with the target of its trap operation, then the condition of the resulting fault operation is the complement of the original trap operation's condition. The target of the fault operation is the enlarged block that results from combining the first block with its other control flow successor. In figure 4.1, when blocks B and C are combined, the trap at the end of B is converted into a fault in block BC. The fault's condition is true whenever block D is suppose to follow block B in the dynamic instruction stream and the fault's target is BD. Block BD contains a corresponding fault operation with a complementary condition and a target that points back to BC. Figure 4.2 shows this transformation using sample code sequences for blocks A—D.

Enlarging atomic blocks changes the way a processor sequences through a control flow graph because the control flow edges traversed are no longer solely determined by the operations that precede it. Enlarging an atomic block causes multiple branches to be placed in a single block. Referring back to figure 4.2, enlarged blocks BC and BD both contain two branches, the fault operation which corresponds to the original trap at the end of B and the trap operation at the end of C or D. This results in block A having three control flow successors, blocks BC, BD, and E. Block A's correct control flow successor is specified by not only the direction taken by block A's trap operation, but by the direction taken by block B's branch (i.e. the fault operations in BC and BD). Because the operations that evaluate block B's condition may be in B itself, the correct successor to A cannot be determined until

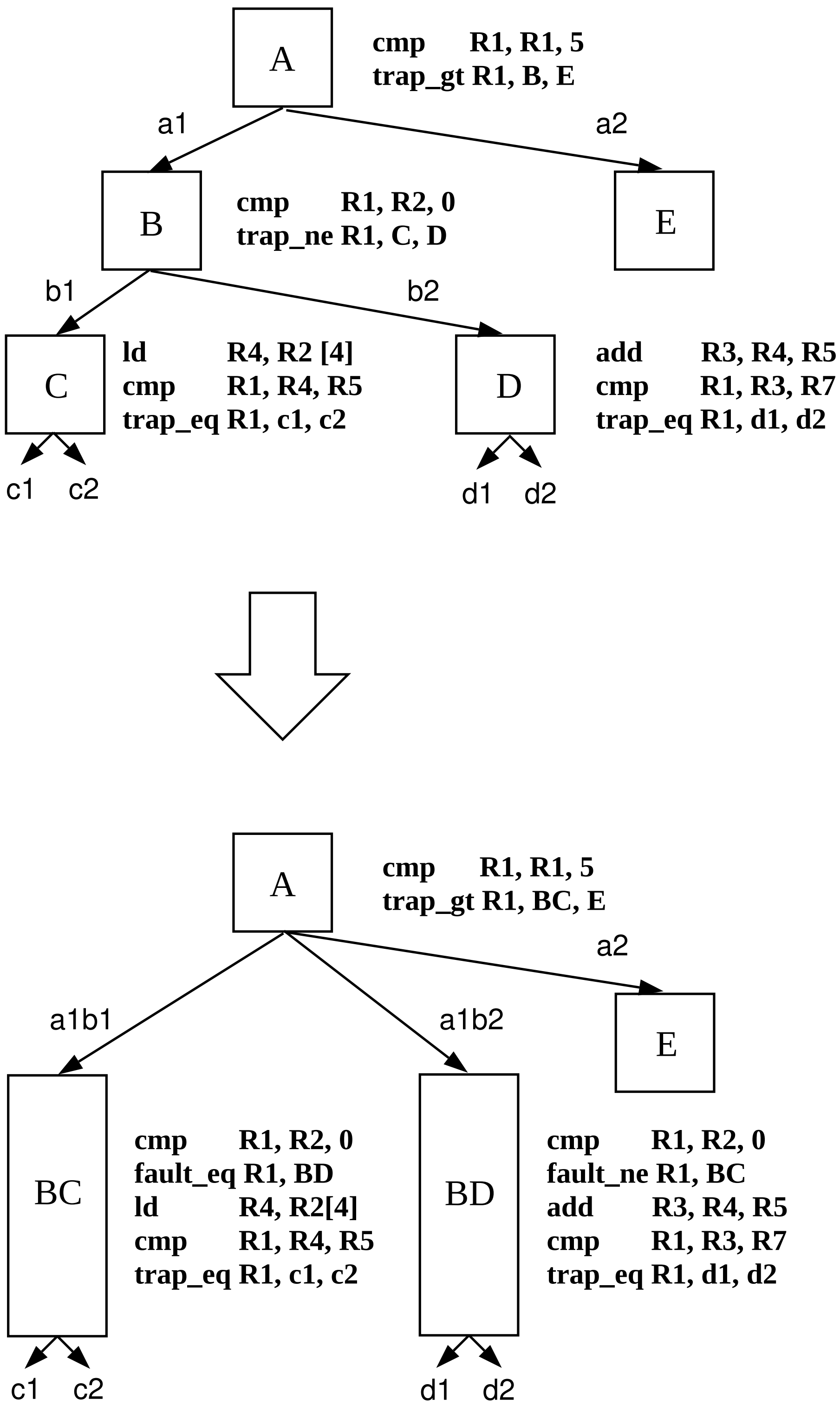

Figure 4.2: Converting traps into faults for the block enlargement optimization.

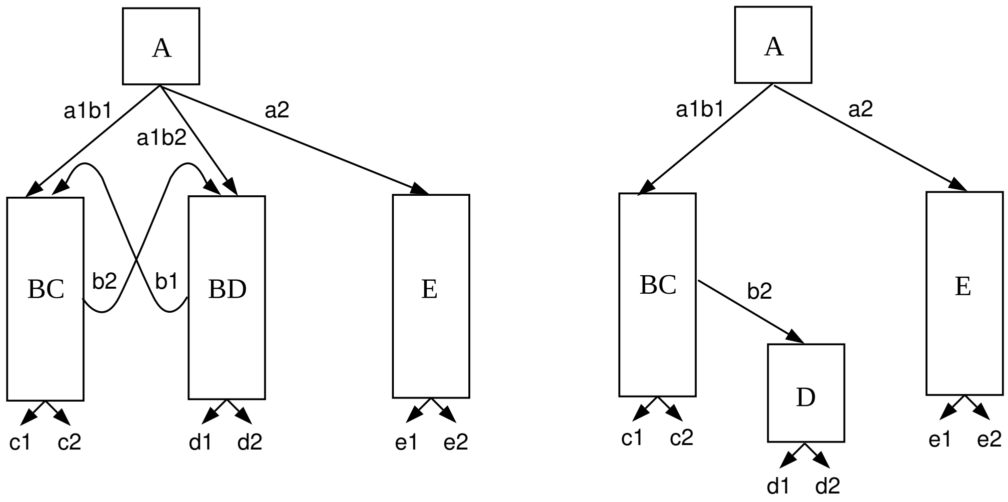

Figure 4.3: Comparing the block enlargement optimization to superblock scheduling.

after BC or BD has been issued. To arrive at the correct successor, a successor candidate must be chosen that is a possible successor given the current trap condition. If block A's trap indicates that control flow edge a1 is to be taken, blocks BC or BD can be chosen as the successor candidate. In effect, block A's trap serves only to partition the control flow successors into two subsets, the taken subset and the not taken subset. It does not specify exactly which control flow successor is the correct one. The processor must then depend on the faults within the chosen successor block to redirect control to the correct successor block as previously described.

Figure 4.3 contrasts the difference between the block enlargement optimization and superblock scheduling. Starting with the original control flow graph found on the left sides of figures 2.1 and 4.1, the control flow graph on the left side of figure 4.3 is the result of applying the block enlargement optimization. The control flow graph on the right is the result of applying superblock scheduling. There are two key differences. First, the block enlargement optimization gives the processor the opportunity to fetch either blocks B and C together in a single cycle or blocks B and D together in a single cycle. Superblock scheduling allows only blocks B and C to be fetched together. For the block enlargement case, the dynamic branch predictor can be used to predict every branch in the program. For the superblock case, the dynamic branch predictor can be used to predict only the branches that end superblocks. Predictions for branches that reside inside a superblock are constrained to be the static branch predictions used to form the superblocks. Because dynamic branch predictors usually achieve significantly higher prediction accuracies than static branch predictors, this extra degree of freedom provides a performance advantage for the block enlargement optimization over superblock scheduling. Second, mispredicting a branch inside an enlarged block (i.e. a fault operation) incurs an extra penalty not associated with ordinary branch mispredictions, because it causes all the work in that block to be discarded. Some of this

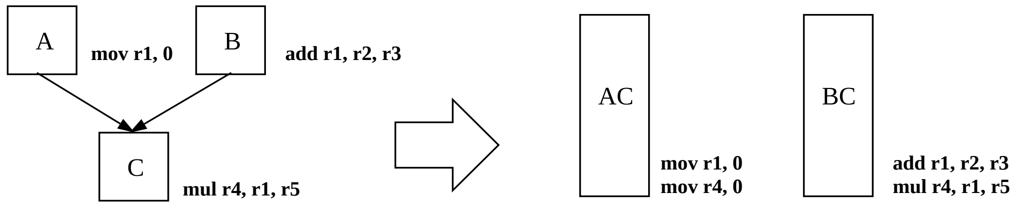

Figure 4.4: Using block enlargement to increase the compiler's scope for local optimization.

work may have to be issued and executed again after the correct block is fetched. For example, in figure 4.3, if block BC is fetched and issued and its fault operation fires, then the entire block will be discarded and block B will have to be fetched and issued again in the form of block BD. For a misprediction of a branch inside a superblock, the entire superblock is not discarded, just the work that follows the mispredicted branch. Referring again to figure 4.3, if block BC is fetched and issued and the branch inside the block redirects the control flow to block D, block C is suppressed but block B remains in the machine.

Using the block enlargement optimization, block-structured ISAs are able to increase the instruction fetch rate without suffering the disadvantages associated with traditional approaches — the need to fetch multiple non-consecutive cache lines each cycle and the constraint of having to use static branch prediction instead of dynamic branch prediction for certain select branches in the program.

4.1.2 Enlarging the Compiler's Scope for Optimization

In addition to increasing the instruction rate, the block enlargement optimization also increases the compiler's scope for local optimization in the same manner as is done in trace scheduling [14] and superblock scheduling [22]. Compiler optimizations can be divided into two classes: local and global [1]. Local optimizations are optimizations that are performed upon instructions that reside within a single block. They are narrow in focus. Global optimizations are optimizations that are performed across multiple basic blocks. By considering a wider scope, global optimizations can find opportunities for improving the code beyond what is done by local optimizations. However, when applying a global optimization across a set of basic blocks, the compiler must ensure that the optimization does not cause incorrect results or increase the execution time for any path the program may take through those basic blocks. Consider the control flow graph on the left side of figure 4.4. If the program's path of execution is assumed to proceed from block A to block C, then the instruction **mul r4, r1, r5** in block C can be optimized to **mov r4, 0**. However, if the program's path of execution was to proceed from block B to block C, this optimization would cause incorrect results.

When a set of basic blocks are combined together into a single enlarged block via the block enlargement optimization, that set of basic blocks becomes an atomic unit and can be treated as a single basic block. Thus the compiler can apply optimizations to that set of blocks as if the optimizations were local optimizations. The compiler no longer has to worry about the optimizations' effects on the different program paths through the set of basic blocks. Consider the control flow graph on the right side of figure 4.4, the result of applying the block enlargement optimization to the control flow graph on the left. The **mul r4, r1, r5** from block C has been optimized to **mov r4, 0** for block AC while remaining in its original form in block BC. If the program's path of execution were to proceed from block A to block C, enlarged block AC would be issued into the machine. If the path of execution were to proceed from block B to block C, enlarged block BC would be issued into the machine. In either case, the best possible version of block C is issued into the machine.

4.1.3 Effect on ICache Performance

In addition to the average block size, the icache hit rate is another important factor that determines the instruction fetch rate. For the block enlargement optimization to be effective, it must increase the average block size without significantly reducing icache performance. Because it increases the size of the executable, the block enlargement optimization may have a negative effect on icache hit rate. Each time a block is combined with its successors, a separate copy of that block is created for each successor. In figure 4.1, combining block B with it's successors C and D resulted in the creation of an extra copy of B. This duplication may increase the number of icache capacity misses during program execution and lower performance. This will be the case if all the enlarged blocks formed from combining the block with its successors are accessed with sufficient frequency. On the other hand, if an enlarged block is never accessed, then it is never brought into the icache. The duplication incurred by such a block has no effect on the icache miss rate or the memory bandwidth used by the icache. The block enlargement optimization must be controlled so that the degree of block enlargement is balanced against the increase in the program's icache space requirements.

4.1.4 Simulation Concerns

As mentioned in chapter 3.3, for a simulator to accurately model branch misprediction penalties for a processor implementing a block-structured ISA, the simulator must take into account the effect of wrong path instructions. The instructions that compute a fault operation's condition may be in the same block as the fault itself. If such a fault is mispredicted, the only way to determine the amount of time required to resolve the fault is to issue the atomic block within which it resides and this block, by definition, is on the wrong path. Consider the enlarged block control flow graph in figure 4.2 on page 4.2. Suppose the correct successor to block A is block BD, but the simulated branch predictor has mispredicted the successor block to be block BC, that is, the fault operation in block BC has been mispredicted. This misprediction will be resolved when block BC's fault operation is executed and thus block BC must be issued into the simulated machine. To correctly model the machine's behavior in this situation, the instruction trace generated by BlockSim's front end will include the instructions in block BC that originally came from block B. The simulator will then stall instruction fetch until the mispredicted fault operation is resolved. Because BlockSim's front end does not proceed any further down the wrong path than the first block, the effect of executing instructions from the wrong path is only partially modeled by BlockSim.

4.2 Specification of a Block-Structured ISA

4.2.1 The Base ISA

To explore the performance advantages of block-structured ISAs, I have defined a specific block-structured ISA that incorporates a subset of the features described above. This ISA's architectural unit is the atomic block. The operations that can be found in an atomic block were taken from a subset of the MC88000 ISA [37]. All the non-control flow operations and indirect branches were taken directly from the MC88000 ISA. Added to this core set of operations were the trap, fault, and subroutine call operations. Each atomic block can contain up to sixteen operations, the issue width of the machine [1]. In addition, each atomic block must contain exactly one trap, subroutine call, or indirect branch operation to specify the control flow successor for that block.

4.2.2 Trap Operations

The trap operation has five fields: the trap opcode, the condition code register, the first target, the second second target, and the target count. As discussed above, the set of possible control flow successors for an atomic block can be partitioned into two subsets (the taken and not taken subsets) such that the trap condition specifies the subset that contains the correct control flow successor. The trap condition is the bit in the condition code register that is specified by the trap opcode. The first target of the trap operation is one of the targets from the taken subset. The second trap target is one of the targets from the not taken subset. The target count field of the trap operation specifies the log of the total number of control flow successors for the trap operation's atomic block. This information is used by the dynamic branch predictor as will be explained in chapter 6.

Because the trap operation only specifies two targets, atomic blocks that have more than two control flow successors and end in a trap operation cannot have all their control flow successors explicitly specified. The targets in the taken and not taken subsets that are not specified by the trap operation are implicitly specified by the fault operations within the control flow successors. Referring back to figure 4.2 on page 4.2, block A has three control flow successors, blocks BC, BD, and E. Only blocks BC and E are explicitly specified by the trap operation at the end of block A. Block BD is implicitly specified as a successor to block A by the fault operation within block BC. As a result, when a processor first encounters an atomic block, the processor cannot determine all the possible control flow successors to that block. The processor incrementally learns about the implicitly specified control flow successors when a fault operation is mispredicted which results in the mispredicted fault operation redirecting the control flow to its target.

A key difference between trap operations and conventional ISA branch instructions is as follows: trap operations explicitly specify two targets while branch instructions explicitly specify only one target. The second target of a branch instruction is implicitly specified as the instruction that follows the branch. The trap operation was defined to have two explicit

[1] The maximum atomic block size does not necessarily have to be restricted to the issue width of the machine if the machine has the microarchitectural support to issue atomic blocks that require more than one cycle to issue (e.g. the scratch pad alias table [52]). This dissertation does not investigate such machines.

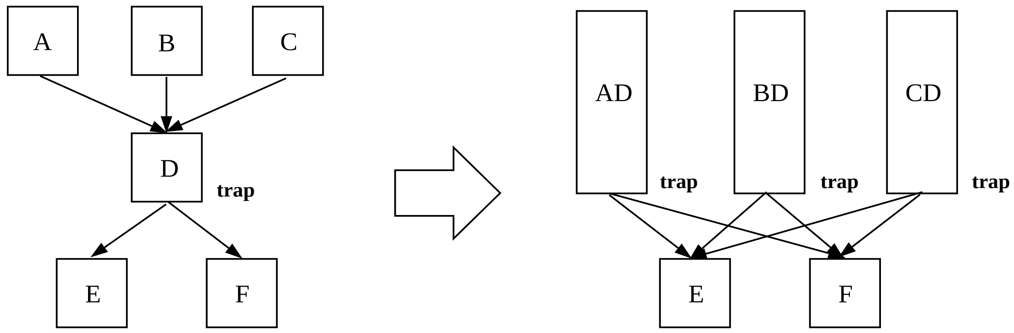

Figure 4.5: The fall through trap target problem for block-structured ISAs.

targets instead of one explicit target and one implicit fall-through target because the block enlargement optimization could create control flow graphs for which it was impossible to place one of an atomic block's targets after that atomic block to serve as the fall-through target. Figure 4.5 gives an example. Enlarged blocks AD, BD, and CD all have blocks E and F as their successors. Regardless of how the blocks are ordered, at least one of those three enlarged blocks will have neither blocks E or F as its fall-through target. This problem could be solved by creating a new block G that is either a duplicate of block E (or F) or an unconditional jump to block E (or F). Block G could then serve as the fall-through target for the third enlarged block. However, these solutions may reduce the instruction fetch rate by either reducing the icache hit rate because of the extra code duplication or reducing the average block size because of the extra unconditional jump inserted into the dynamic instruction stream. By explicitly specifying two targets for each trap operation, the block-structured ISA is able to eliminate the problem.

4.2.3 Fault Operations

The fault operation has four fields: the fault opcode, the condition code register, the target, and the target index. Based on its opcode, the fault operation checks a specific bit in its condition code register. If that bit is set, it will remove its associated atomic block from the machine and redirect the control flow to its target. As shown in figure 4.2, in the event that the branch predictor has incorrectly predicted block BC to be the successor to block A when block BD is the correct target, the fault operation in block BC will remove block BC's instructions from the machine and redirect the control flow to block BD. The fault operation also specifies an index that is associated with its target. This information is used by the dynamic branch predictor as will be explained in chapter 6.

4.2.4 Subroutine Calls

The subroutine call operation has three fields: the opcode, the call address, and the return address. The subroutine call directs the instruction stream to the specified call address and saves the specified return address in the return address register. The key difference between this subroutine call and that of a conventional ISA is that this subroutine call explicitly specifies the return address to be used after the called function completes. In a

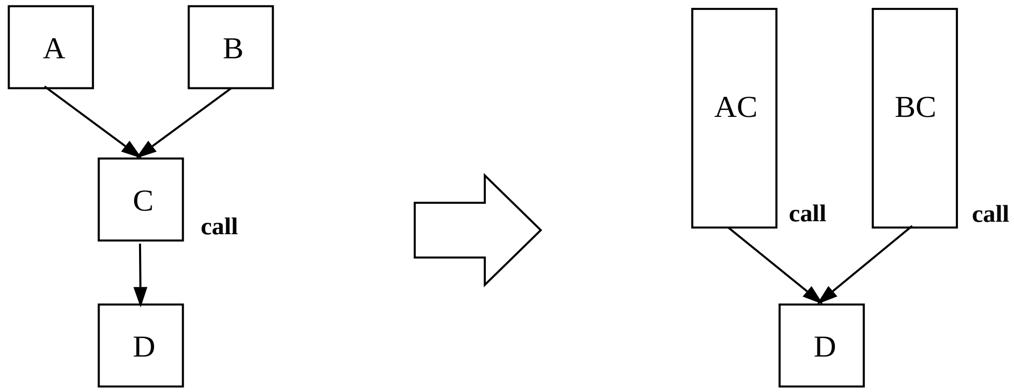

Figure 4.6: The fall through return address problem for block-structured ISAs.

conventional ISA, the return address is implicitly specified as the address that follows the call. The reason why the return address is explicitly specified in the block-structured ISA subroutine call operation is the same reason why the trap operation specifies two targets: the block enlargement optimization can create control flow graphs for which it is impossible to place a subroutine call's return address block after the call's block. Figure 4.6 gives an example. Block C ends with a subroutine call and as a result, blocks AC and BC also end in a subroutine call. Clearly, block D, the return address for the call, cannot be the fall-through block for both blocks AC and BC. By specifying an explicit return address, the block-structured ISA is able to eliminate this problem as well.

CHAPTER 5

Block Enlargement — Measurements and Analysis

This chapter examines how to most effectively apply the block enlargement optimization to increase block size and performance. Branch prediction effects are ignored. They will be examined in the chapters 6 and 7. Perfect branch prediction is assumed for all the experiments presented in this chapter.

5.1 The Base Block Enlargement Optimization

The block enlargement optimization implemented in the block-structured ISA compiler attempts to combine as many different combinations of blocks as possible. The compiler begins with the first block of each function and traverses the control flow graph in a breadth-first order combining blocks until one of the termination conditions listed below is met. The process is recursively repeated with the successor blocks of the newly formed enlarged block. The five termination conditions for the enlargement process are:

1. Blocks can continue to be enlarged until further enlargement would cause the size of the enlarged block to exceed processor issue width. As discussed in chapter 4.2, the maximum block size is restricted to the issue width so as to avoid the complexity of supporting atomic blocks that require more than one cycle to issue. For our experiments the maximum block size will always be sixteen.

2. Each block can contain at most two fault operations, which restricts the number of successor blocks for each block to at most eight. This restriction is due to branch prediction considerations (see chapter 6).

3. Blocks that end in a call cannot be combined with their successors. The interprocedural analysis required to combine such blocks has not been implemented.

4. Blocks that end in a return or an indirect jump cannot be combined with their successors. Because blocks that end in indirect branches may have an extremely large number of successors, combining such blocks with their successors would lead to unacceptable levels of code duplication[1]

[1]The block enlargement optimization could be extended in the future to relax this constraint by allowing

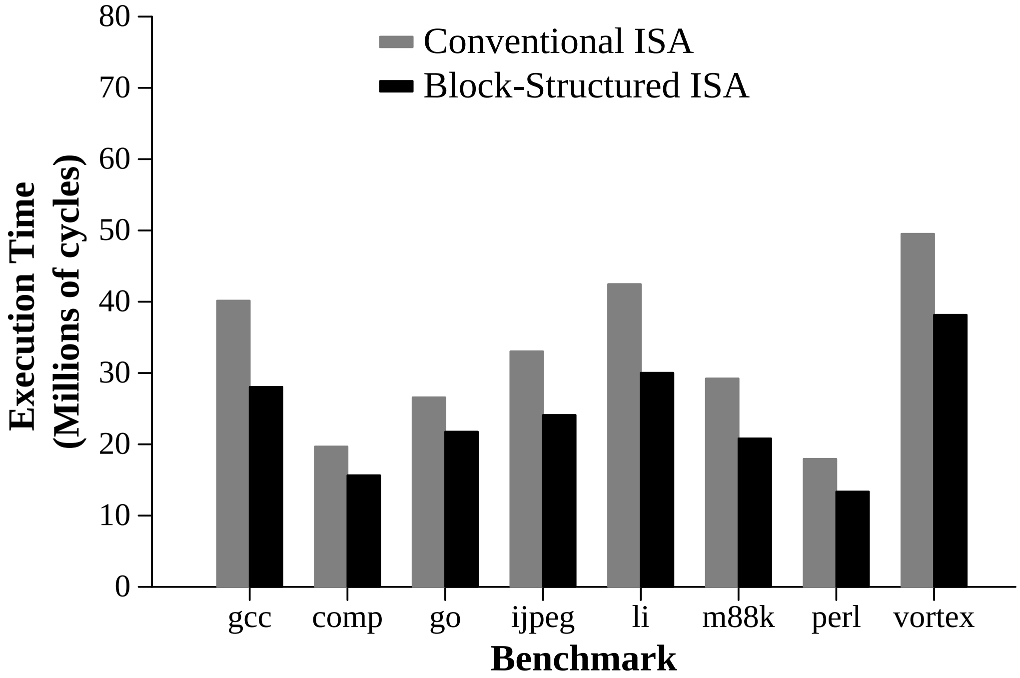

Figure 5.1: Performance comparison of block-structured ISA executables to conventional ISA executables.

5. Blocks that are the targets of a return or an indirect jump cannot be combined with any other blocks. This restriction is also due to branch prediction considerations (see chapter 6).

In addition, some of the C library routines called by the benchmarks were not recompiled with the block enlargement optimization. This was due to unavailability of the source code for the library routines. However, all the library routines that have a significant impact on the overall execution time were compiled.

After the block enlargement pass, the compiler executes another local optimization pass over the enlarged blocks to further improve the quality of code. This second local optimization pass will exploit the new opportunities for optimization that were created by block enlargement (see chapter 4.1.2).

Using the block enlargement optimization described above, the compiler generated block-structured ISA executables for the eight SPECint95 benchmarks. The performance of these executables running on the sixteen wide issue HPS processor described in chapter 3.2 is compared to the performance of the corresponding conventional ISA executables running on an identically configured HPS processor. Figure 5.1 shows the total number of cycles required to execute each benchmark from the two sets of executables. The block-structured ISA executables achieved an average reduction in execution time of 25%.

---

blocks that end in indirect branches to be combined with a small subset of their successors that occur frequently in the dynamic instruction stream.

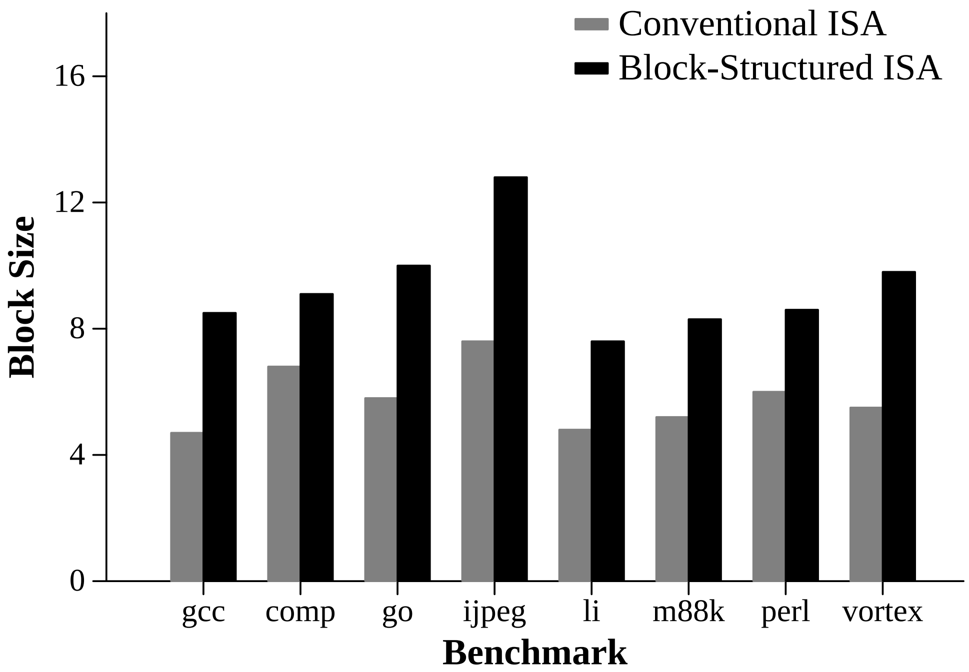

Figure 5.2: Average block sizes for block-structured and conventional ISA executables.

Figure 5.2 compares the average block size for the block-structured ISA executables to that of the conventional ISA executables. Using the block enlargement optimization, the block-structured ISAs increased the average block size by 62% from 5.8 to 9.3 operations.

By increasing the average block size and thereby increasing the instruction fetch rate, the block enlargement optimization is able to improve overall performance. However, the 62% increase in block size does not directly translate into a 62% increase in performance. Two other factors, full-window stalls and icache misses, reduced the performance gains provided by the increased block size.

Figure 5.3 shows the number of full-window stall cycles that occurred during the execution of each benchmark. As discussed in chapter 3.2, the processor stalls the instruction fetch mechanism whenever all the checkpoints within in the machine are taken. This stall continues until one of the checkpoints is freed up. For many of the benchmarks, the number of cycles stalled doubles. For vortex, it increases by an order of magnitude from one to ten million cycles. The block enlargement optimization increases the number of full-window stalls because it enables the processor to issue instructions at a faster rate. This increase in instruction issue rate increases the number of instructions associated with each checkpoint which in turn increases the amount of time needed to retire the checkpoint. Furthermore, the increase in instruction issue rate increases the average length of time an instruction must wait for its dependencies to resolve. Because the block enlargement optimization combines separate blocks into a single block, two instructions that belong to different blocks (and would have been issued in different cycles for a processor implementing a conventional ISA) may now be issued in the same cycle. If the two instructions have a data dependency

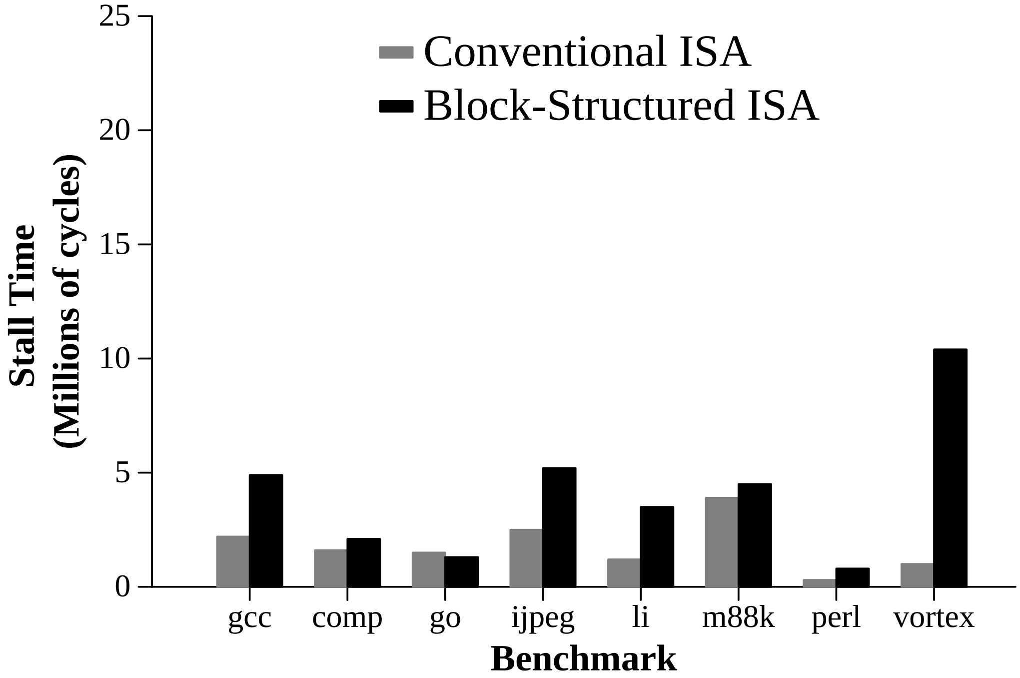

Figure 5.3: The number of cycles that instruction fetch was stalled due to a full-window.

between them, then any difference in their issue times would mask some of the latency of resolving that dependency. By eliminating the difference in their issue times, the block enlargement optimization increases the amount of time an instruction must wait for its data dependencies to resolve. This effect, in turn, further increases the time required to retire checkpoints. Although the increase in full-window stall cycles is a direct consequence of the block enlargement optimization, this increase should not be considered a drawback of the block enlargement optimization. By removing the instruction fetch bottleneck, the block enlargement optimization has given the processor the opportunity to exploit a higher level of instruction level parallelism which resulted in the uncovering of another bottleneck: the finite size of the processor's instruction window.

As discussed in chapter 4.1.3, the code duplication caused by block enlargement may increase the icache miss rate. Figure 5.4 shows the number of cycles that the processor stalled the instruction fetch mechanism due to an icache miss. The icache used in this experiment was 128KB, four-way set associative. With the exception of the gcc and go benchmarks, the block-structured ISA executables show little increase in icache miss cycles as compared to the conventional ISA. The gcc and go benchmarks showed significant increases in icache miss cycles for two reasons. First, both benchmarks are large to begin with. For the conventional ISA executables, only the gcc, go, and vortex benchmarks show a significant number of icache misses, indicating that the benchmarks are a little bit too large for the 128KB icache. The code duplication incurred by the block enlargement optimization for these benchmarks will translate directly to increased icache misses. Second, the gcc and go benchmarks contain many unbiased branches. As a result, a larger number of the successor blocks for each

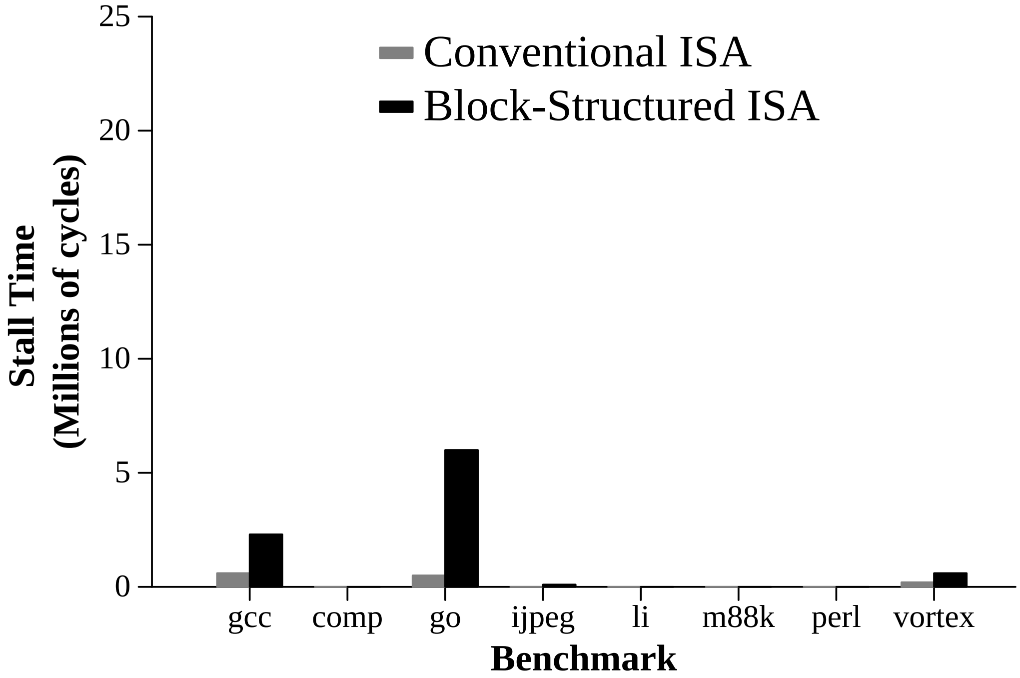

Figure 5.4: The number of cycles that instruction fetch was stalled due to an icache miss.

enlarged block in these two benchmarks will be accessed with significant frequency which in turn increases the number of duplicated blocks that must be held in the icache. The vortex benchmark, on the other hand, contains many highly-biased branches, reducing the number of duplicated blocks that must be held in the icache. As a result, the block enlargement optimization does not incur as severe a penalty for vortex despite the fact that vortex is a large benchmark.

To quantify the performance impact of the icache misses, figure 5.5 shows the execution times of the three block-structured ISA executables, gcc, go, and vortex, for a processor with a perfect icache. These three benchmarks were the only ones that had a non-trivial number of icache misses for the 128KB icache size. Only the go benchmark showed a significant slowdown due to the icache misses which was due to code bloat as discussed above. In addition, for all three benchmarks, the difference between the total execution time with the 128KB icache and the number of icache miss cycles was slightly smaller than the total execution time with the perfect icache. This small difference was due to the HPS model's ability to accomplish real work and make forward progress even in the face of icache misses.

5.2 ICache Performance Issues

As shown in figure 5.4, a 128KB icache is large enough to handle the code duplication incurred by the block enlargement optimization for the majority of the SPECint95 benchmarks.

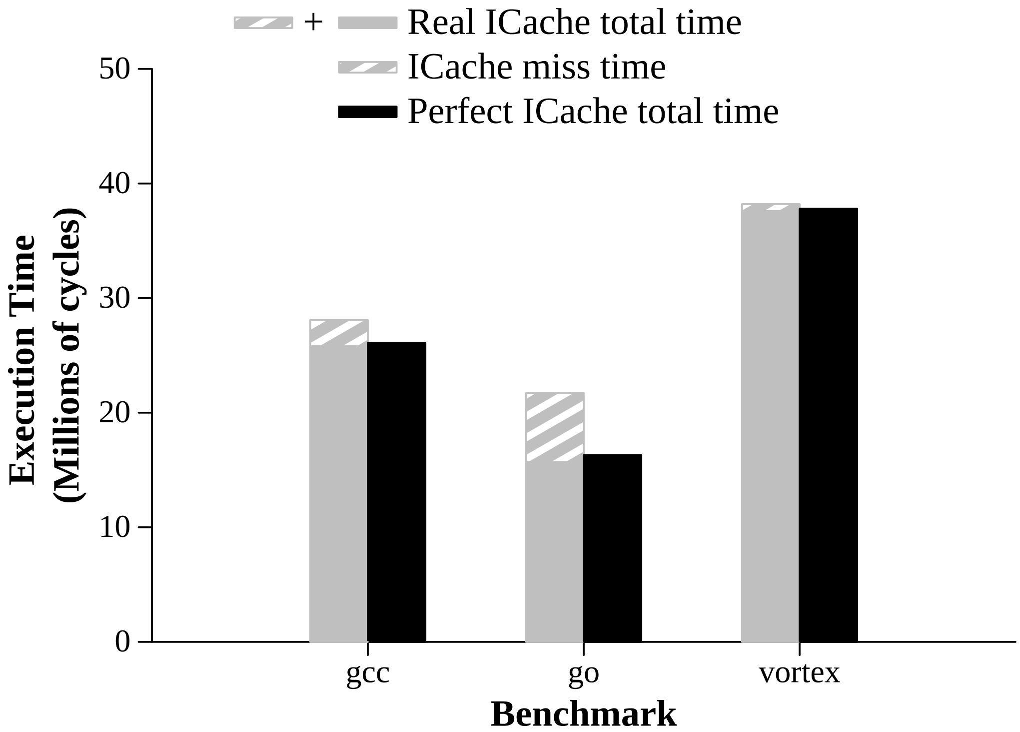

Figure 5.5: Execution times for a perfect and a 128KB icache.

This section quantifies the effect block enlargement optimization has on icache performance when the program executed just fits into the icache or is too large for the icache. This section also considers ways to control the block enlargement optimization so as too reduce the degree of code duplication without reducing overall performance.

To model icache performance when the executed program is too large for the icache, the SPECint95 benchmarks were simulated on processors with smaller icache sizes. These simulations measured the total execution time as well as the number of icache miss cycles. Figures 5.6–5.13 show these numbers for the block-structured and conventional ISA executables executing on their corresponding HPS implementations. Only the gcc, go, perl, and vortex benchmarks show significant a significant number of icache misses at the smaller icache sizes.

The block enlargement optimization's negative impact on icache performance can be reduced by reducing the amount of code duplication performed by the optimization. However, any reduction in code duplication will also reduce the average size of the enlarged blocks formed. Thus, any variation of the block enlargement optimization that attempts to reduce code duplication must balance the resulting icache performance gains against the performance lost due to decreasing the block size. This section considers two approaches that attempt to effectively exploit this tradeoff.

The first approach reduces code duplication by simply reducing the number of blocks that can be combined into a single enlarged block. This can be done by reducing the maximum number of fault operations allowed in each enlarged block from two to one. In effect, this restriction reduces the number of basic blocks that can be combined together from three to two.

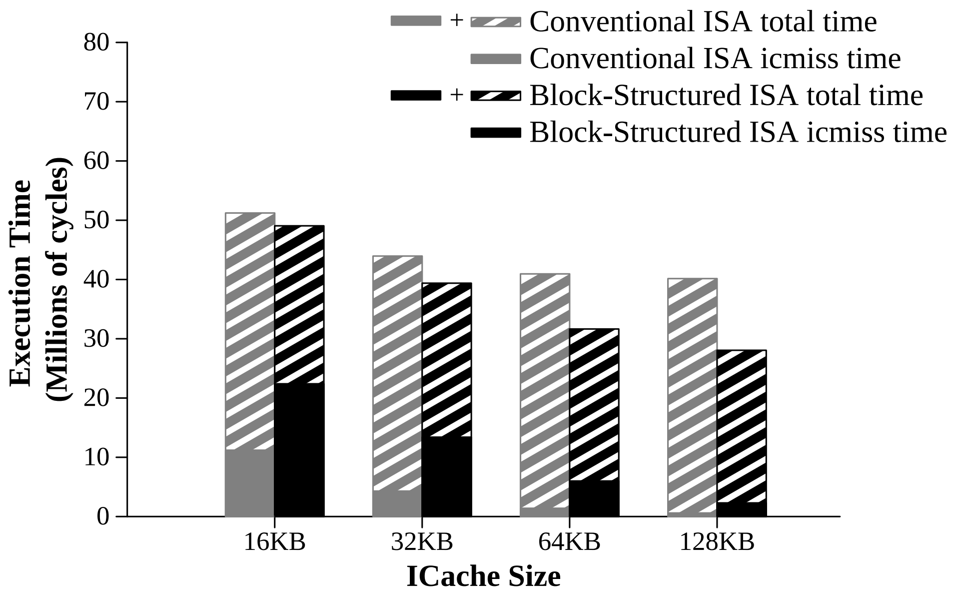

Figure 5.6: The number of icache miss and total execution cycles for the gcc benchmark.

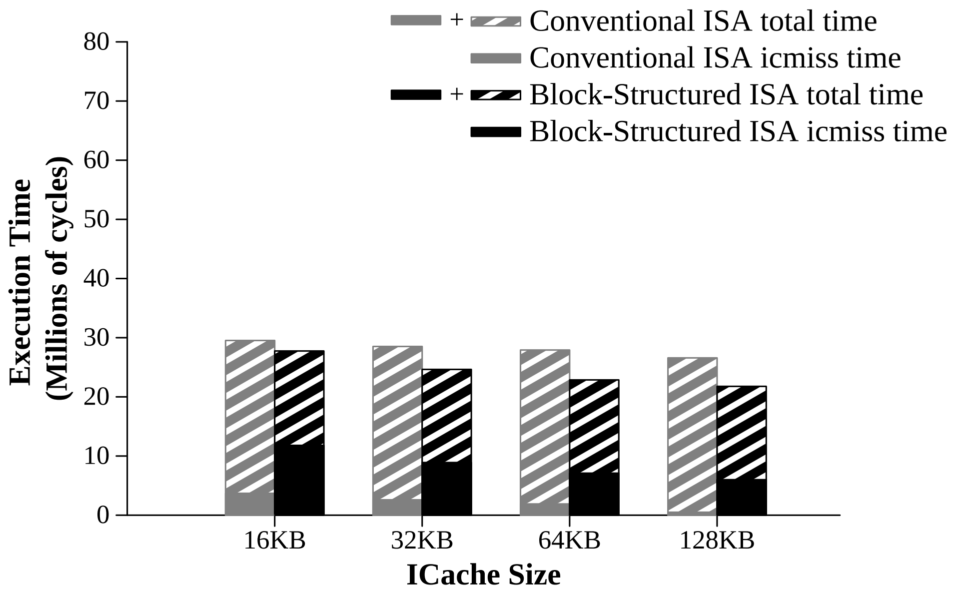

Figure 5.7: The number of icache miss and total execution cycles for the go benchmark.

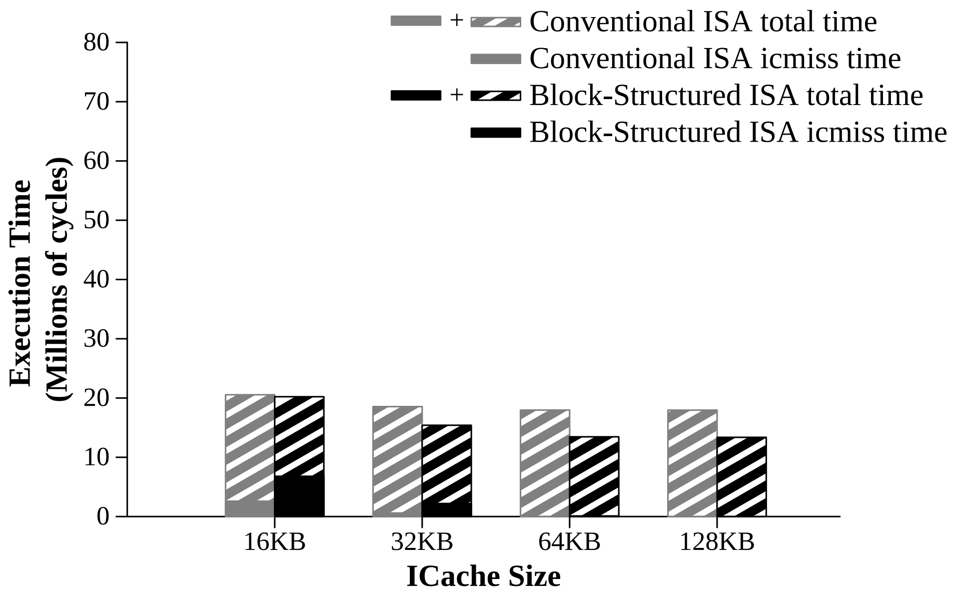

Figure 5.8: The number of icache miss and total execution cycles for the perl benchmark.

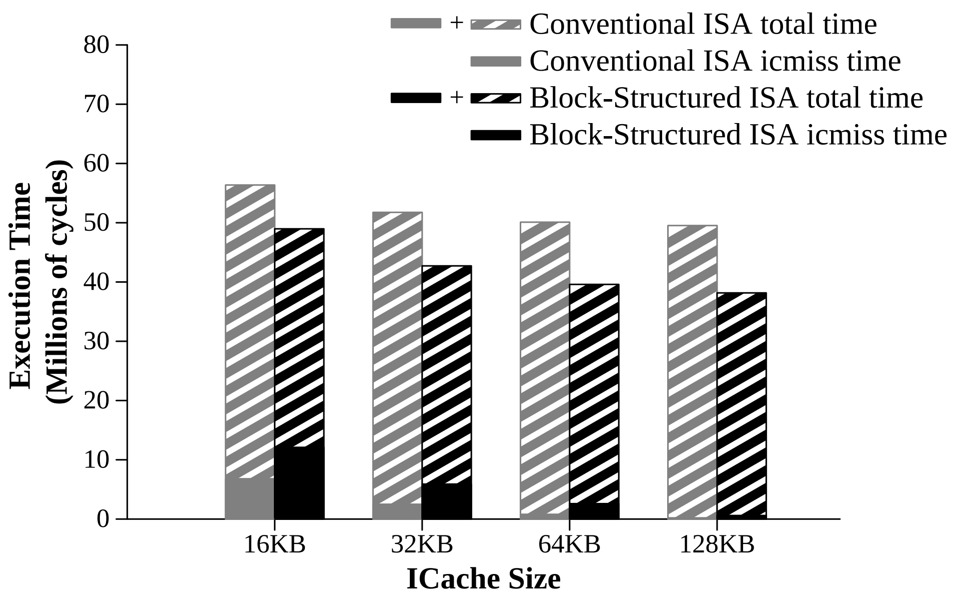

Figure 5.9: The number of icache miss and total execution cycles for the vortex benchmark.

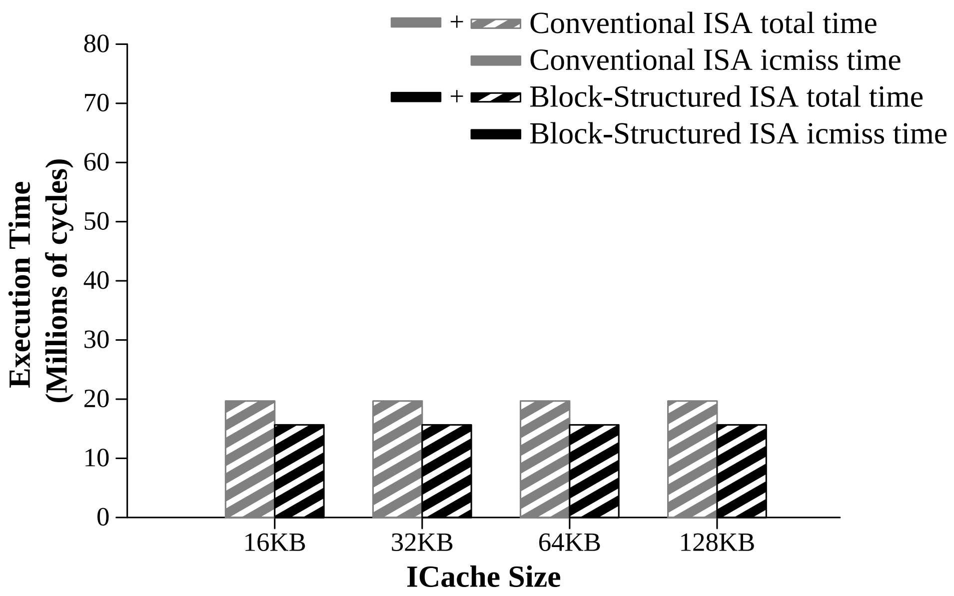

Figure 5.10: The number of icache miss and total execution cycles for the compress benchmark.

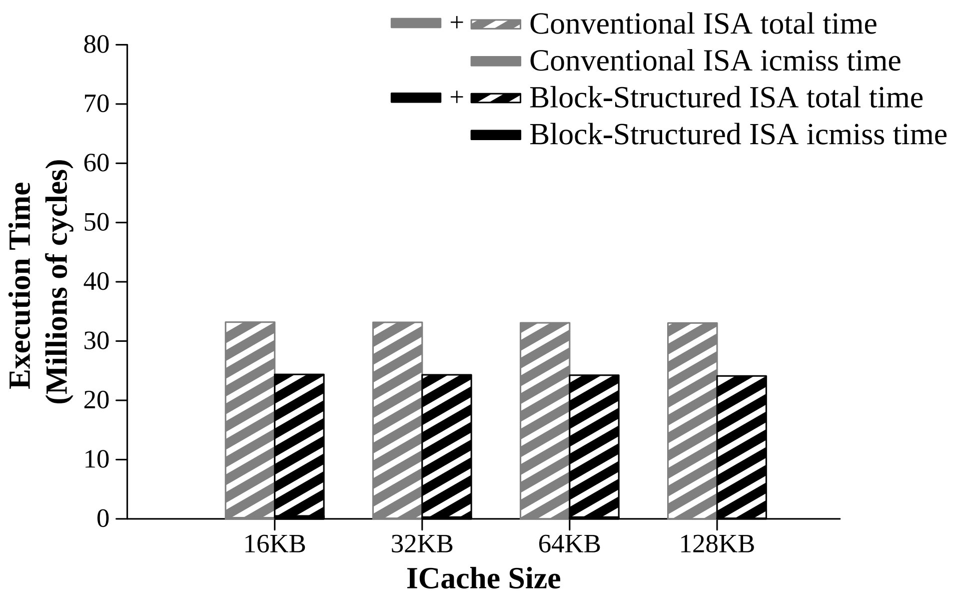

Figure 5.11: The number of icache miss and total execution cycles for the ijpeg benchmark.

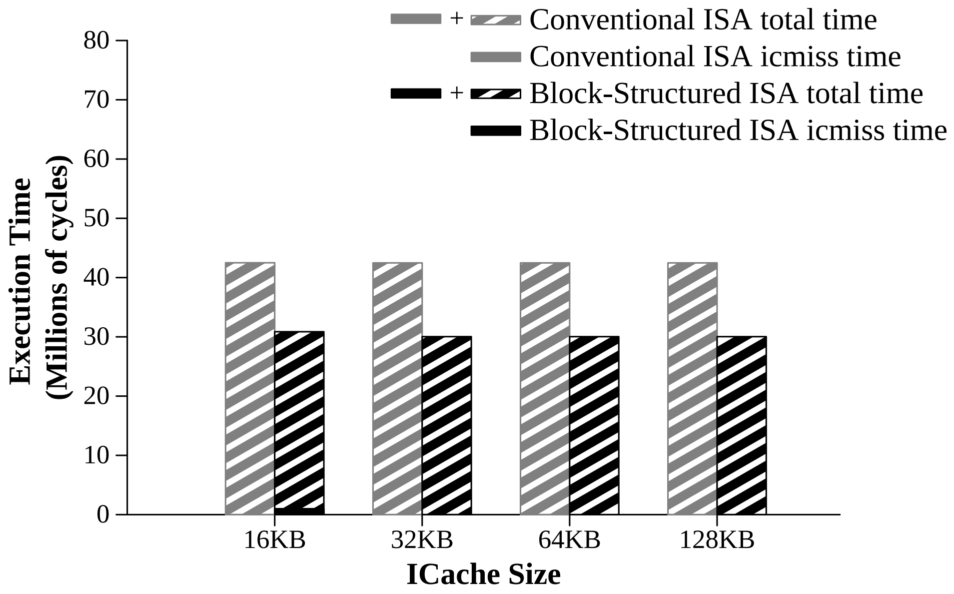

Figure 5.12: The number of icache miss and total execution cycles for the xlisp benchmark.

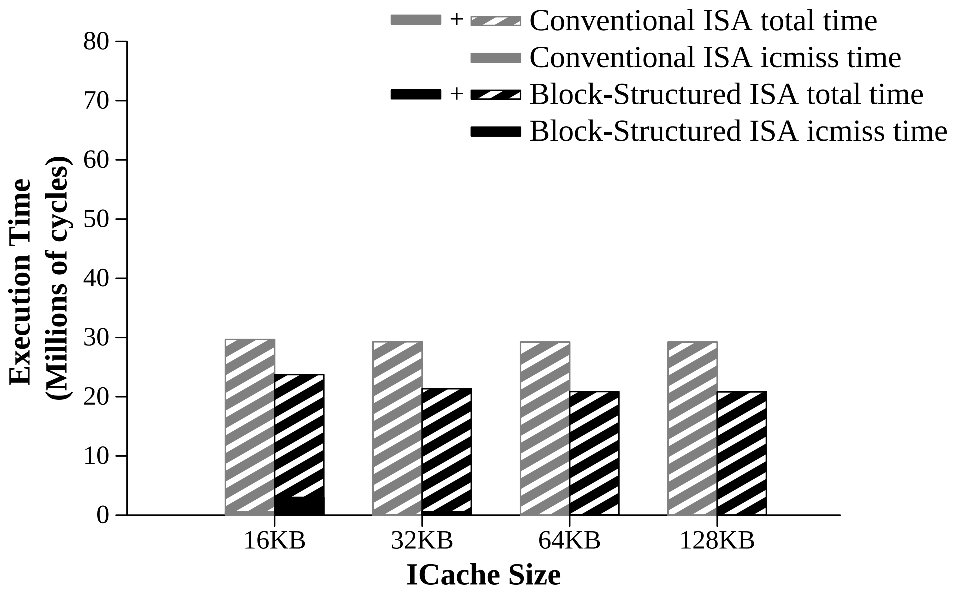

Figure 5.13: The number of icache miss and total execution cycles for the m88ksim benchmark.

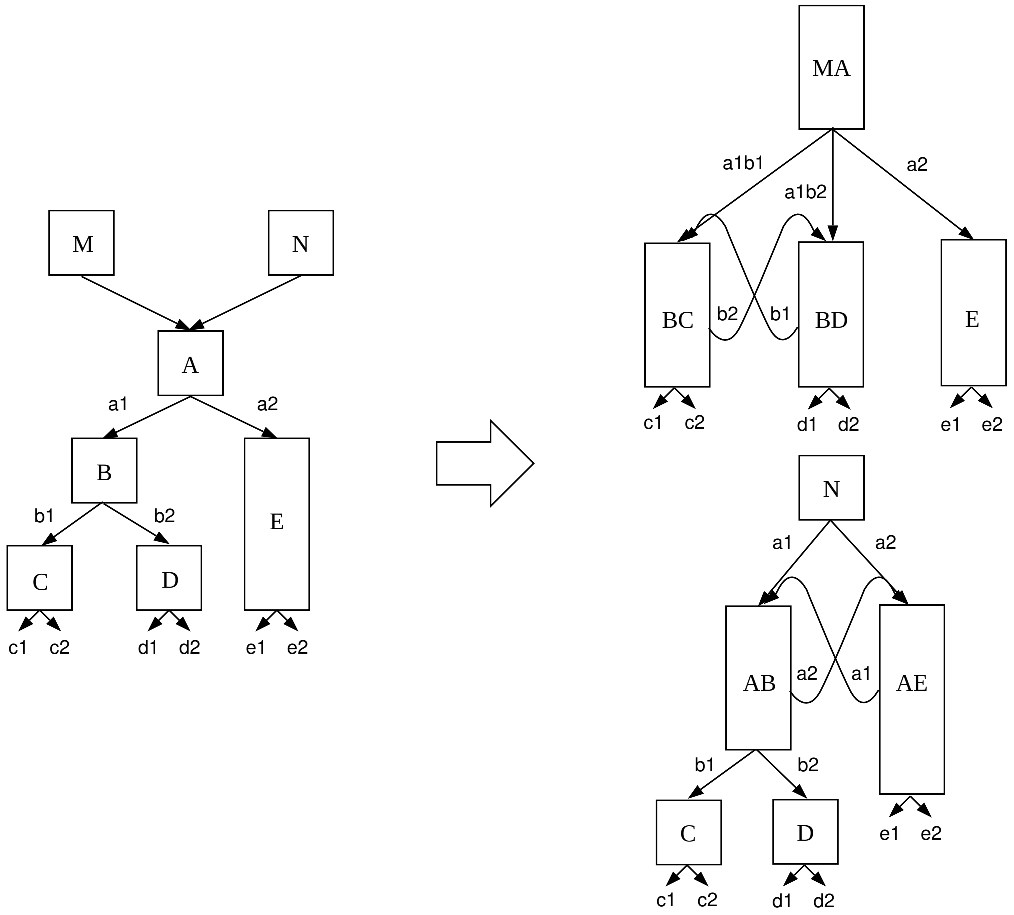

Figure 5.14: Forming overlapped enlarged blocks with the block enlargement optimization.

The second approach restricts the block enlargement optimization from making overlapping enlarged blocks. Two enlarged blocks are defined to be overlapping if the basic blocks that occur at the end of one enlarged block occur at the beginning of the other enlarged block. Figure 5.14 gives an example of overlapped enlarged blocks. The control flow graph on the right of figure shows the result of applying the base block enlargement optimization to the control flow graph on the left. Because block M could be combined with block A and block N could not be combined with block A, two overlapping enlarged blocks, MA and AB, were formed. Continuing the block enlargement optimization from block MA caused the formation of blocks BC and BD which overlap with block AB. As can be seen from the figure, once the block enlargement optimization's progress through the control flow graph gets out of phase as it does here, the formation of overlapped enlarged blocks will continue until some other block enlargement termination condition (i.e. a procedure call) is encountered.

By preventing the formation of overlapping enlarged blocks, the code duplication that occurs due to such blocks is eliminated. This restriction is implemented by adding two more termination conditions to the block enlargement process:

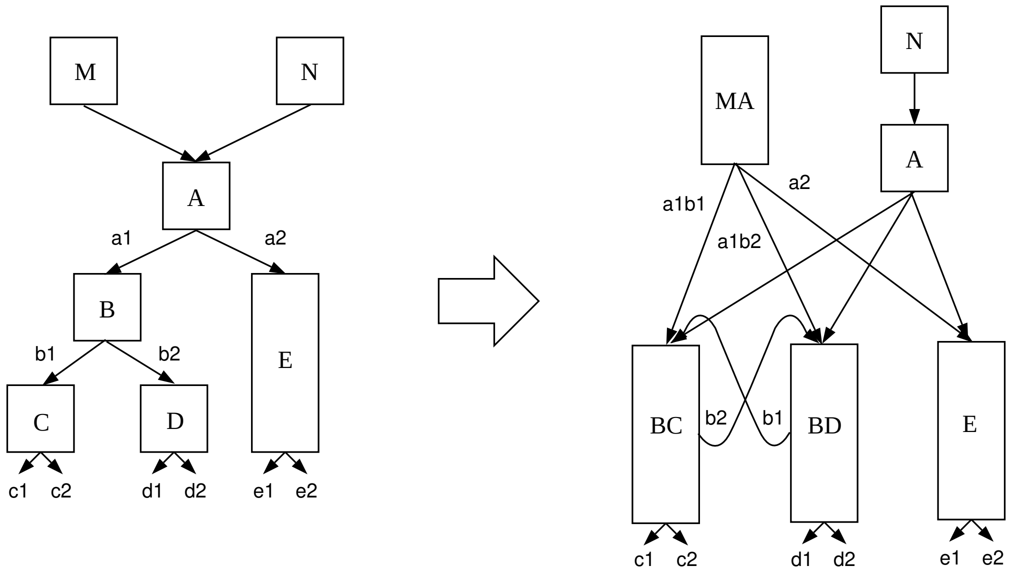

Figure 5.15: Restraining the block enlargement optimization to prevent the formation of overlapped enlarged blocks.

1. An enlarged block cannot be combined with its successor if its successor is the first block of some other enlarged block.
2. An enlarged block cannot be enlarged any further if its last basic block is the final basic block in some other enlarged block.

Figure 5.15 shows the application of the no overlap restriction to the control flow graph in figure 5.14. In this example, block A is prevented from combining with block B which prevents the formation of the overlapped blocks that occurred in figure 5.14.

By adding additional constraints to block enlargement, the no overlap restriction decreases the size of the enlarged blocks formed. To reduce the impact on block size, profile information is used to guide the application of the optimization. A profile is taken of the program to determine its most frequently executed paths. During the block enlargement optimization, this information is used to decide which blocks to enlarge. Rather than traversing the control flow graph in a breadth-first order, the block enlargement optimization enlarges the blocks along the most frequently executed program paths first. Because these blocks are enlarged first, the no overlap restriction does not apply to them. As a result, the reduction in block size due to the no overlap restriction will occur only for the less frequently executed enlarged blocks.

Figures 5.16–5.19 compares the performance of the three variations of the block enlargement optimization, base, one fault max, and no overlap, for the four SPECint95 benchmarks that showed significant numbers of icache misses. The no overlap variation was more effective than the one fault variation in reducing the number of icache miss cycles (an average of 27% versus 9% for the 16KB icache). Furthermore, for the 16KB icache, the no overlap variation reduced execution time by 7% while the one fault variation increased execution time by 3%.

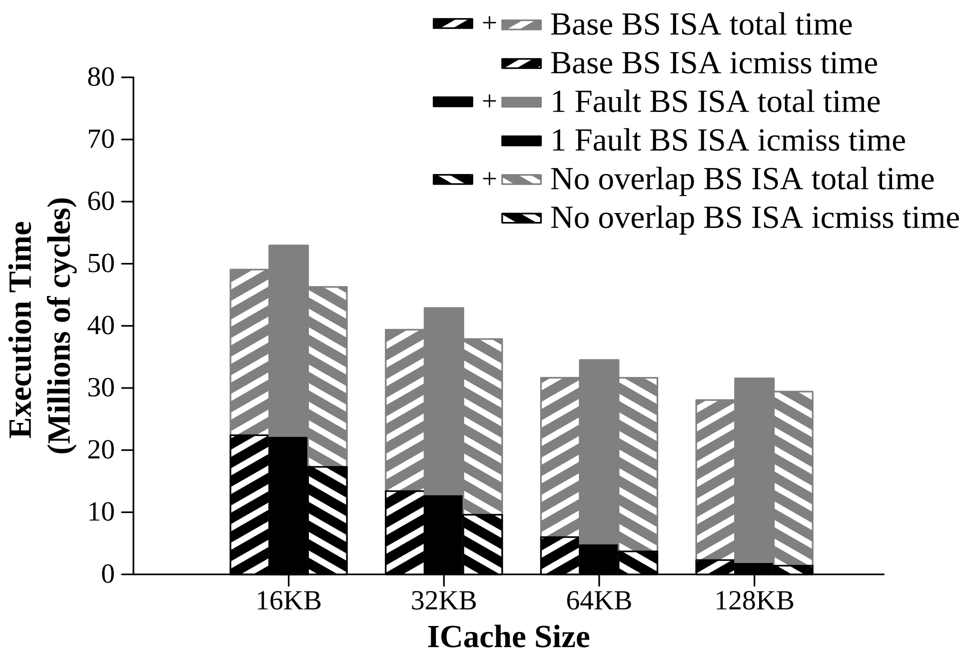

Figure 5.16: Comparing the base, one fault, and no overlap variations of the block enlargement optimization for the gcc benchmark.

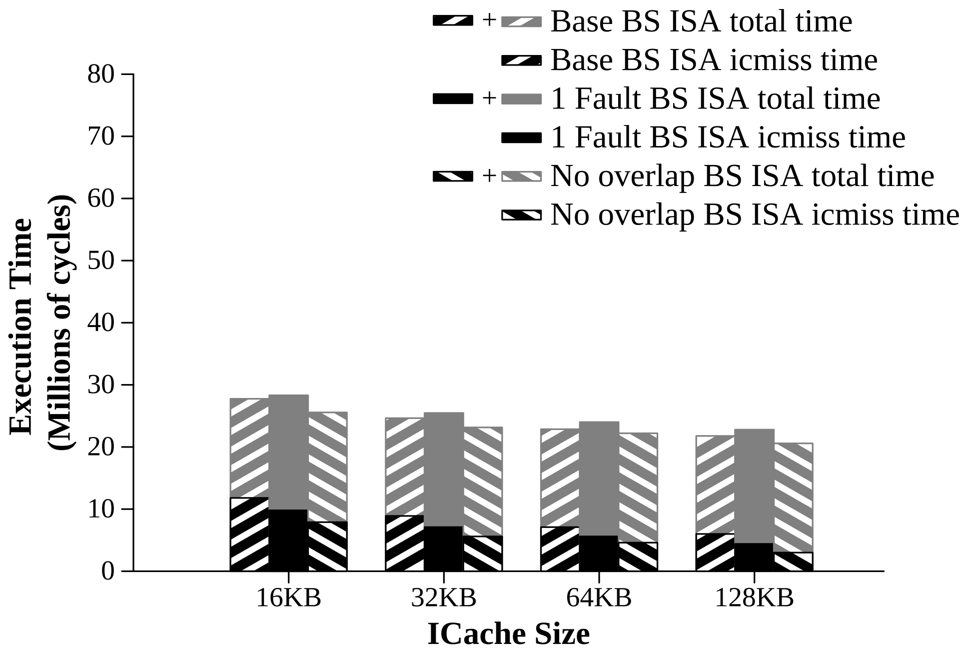

Figure 5.17: Comparing the base, one fault, and no overlap variations of the block enlargement optimization for the go benchmark.

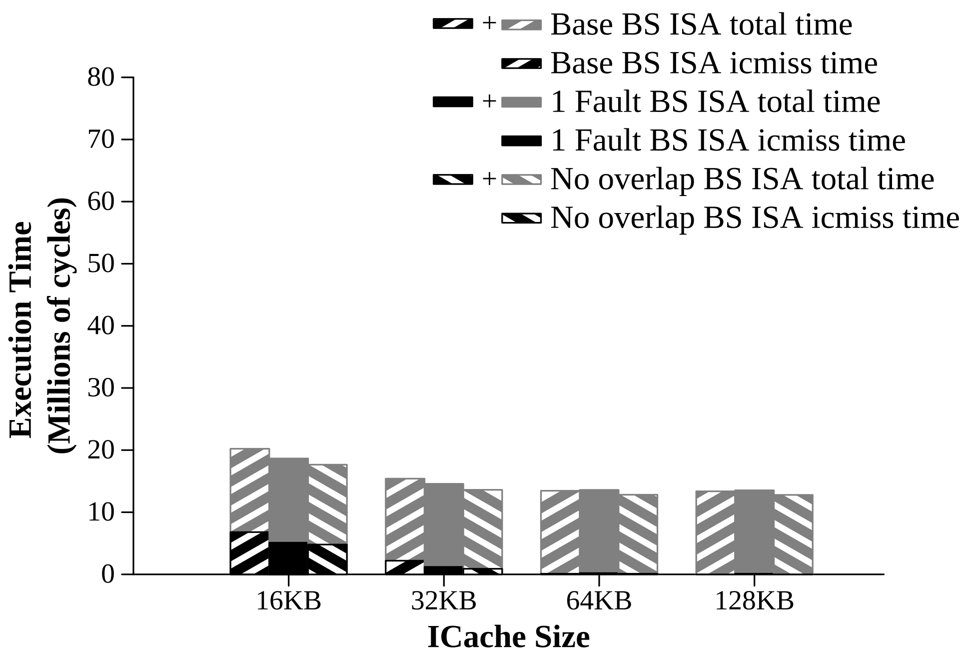

Figure 5.18: Comparing the base, one fault, and no overlap variations of the block enlargement optimization for the perl benchmark.

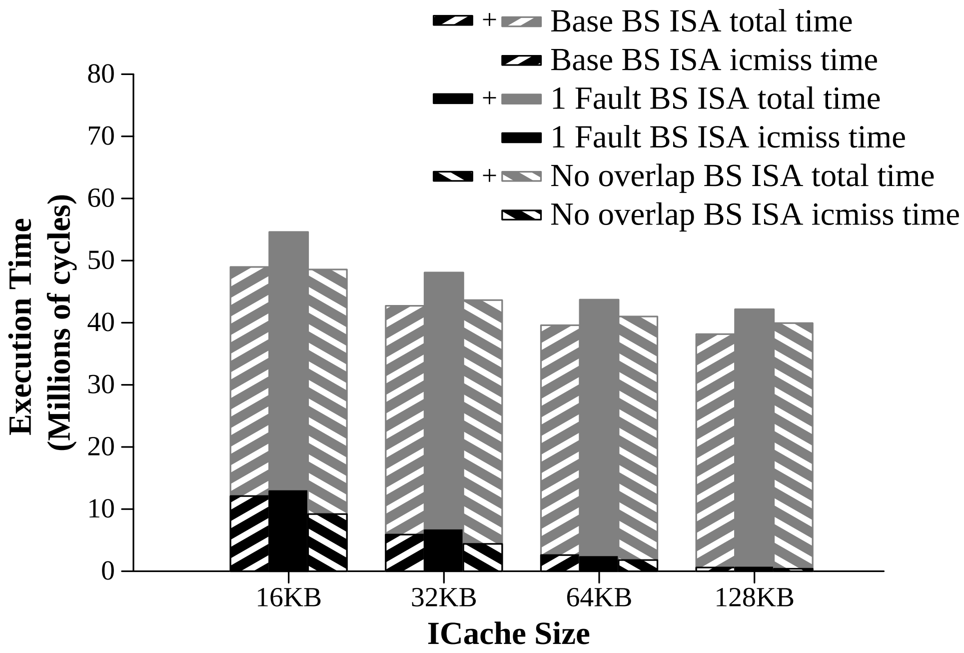

Figure 5.19: Comparing the base, one fault, and no overlap variations of the block enlargement optimization for the vortex benchmark.

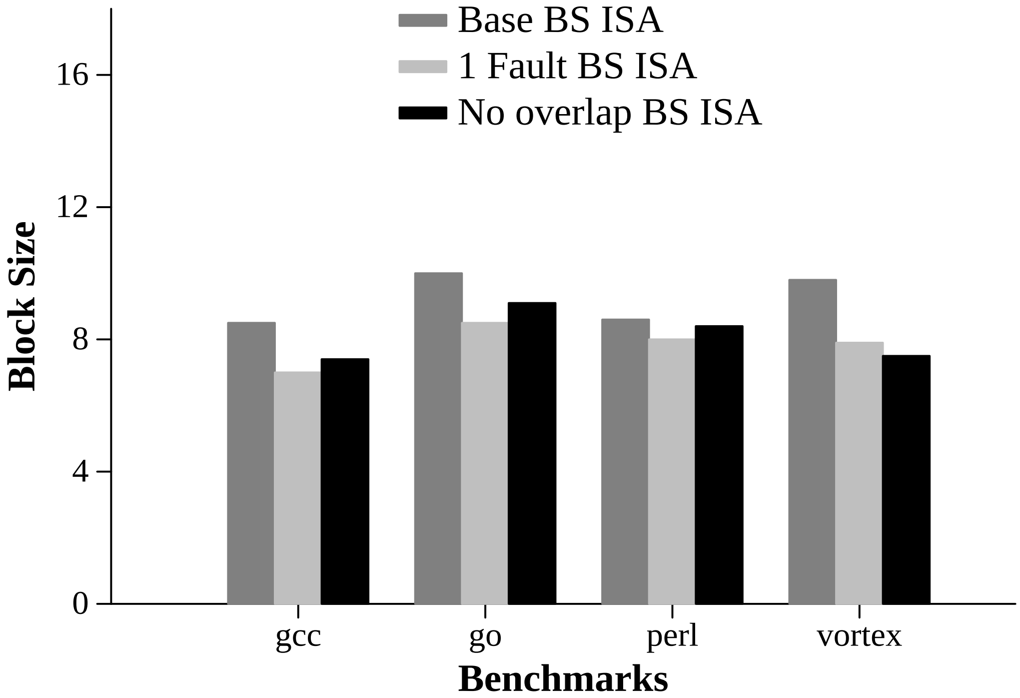

Figure 5.20: The average block sizes for the base, one fault, and no overlap variations of the block enlargement optimization.

Figure 5.20 shows the average block sizes for each of the variations. The no overlap variation decreased block size by 12% as compared to the base variation while the one fault variation decreased block size by 15%. Not only was the no overlap variation more effective than the one fault variation in reducing the icache miss penalty, but it was also able to form larger blocks. For the gcc and vortex benchmarks, as the icache size was increased and the impact of icache misses to the total execution time was reduced, this decrease in block size resulted in a performance slow down for the no overlap variation as compared to the base variation.

5.3 Block Enlargement Obstacles

Despite increasing the block size by 62% to 9.3 operations, the base block enlargement optimization still has almost half the processor fetch bandwidth left unused. This section examines why the block enlargement optimization was unable to create larger blocks.

Figures 5.21–5.28 show for each block size the dynamic frequency with which it occurred. The histogram bar for each block size is partitioned into the reasons why the blocks of that size could not be enlarged any further. There were five reasons for terminating block enlargement:

1. max size – the block could not be combined with any of its successors without exceeding the max size constraint.

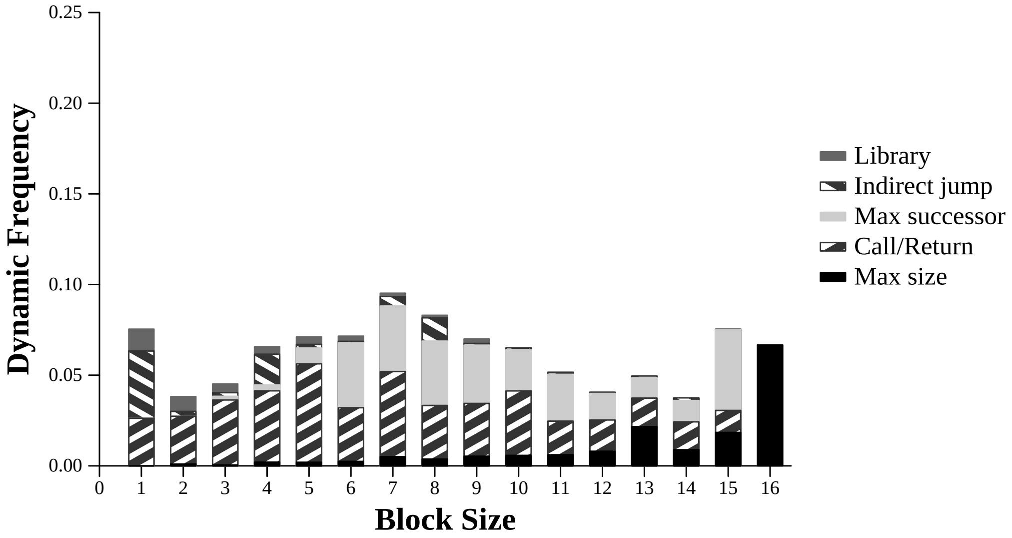

Figure 5.21: Block termination reasons for the gcc benchmark.

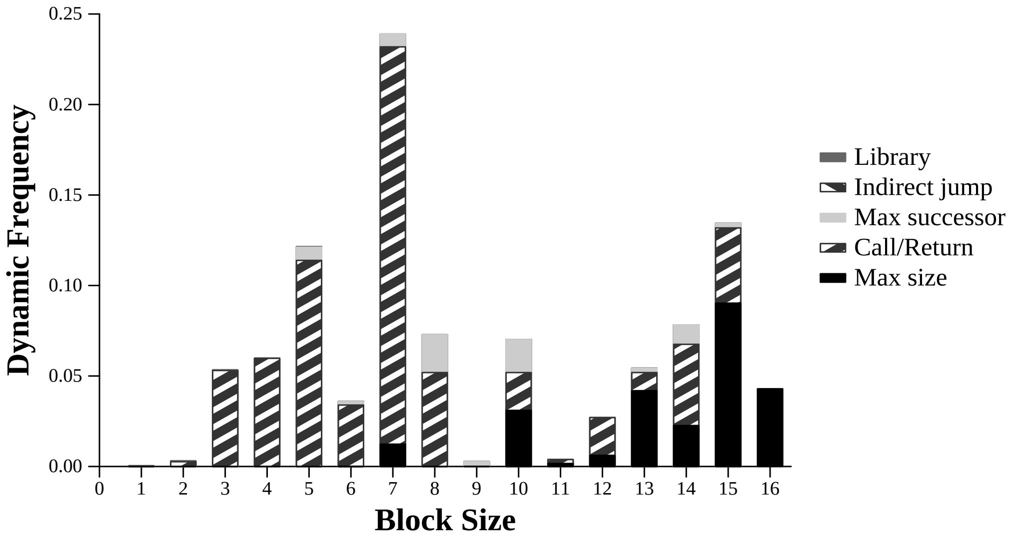

Figure 5.22: Block termination reasons for the compress benchmark.

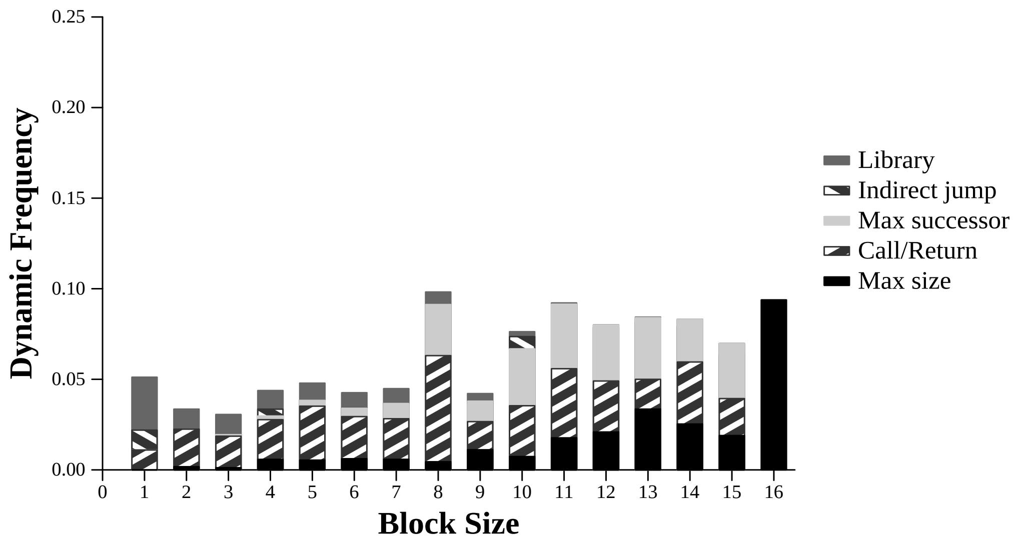

Figure 5.23: Block termination reasons for the go benchmark.

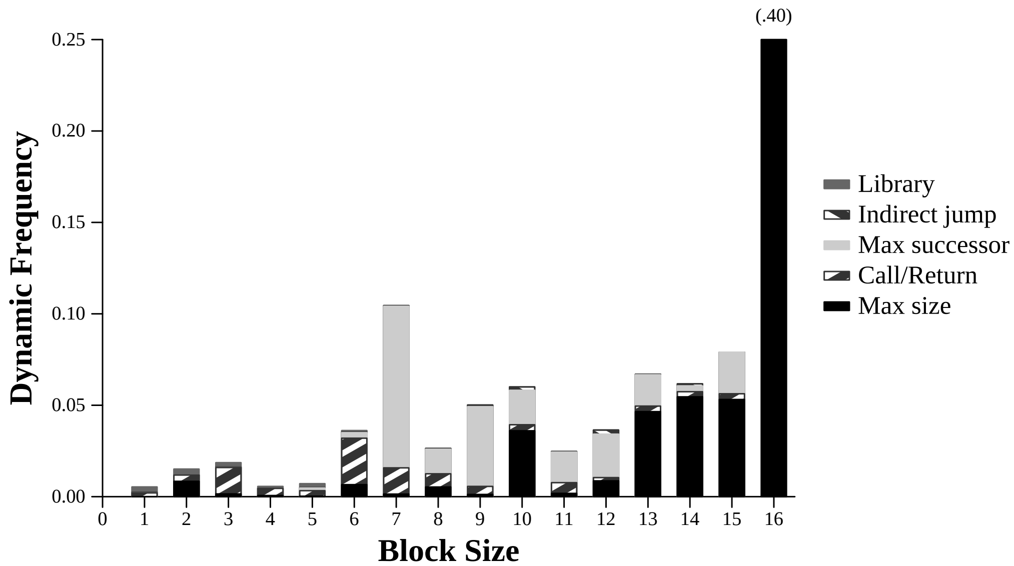

Figure 5.24: Block termination reasons for the ijpeg benchmark.

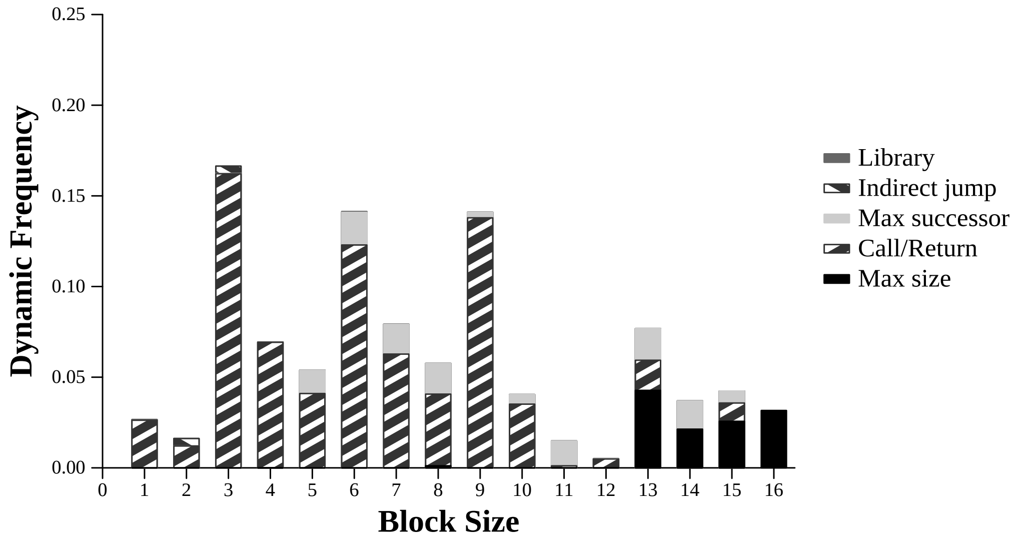

Figure 5.25: Block termination reasons for the li benchmark.

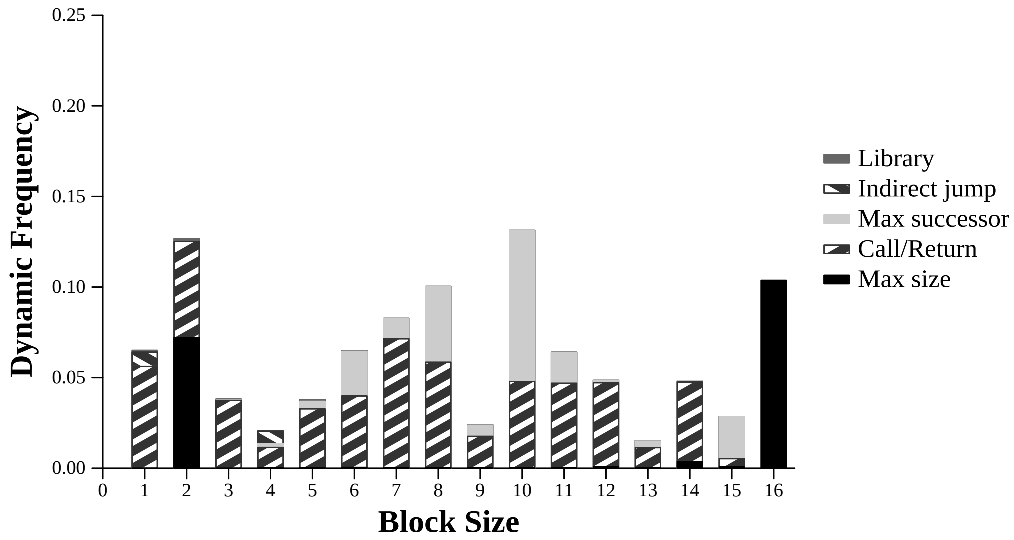

Figure 5.26: Block termination reasons for the m88ksim benchmark.

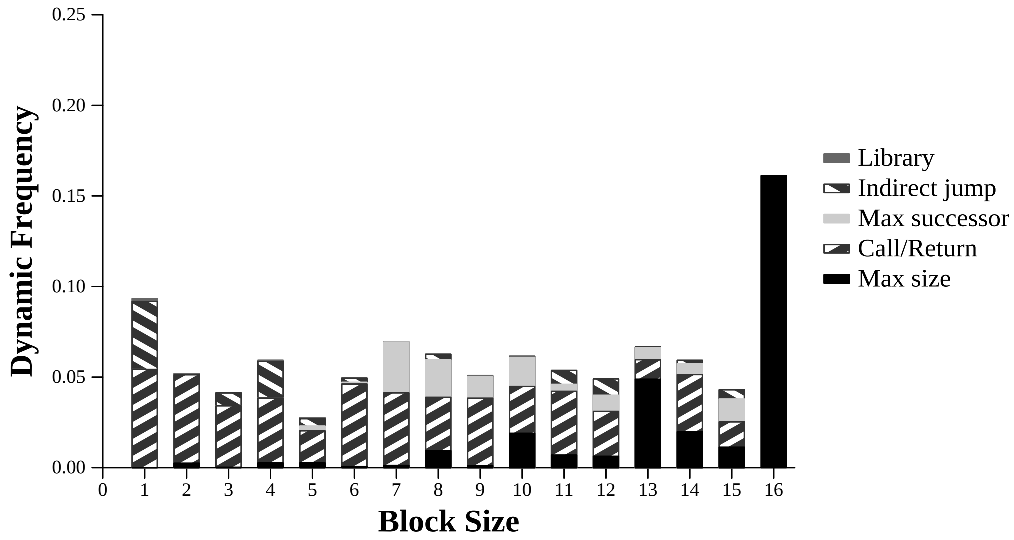

Figure 5.27: Block termination reasons for the perl benchmark.

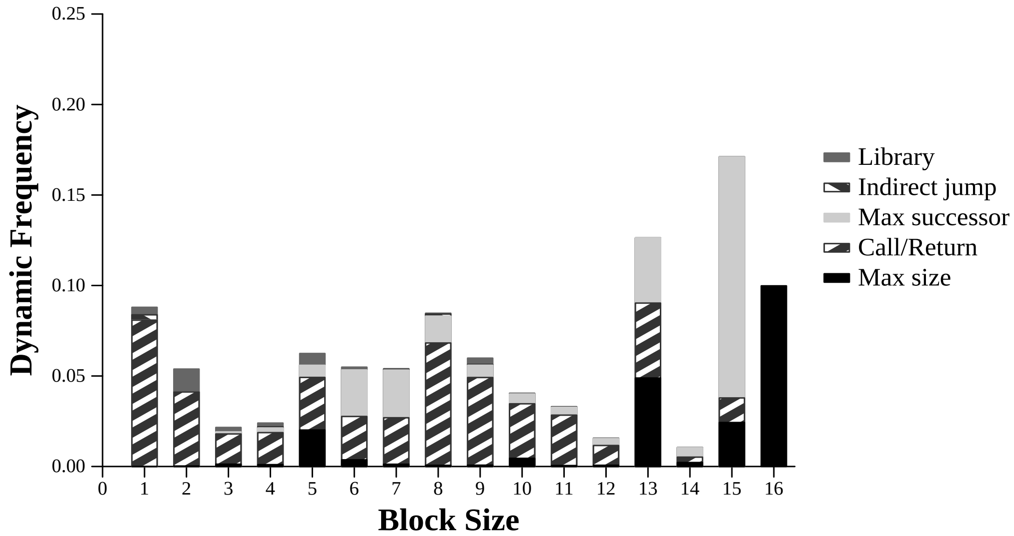

Figure 5.28: Block termination reasons for the vortex benchmark.

2. max successor – the block already contains two fault operations. Combining it with its successors would result in its parent block exceeding the max successor constraint.

3. call/return – the block cannot be enlarged any further because it either ends in a call or return operation or the block itself is the target of a call or return operation.

4. indirect jump – the block cannot be enlarged any further because it either ends in an indirect jump or the block itself is the target of an indirect jump.

5. library – the block belongs to a library routine that has not been compiled with the block enlargement optimization.

The figures show that the call/return constraint terminates the overwhelming majority of the enlarged blocks. The max successor and max size constraints terminate the majority of the remaining enlarged blocks. However, because the majority of the blocks terminated by the max size constraint are of size thirteen or larger, the max size constraint has a very small impact on performance. The indirect jump constraint causes a small number of enlarged block terminations in the gcc and perl benchmarks but is insignificant in the other benchmarks. The library routines that were not recompiled with block enlargement had an insignificant effect across all the benchmarks.

5.4 Function Inlining

This section studies the use of function inlining to overcome the call/return block enlargement constraint. Function inlining eliminates calls and returns in a program by replacing selected function calls with the bodies of the called functions. In addition to eliminating call and return instructions, function inlining also eliminates the extra code needed to set up the function call such as moves that set up the function's arguments and return values, loads and stores of caller and callee saved registers, and increments and decrements to the stack pointer. Furthermore, function inlining increases the scope with which the compiler can apply global optimizations by making the inlined function's code visible at the caller function's level. While the block-structured ISA compiler will exploit all these advantages, the key motivation for using function inlining is to eliminate calls and returns so that even larger blocks can be formed by the block enlargement optimization.

Because it duplicates the code for the inlined function, function inlining will increase the program size and more importantly, may decrease the icache hit rate. This optimization must be carefully controlled to ensure that the performance benefits of function inlining are not offset by increased icache miss penalties. To do this, only a select set of function calls (or call sites) within the program are chosen for function inlining. The block-structured ISA compiler uses a variation of the algorithm proposed by Hwu and Chang [21] to select the call sites to be inlined. The program to be compiled is first profiled to determine the dynamic frequencies of each call site within the program where dynamic frequency is defined to be the ratio of the number of times the call site occurred to the total number of blocks in the program. The call sites are then ordered according to frequency and are selected for inlining by the following conditions:

Benchmark	Number of Call Sites	Total Call Site Frequency	Inlined Code Size (bytes)
gcc	0	0.00%	0
compress	6	5.99%	1288
go	5	1.02%	2612
ijpeg	2	0.48%	624
li	27	10.76%	2500
m88ksim	19	5.32%	8448
perl	12	5.32%	2528
vortex	13	3.75%	3588

Table 5.1: Call sites that were selected for function inlining.

1. The called function cannot be recursive. It is impossible to completely inline a recursive function.

2. The dynamic frequency of the call site must be above the specified threshold. This guarantees that only the call sites that have a significant impact on performance will be considered for inlining.

3. The total amount of inlined code must not exceed the specified maximum size. This condition controls the amount of code duplication incurred by function inlining [2].

For the experiments here, the minimum dynamic frequency threshold was set to .001 and the maximum inlined code size was set to 16KB. Table 5.1 lists the number of call sites chosen for inlining for each benchmark along with the total dynamic frequency and inlined code size for the chosen call sites. Appendix D lists the locations of the chosen call sites.

Figure 5.29 compares the performance of block-structured ISA executables compiled with function inlining to the performance of those compiled without function inlining. The average improvement in execution time is 6.3%. The benchmarks that had the most significant improvements were li, perl, and compress with improvements in execution times of 20%, 15%, and 9%. The gcc benchmark showed no difference because nothing was inlined. The go benchmark showed a slight decrease in performance, about 1%, due to the increased icache miss rate offsetting the benefits of the inlined functions.

Figure 5.30 compares the average block size of block-structured ISA executables compiled with function inlining to those compiled without function inlining. The average block size increased from 9.3 to 10.2. There are two interesting points in this figure. First, the compress benchmark had the largest increase in block size, going from 9.1 to 12.4. This increase in block size translated to only a 9% decrease in execution time because the full window stalls for the inlined version of compress increased significantly from two million cycles to six million cycles. The full window stalls for the compress benchmark accounted for over 40% of the

[2] A better heuristic would have constrained the total amount of inlined code based on the icache requirements of the original program. Programs that use a small fraction of the total icache can afford greater amounts of inlined code than programs than programs that use a large fraction of the total icache space.

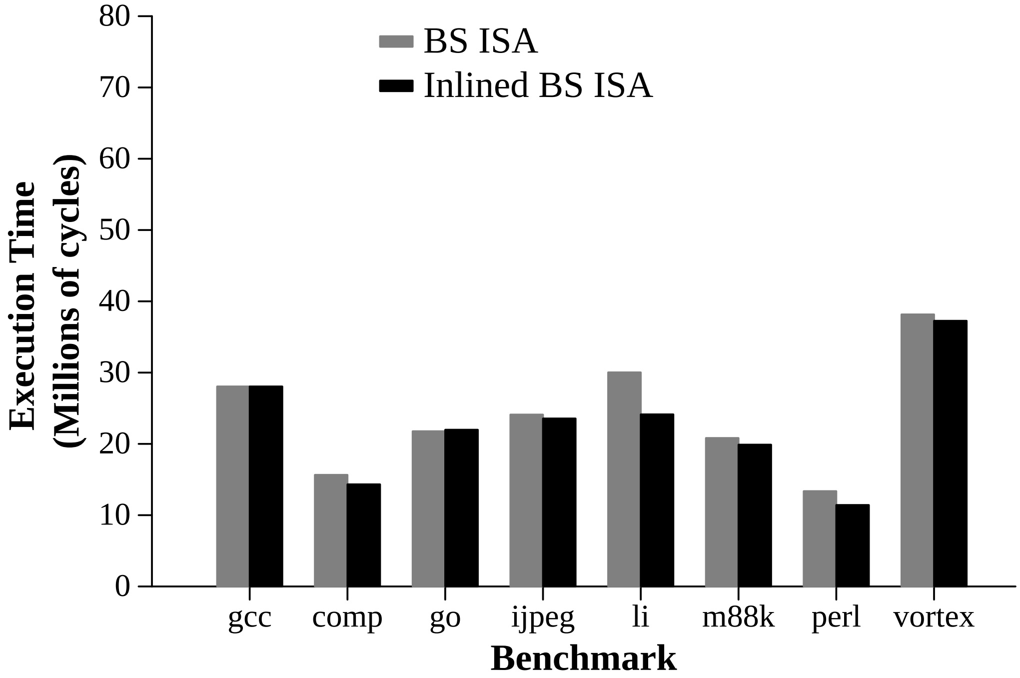

Figure 5.29: Execution times of block-structured ISA executables compiled with and without function inlining.

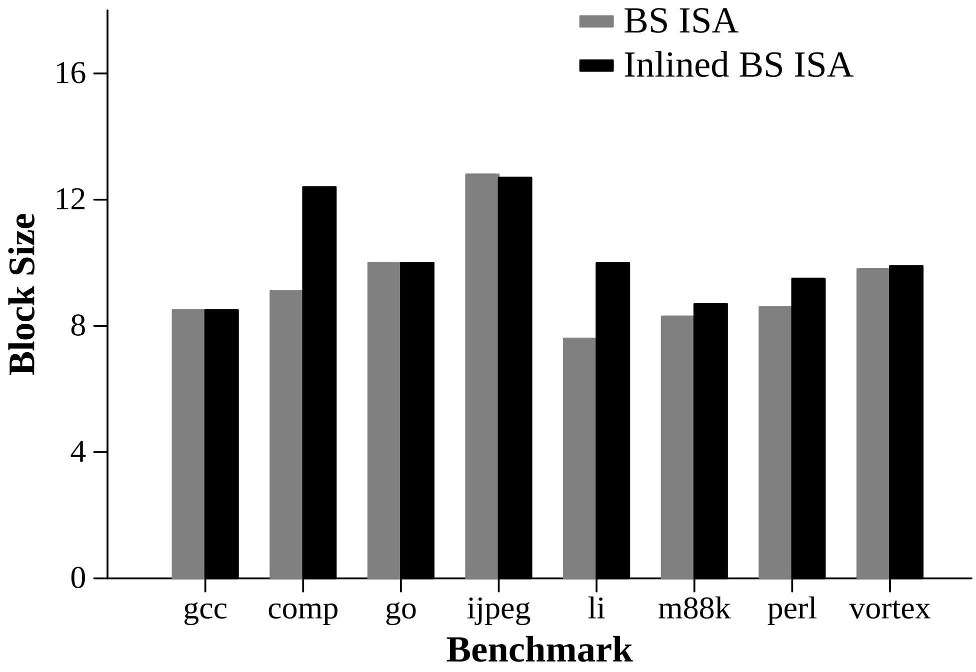

Figure 5.30: Average block sizes of block-structured ISA executables compiled with and without function inlining.

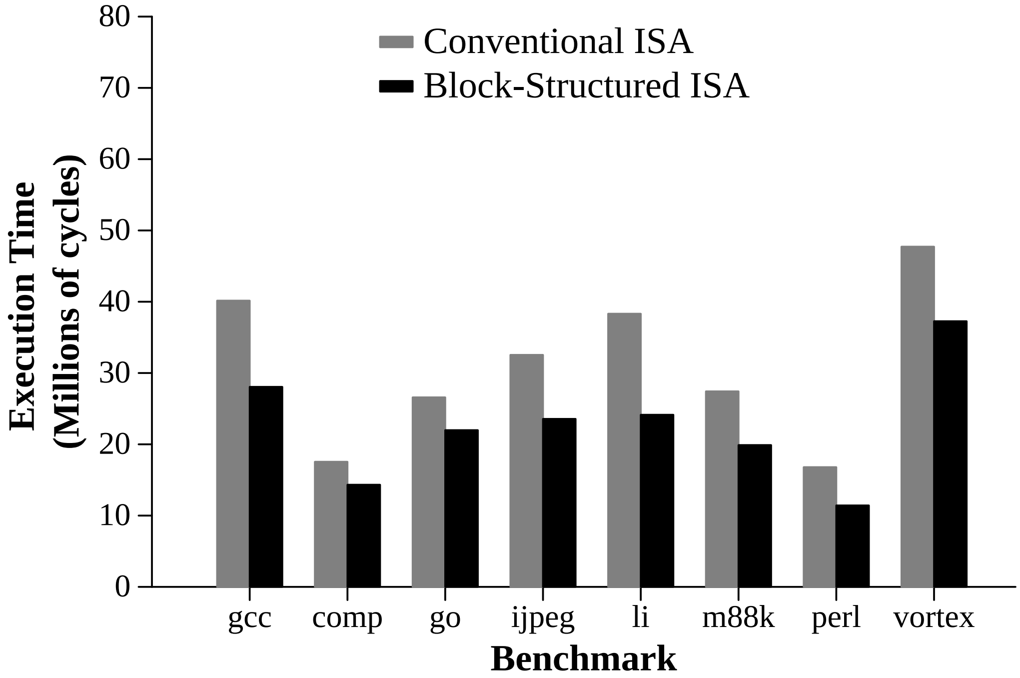

Figure 5.31: Comparing the performance of inlined conventional ISA executables to inlined block-structured ISA executables.

total execution time. Second, the average block size for the ijpeg benchmark shrank slightly. The reason for this is that the function being inlined was very small. Once the function was inlined and all the instructions associated with the call and return of that function were eliminated, the resulting block was smaller than any of the original blocks.

Figure 5.31 compares the performance of inlined block-structured ISA executables to inlined conventional ISA executables. Both sets of executables inlined the same call sites. The block-structured ISA executables reduced execution time by an average of 26.5%, a slight improvement over the performance difference for the non-inlined versions of the executables. Because the dynamic frequency of the functions to be inlined were required to be greater than the minimum threshold, the number of functions that qualified for inlining was not large enough to make a significant performance difference. A better approach to overcoming the call/return constraint might be to partially inline function calls and returns that prematurely end block enlargement. See chapter 9.2 for more details.

5.5 Handling the Max Successor Constraint

This section examines a profile-guided approach to reducing the performance penalty due to the max successor constraint. As will be discussed in chapter 6, the branch predictor restricts each enlarged block in a block-structured ISA program to at most eight control flow successors — four successors from the taken trap side and four successors for the not taken trap side. The base block enlargement optimization enforces this restriction by allowing each enlarged block to have at most two fault operations. The disadvantage of this approach is that it restricts all the paths in the program to the same degree of enlargement. A better approach would allow greater degrees of enlargement along frequently executed paths by reducing the degree of enlargement along infrequently executed paths. To exploit this tradeoff, the base block enlargement optimization is modified to incorporate program profile information that specified the relative frequencies of all the paths in the program. Rather than traversing the control flow graph in a breadth-first order, the block enlargement optimization will now enlarge the blocks along the most frequently executed program paths first. Furthermore, block enlargement will no longer stop when an enlarged block has accumulated two fault instructions. Block enlargement will stop if further enlargement causes a block to have more than the maximum number of successors. As a result, the block enlargement optimization will now be able to form even larger blocks along the frequently executed program paths.

Figure 5.32 contrasts the difference between applying the block enlargement optimization with the two fault restriction and applying the block enlargement optimization with the profile-guided approach. The original control flow graph is shown at the top. The control flow graph on the lower left is the result of applying the block enlargement optimization with the two fault restriction. The control flow graph on the lower right is the result of applying the block enlargement optimization with the profile-guided approach where the path through blocks B, C, E, and I is assumed to be the most frequently executed path. When the block enlargement optimization is restricted to two faults, it can combine at best blocks B, C, and E together along the most frequently taken path. When the block enlargement optimization is able to focus on the frequently taken path, it can combine blocks B, C, E, and I together along the most frequently taken path. The tradeoff is that the profile-based approach can combine only blocks B and D together along the less frequently taken path instead of blocks B, D, and G or H.

Figure 5.33 compares the performance of block-structured ISA executables generated using the the block enlargement optimization with the profile-based approach to the performance of block-structured ISA executables generated using the block enlargement optimization with the two fault restriction. Figure 5.34 compares the average block sizes of the two sets of executables. The profile-based approach is able to reduce the execution time by an average of 2.6% by increasing the average block size by 4.2%, from 10.2 to 10.6. This further reduction in execution time reduces the performance difference between block-structured ISA executables and conventional ISA executables to 28.4%. Further increases in block size and performance may be achieved by extending the profiled-based approach to dedicate an even greater fraction of a block's successors to the most frequently taken paths.

5.6 Summary

The key results presented in this chapter are:

- For the SPECint95 benchmarks, the block-structured ISA executables running on a sixteen wide HPS processor show a performance improvement of 28.4% as compared to the conventional ISA executables running on an identically configured HPS processor that fetches only one basic block each cycle. This performance improvement is due

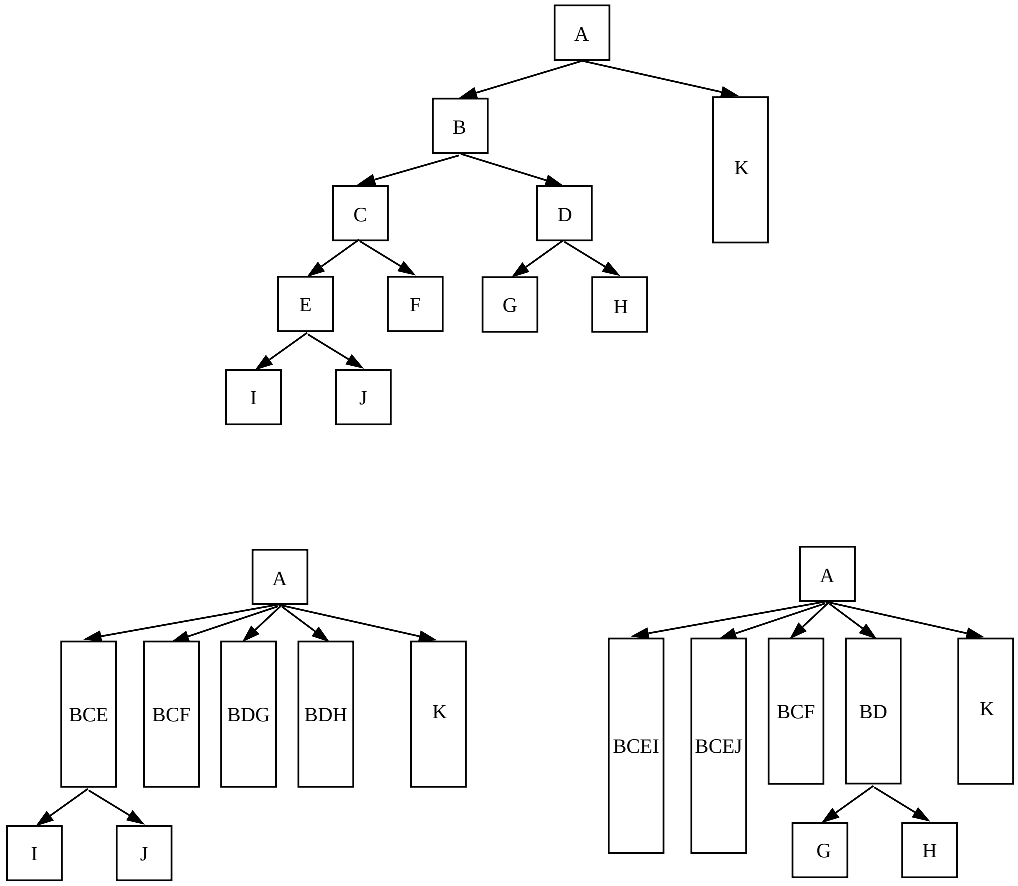

Figure 5.32: Comparing the two fault restriction to the profile-guided approach for enforcing the max successor constraint. The control flow graph on the lower left is the result of using the two fault restriction. The control flow graph on the lower right is the result of using the profile-guided approach.

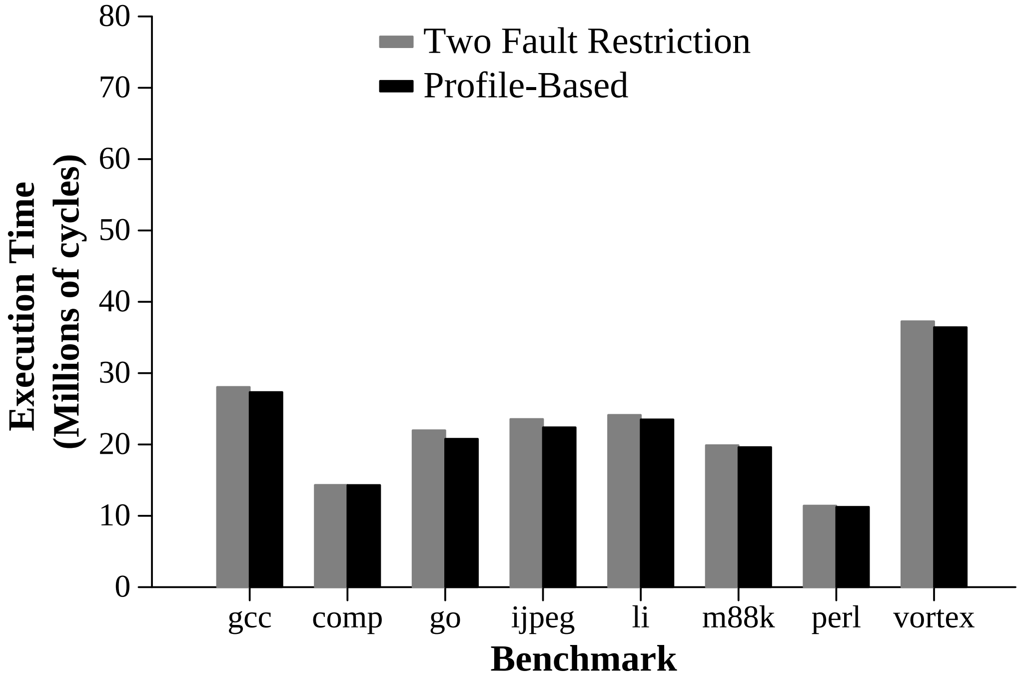

Figure 5.33: Performance of the two fault restriction and profile-based approach to enforcing the max successor constraint.

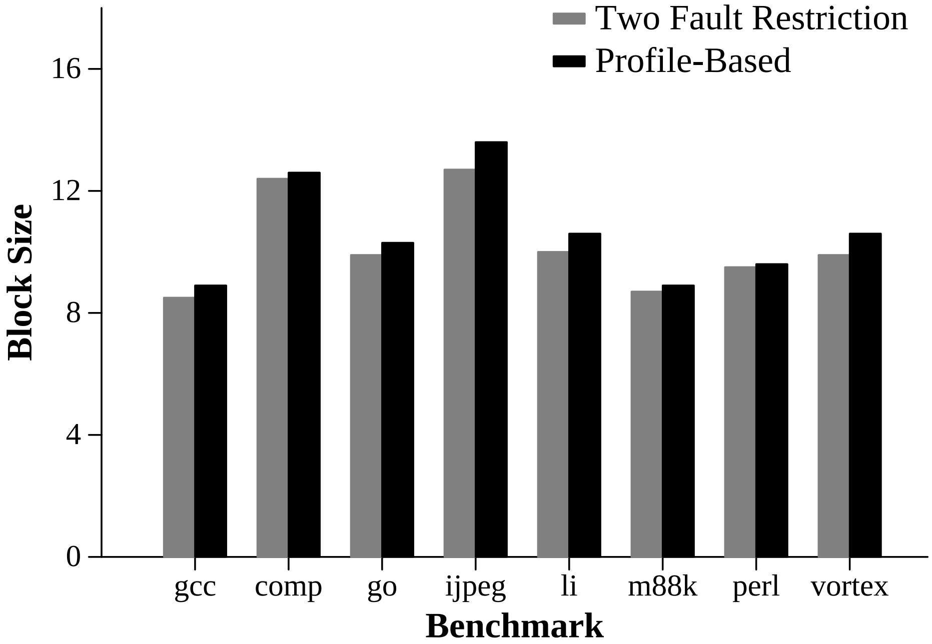

Figure 5.34: Average block sizes for the two fault restriction and profile-based approach to enforcing the max successor constraint.

the block enlargement optimization increasing the average block size from 5.8 to 10.6 instructions.

- The major constraint that prevents the block enlargement optimization from producing even larger blocks is the optimization's inability to enlarge blocks beyond call and return instructions. The other significant constraint is the restriction on the number of control flow successors for each block that is imposed by the branch predictor.
- Inlining frequently called routines reduces execution time by 6.3%, reducing the performance impact of the call/return constraint.
- Using the profile-based approach instead of the two fault restriction to enforce the max successor constraint reduces execution time by an additional 2.6%.
- The code duplication due to block enlargement incurs an insignificant number of icache misses for an icache of size 128KB. The exception to this observation was the go benchmark, which spent 27% of its total execution time stalled due to icache misses.

CHAPTER 6

Branch Prediction for Block-Structured ISAs

Because the block enlargement optimization combines multiple basic blocks into a single enlarged atomic block, each atomic block in a block-structured ISA can contain multiple branches. To be effective, a branch predictor for a block-structured ISA processor must be able to make multiple branch predictions per cycle. This chapter describes a way to extend the Two-Level Adaptive Branch Predictor and the Branch Target Buffer so that accurate branch predictions can be made for block-structured ISAs. Using these extensions, a specific branch predictor implementation is presented. The performance of this predictor implementation will be studied in chapter 7. Finally, this chapter will discuss branch prediction issues that are specific to block-structured ISAs and the predictor implementation being considered.

6.1 Effect on Prediction Accuracy

Because the predictor for a block-structured ISA must make multiple predictions per cycle, the probability that it will make a misprediction for any given cycle is greater than that for a predictor making a single prediction per cycle. However, this does not imply that the overall prediction accuracy achieved by the block-structured ISA predictor is lower than that achieved by a predictor making a single prediction per cycle. This is because prediction accuracy is defined as the ratio of the number of correctly predicted branches to the number of predicted branches. Thus, although the block-structured ISA predictor may be making more mispredictions per cycle, it is also making more predictions per cycle which makes the overall ratio between mispredicted branches (or conversely, correctly predicted branches) and predicted branches equivalent to that achieved by a conventional predictor. So from a probabilistic viewpoint, the number of branches predicted each cycle by a predictor should not affect its overall prediction accuracy.

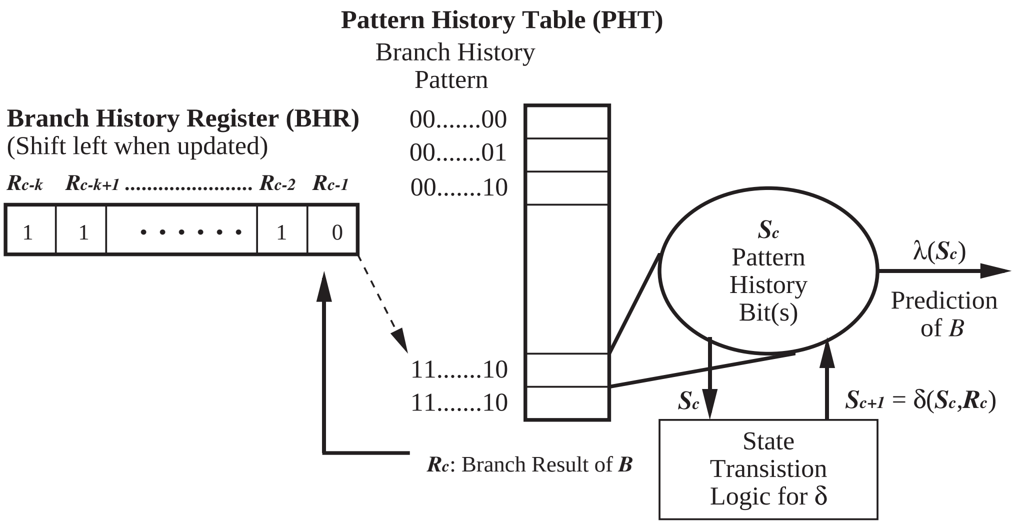

Figure 6.1: Structure of the global variation of the Two-Level Adaptive Branch Predictor.

6.2 Two-Level Adaptive Branch Prediction

6.2.1 Background

The Two-level Adaptive Branch Predictor exploits the correlation among a program's branches to accurately predict their outcomes [55] [56] [38] [57]. This correlation can be detected by recording the program's branch behavior at two levels, branch execution history and pattern history. The Two-Level Adaptive Branch Predictor has two key variations: global and per-address. For the global variation, the branch execution history records the direction taken by the last k branches executed in the program. This history can be stored as a k-bit pattern where a "0" represents a not-taken branch and a "1" represents a taken branch. The pattern history records for a specific branch and k-bit pattern, the outcomes of that branch for its last j occurrences when the branch history had that k-bit pattern. The per-address variation works in the same general manner as the global variation except that the branch execution history is recorded on a per branch basis (i.e. for each branch, there is a record of the last k directions taken by that branch).

Figure 6.1 illustrates an implementation of the the global variation of the Two-Level Adaptive Branch Predictor. The branch execution history is stored in the branch history register. The pattern histories for a given branch are stored in the pattern history table. The predictor possesses one or more pattern history tables (PHTs), where each table is shared by some number of branches. To predict a branch, the branch's address selects its corresponding pattern history table. The current value in the branch history register is used as an index into that table. The prediction is made based on the contents of the specified table entry. Research has shown that the two-bit saturating up-down counter serves as an effective PHT entry [56]. The counter is incremented each time the branch is taken and decremented each time the branch is not taken. If the counter's value falls into the lower half of the range of possible values, the PHT entry produces a not taken prediction. If the counter's value falls into the upper half of the range of possible values, the PHT entry produces a taken prediction. In this way, the counter captures which branch direction has been most frequent in the recent past.

6.2.2 Extensions for Block-Structured ISAs

For conventional ISAs, the branch predictor selects one target out of two possibilities. For block-structured ISAs, the branch predictor must be able to select one target out of 2^n possibilities where n is the maximum number of branches allowed in a single block. To extend the Two-Level Adaptive Branch Predictor so that it can select one target out of 2^n possibilities, the pattern history table and branch history register must be modified.

The Pattern History Table

As discussed above, the purpose of each PHT entry is to record for its corresponding branch history the branch direction (or branch target) that has occurred most frequently in the recent past. For a conventional ISA in which each branch has at most two targets from which to choose, the two bit counter serves this purpose effectively. However, for a block-structured ISA that may have up to n branches in each enlarged block, the predictor must choose among 2^n possibilities. To make this choice, each PHT entry is extended to include 2^n two bit counters. Each counter is associated with one of the possible targets and the counter's job is to record the relative frequency with which its associated target has occurred in the recent past. Thus, the predictor can then determine the target which has occurred most frequently in the recent past by comparing the values of all the counters in a PHT entry and selecting the target whose counter has the largest value.

The actual implementation of the branch predictor PHT entry included one more two bit counter in addition to the 2^n target counters. This two bit counter, the trap counter, is used to predict the direction taken by the current block's trap operation. The usage and update rules for this counter are identical to those for the conventional Two-Level Adaptive Branch Predictor. The target counters are partitioned into two 2^{n-1}-sized subsets, according to whether the current block's trap operation had to be taken or not taken for the counter's target to be a possible control flow successor for the current block (see chapter 4.1.1). To make a prediction, the trap counter is first checked to predict the trap's direction and select the appropriate target counter subset. The predicted target is the target whose counter in that subset is set to the maximum possible counter value. The counter update rules guarantee that at least one counter in each subset will be set to the maximum value. If more than one counter is set to the the maximum value, then the predicted target with the higher priority (i.e. the target with the lower target index — see section 6.3) is chosen. Figure 6.2 illustrates the prediction generation process.

The update rules for the target counters are as follows: whenever a target occurs in the dynamic instruction stream, the target's counter in the appropriate PHT entry is incremented if that counter's value is not already set to the maximum possible counter value. If the target's counter value is already at the maximum, the counter values for all the other targets in its subset are decremented. As a result, each counter gives the frequency with which its

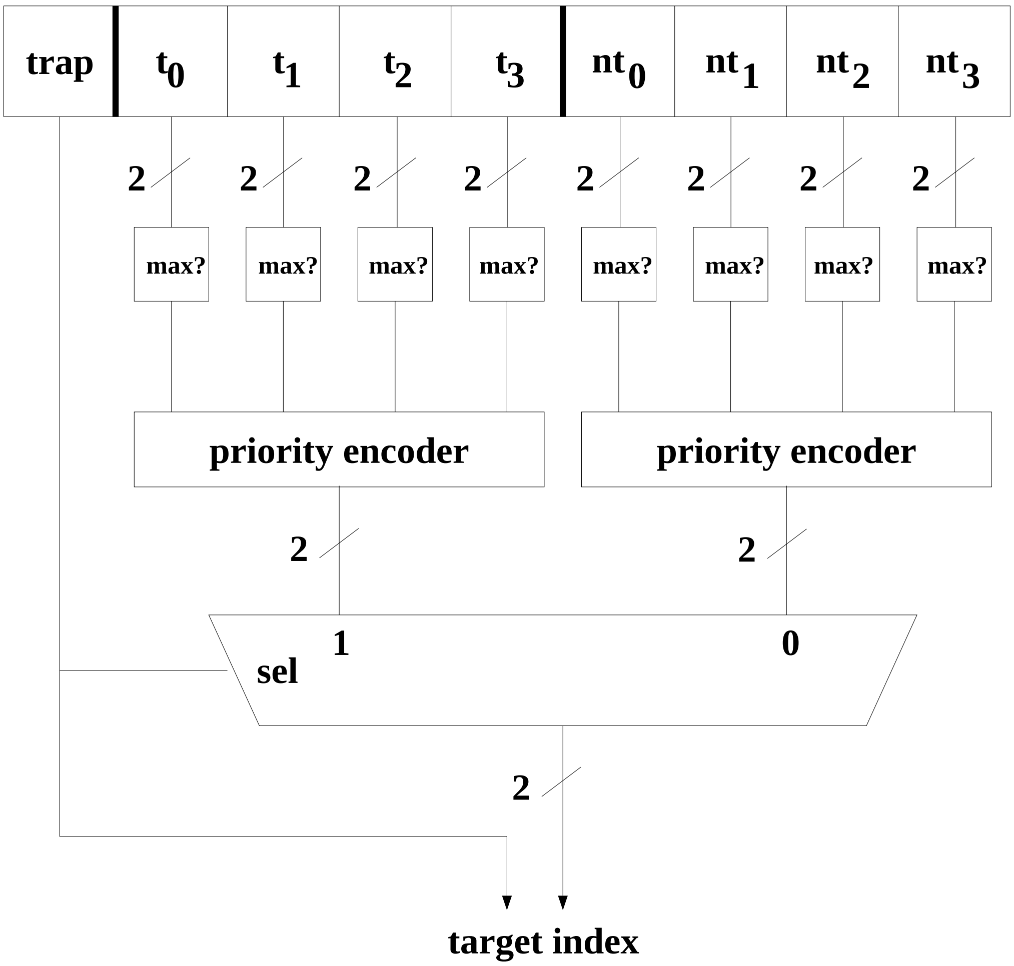

Figure 6.2: Generating the predicted target index from a PHT entry in a block-structured ISA branch predictor that supports eight targets per block.

associated target has occurred relative to the other counters in its subset.

The target counters were split into two subsets so that the predictor could always maintain information about the relative frequencies of the targets from both sides of the trap operation. Thus in the event of a trap misprediction, the predictor would still have some information available to help make an accurate prediction among the targets on the other side of the trap operation. Keeping the counters in a single set would have allowed the repeated occurrence of one target for a given branch history to cause all the other counters in the PHT entry to be decremented to zero. As long as this target remained the correct prediction for the given branch history, this effect would not be a problem. However, if at some point in the future, the correct target were to change to another target on the other side of the trap operation, then upon recovering from the misprediction (which the predictor would inevitably make because the repeated occurrence of the first target had trained the predictor to choose the first target), the predictor would have no information as to which target on the other side of the trap to choose as the point of recovery. This is because all the other target counters have been decremented to zero, as mentioned above. By splitting the target counters into two subsets, this problem is avoided.

The Branch History Register

Because the predictions produced by the Two-Level Adaptive Branch Predictor are now target indexes (that select which target out of the set of possible successors is the next block) instead of branch directions, the branch history register will now record the target indexes for the most recent blocks in the dynamic instruction stream instead of the directions taken by the most recent branches. This change, however, does not change the information content in the branch history register because the target indexes are really just another representation of the taken branch directions. Each successor target of a block can be mapped to a unique sequence of branch directions that must occur in order for that successor to be the correct successor for the block. In addition, the predictor maps each successor target to a unique target index. Thus, there is a one-to-one mapping between target indexes and branch direction sequences.

While target indexes may contain the same amount of information as branch histories, target indexes do not represent the information as efficiently. If each block in a block-structured ISA can have up to 2^n successors, then each target index will consist of n bits. However, if a given block has only two successors, then representing the prediction made at that block as a n bit value will waste space — clearly only one bit is needed to represent the prediction made. For a branch history register of finite size, shifting in the extra $n-1$ bits will result in the history register having to sacrifice $n-1$ bits of older history that could have provided useful information for subsequent predictions. To eliminate this waste, each block is tagged with the minimum number of bits required to uniquely specify all the block's successors (see chapter 4.2.2). This number is used by the predictor to determine how many bits from the target index of the block's successor to record in the branch history register. As a result, the branch history register need not shift in the full target index for each prediction.

6.3 Extensions to the Branch Target Buffer

The Branch Target Buffer (BTB) [46, 27] must be extended so that it can keep track of all the possible control flow successors for a block. For a conventional ISA, the number of successors is limited to two, the taken and fall-through targets of the block's branch. For a block-structured ISA where n is the maximum number of branches allowed in a single block, the number of successors can be as great as 2^n.

In addition to keeping track of the 2^n possible successors for each block, the BTB must maintain a mapping between each successor (or target) and its corresponding target index just as the BTB distinguishes between the taken and fall-through targets in a conventional ISA. This mapping is established by the compiler. The targets specified by the trap operation itself are assigned the target indexes zero and $2^n - 1$. As discussed in chapter 4.2.2, the predictor incrementally learns about the other possible targets for the block when the fault operations in the explicit trap targets are mispredicted. These fault operations will redirect the instruction stream to the other possible targets. The target indexes to be associated with these targets are encoded by the compiler in the corresponding fault operations.

Instead of creating the mapping statically, the mapping between targets and target indexes could have been created dynamically by the hardware. As each successor for a block is encountered for the first time, the BTB assigns the next available target index to that successor. The exact mapping created by the hardware would depend on the order in which the block's successors occurred in the dynamic instruction stream. The decision was made to create the mapping at compile-time to guarantee that the exact mapping between targets and target indexes would always be the same. This guarantee has a significant performance impact when executing a program that incurs a significant number of BTB capacity or conflict misses. Consider a block whose BTB entry was evicted from the BTB and then brought back in again at some later point. When the block's entry is brought back in again to the BTB , the block's targets must once again incrementally reacquired and the target mapping must once again be incrementally recreated. If the target mapping is specified by the compiler, then the recreated mapping is guaranteed to be the same as the mapping used when the block's entry first resided in the BTB. By using the same mapping as before, the predictor can exploit any branch behavior information about this block that may still remain in the PHTs. If the target mapping is created by the hardware, then there is no guarantee that the recreated mapping will be the same as the original mapping. If the recreated mapping is different than the original mapping, then the predictor would not be able to exploit the remaining branch behavior information in the PHTs. This information would be nonsensical to the predictor because it was based on a different target mapping. As a result, more accurate branch predictions can be made in the presence of BTB misses if compiler-specified target mapping is used.

6.4 A Specific Implementation

For the experiments in chapter 7, the branch predictor used in the block-structured ISA processor is a hybrid branch predictor [30, 8, 6] with two component predictors. For each branch, each component predictor generates a prediction. The hybrid predictor selects the prediction from the component predictor that has been more accurate for that branch in the recent past to serve as the final branch prediction. The hybrid predictor uses a table of counters to determine for each branch which component predictor has been more accurate. Each counter is associated with a particular program branch. If the first predictor generates a correct prediction for the branch and the second predictor generates an incorrect prediction, the counter is decremented. If the first predictor generates an incorrect prediction for the branch and the second predictor generates a correct prediction, the counter is incremented. If both predictors generate correct predictions or both predictors generate incorrect predictions, the counter is left unchanged. Under these update rules, the first component predictor's prediction is selected as the final prediction if the counter value is within the lower half of the range of possible values and the second component predictor's prediction is selected if the counter value is within the upper half of the range of possible values.

The two component predictors used were the gshare variation [30] and the PAs variation [56] of the Two-Level Adaptive Branch Predictor. The gshare variation records global branch history in its branch history register. The direction of every branch that occurred in the dynamic instruction is recorded in the history register. In addition, the index into to the PHT was formed by xor'ing the branch history with the current branch address. The PAs variation records per-address branch history in its branch history registers. The PAs variation associates one branch history register with each branch. Each branch history register records only the directions taken by its associated branch. For the block-structured ISA predictor, the gshare and PAs component predictors were modified as described above.

CHAPTER 7

Branch Prediction — Measurements and Analysis

This chapter examines the performance of the block-structured ISA branch predictor described in the previous chapter. Using this branch predictor, the performance of a block-structured ISA processor is compared to the performance of an identically configured conventional ISA processor. The conventional ISA processor will use a branch predictor that has the same configuration as the block-structured ISA predictor but without the extensions for block-structured ISAs.

7.1 Base Comparison

Figure 7.1 compares the performance of the SPECint95 benchmarks executing on a block-structured ISA processor with a 32KB branch predictor to the performance of those benchmarks executing on a conventional ISA processor with the same sized branch predictor. The configuration of the block-structured ISA branch predictor was gshare(13,1)/PAs(13,1). The configuration of the conventional ISA branch predictor was gshare(16,1)/PAs(16,1). The block-structured ISA executables were compiled with the base variation of the block enlargement optimization (see chapter 5.1). The block-structured ISA executables achieved an average reduction in execution time of 12%, about half of what was achieved when perfect branch prediction was assumed. The go benchmark suffered the most dramatic decrease in performance. The block-structured ISA version of that benchmark is 14% slower than the conventional ISA version. As will be shown in the following sections, this change in performance was due to the block-structured ISA version of the go benchmark suffering a larger penalty due to mispredicted branches as compared to the conventional ISA version of the benchmark.

7.2 Branch Predictor Performance

Figure 7.2 shows the number of cycles instruction fetch was stalled due to branch mispredictions while executing each benchmark. The misprediction stall cycles accounted for 28% of the total execution time for the conventional ISA executables and 39% of the total execution time for the block-structured ISA executables. Because they accounted for such a large fraction of the total execution time for both the conventional ISA and block-structured ISA

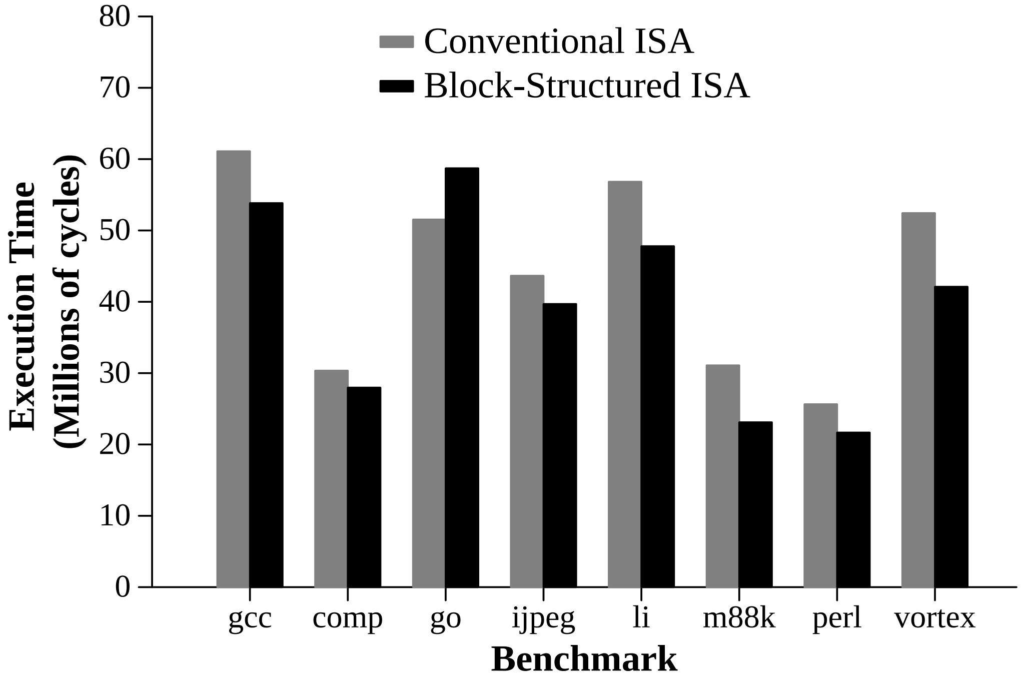

Figure 7.1: Execution times for block-structured ISA executables and conventional ISA executables when using a 32KB branch predictor.

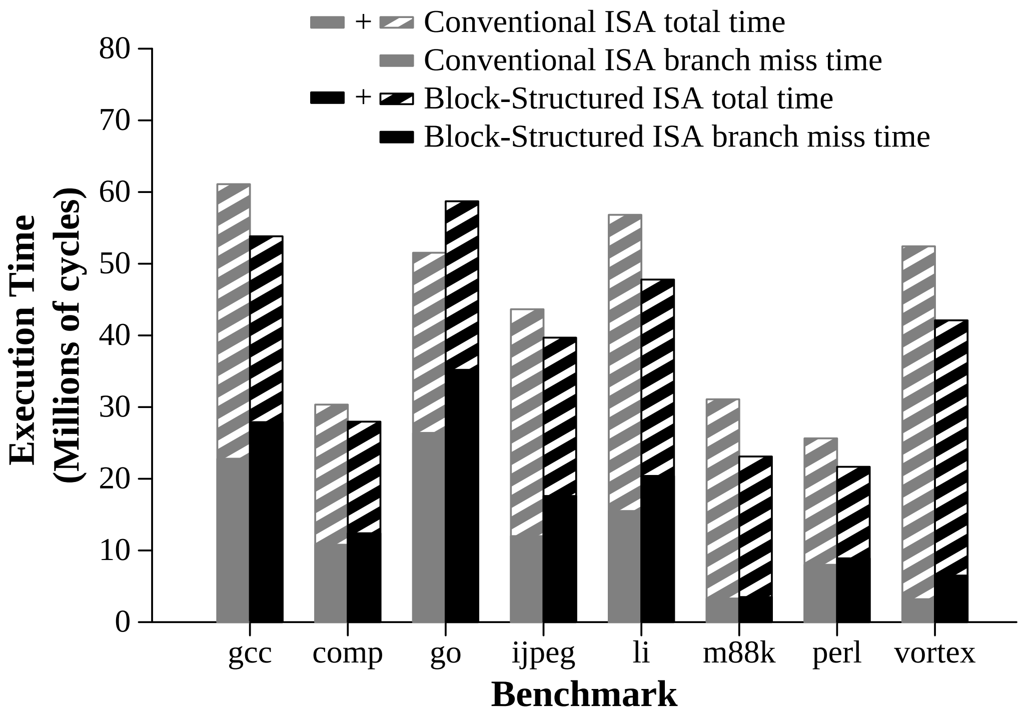

Figure 7.2: Execution times for block-structured ISA executables and conventional ISA executables when using a 32KB branch predictor.

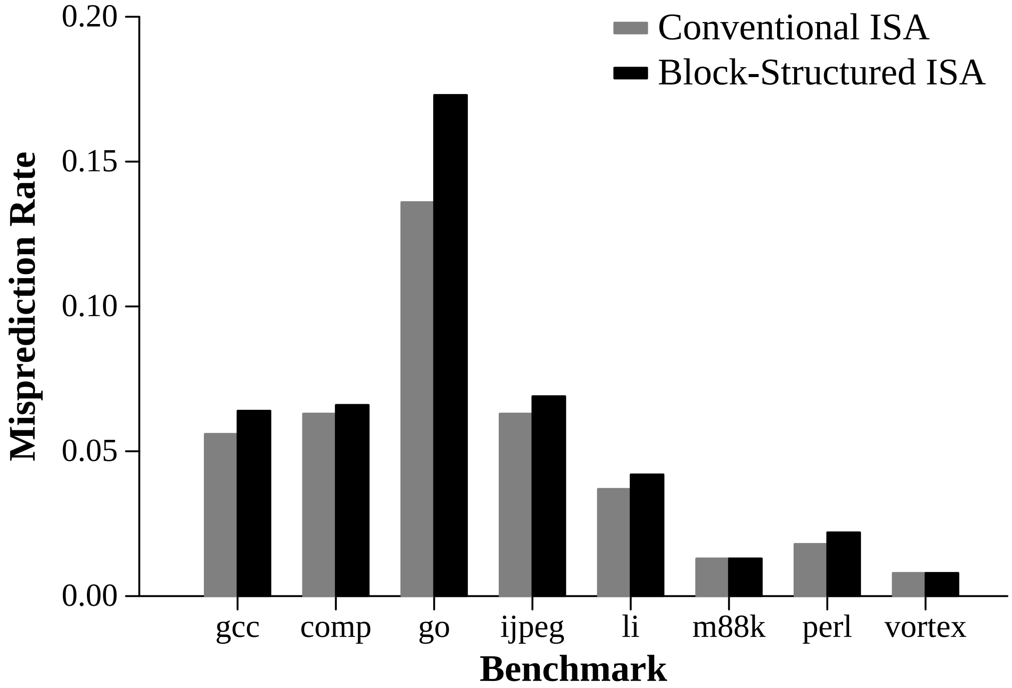

Figure 7.3: Misprediction rates for a 32KB block-structured ISA predictor and a 32KB conventional ISA predictor.

executables, the misprediction stall cycles reduced the relative performance improvement achieved by the block-structured ISA. This relative performance improvement was further reduced by the fact that the absolute number of misprediction stall cycles was 34% greater for the block-structured ISA executables than for the conventional ISA executables. This increase in misprediction stall cycles is due to two reasons: lower prediction accuracy and greater branch resolution time.

7.2.1 Branch Prediction Accuracy

Figure 7.3 compares the misprediction rate of the block-structured ISA branch predictor to the conventional ISA branch predictor. The misprediction rate of the block-structured ISA predictor shows only an 11% increase over the misprediction rate of the conventional ISA predictor. From a probabilistic viewpoint, the block-structured ISA branch predictor should be able to achieve the same prediction accuracy as that of the conventional ISA branch predictor (see chapter 6.1). However, because the individual PHT entries of the block-structured ISA branch predictor are about 8x larger than those of the conventional ISA branch predictor, the branch history register length for the block-structured ISA predictor is three bits shorter than that of the conventional ISA predictor when the predictors are of equal size. This reduction in the amount of history that the block-structured ISA predictor can record decreases its accuracy. To demonstrate the importance of these missing history bits, figure 7.4 compares the prediction accuracies of the block-structured ISA and conventional

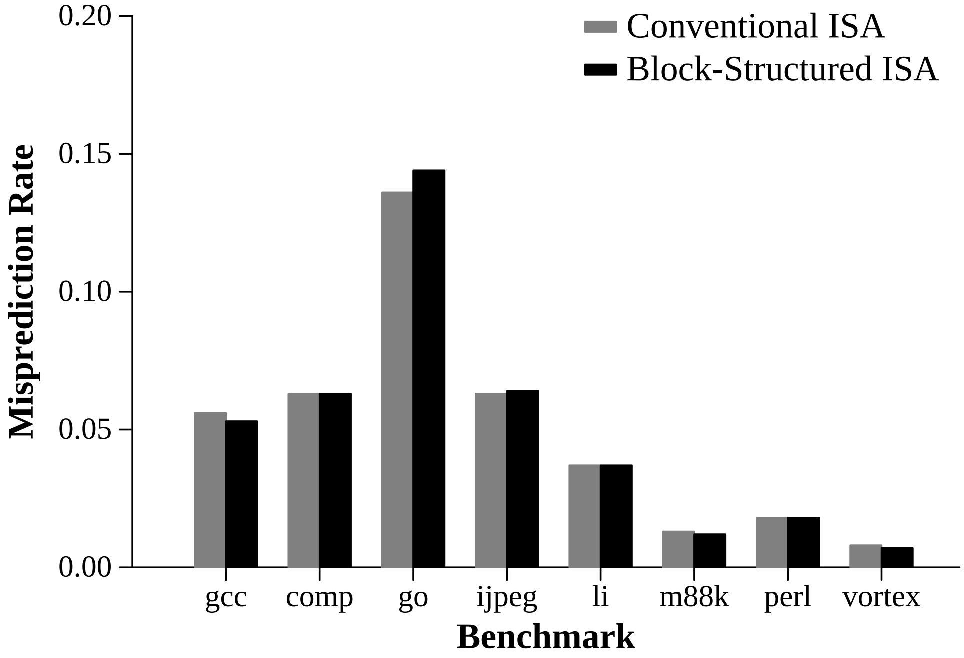

Figure 7.4: Misprediction rates for block-structured and conventional ISA predictors with 16 bit history registers.

ISA branch predictors where the size of the block-structured ISA predictor has been increased eight-fold so that its branch history register length is the same as that of the conventional predictor. By using the same history register length for both predictors, the difference in the two predictor's misprediction rates becomes negligible for all the benchmarks except go.

The go benchmark shows a significantly larger increase in misprediction rate than the other benchmarks even after the performance impact of a shorter history register was taken into account. This is because a large number of the extra mispredictions suffered by the block-structured ISA predictor was due to BTB misses. The go benchmark contains a large number of static branches that are frequently executed which results in a significant number of BTB misses during the execution of the benchmark. These BTB misses incur a greater number of mispredictions for the block-structured ISA predictor than for the conventional ISA predictor because the block-structured ISA predictor's accuracy is more severely affected by a BTB miss. When a BTB miss occurs for a conventional ISA predictor, the predictor is unable to use its PAs component (because the branch's per-address history register is stored in its BTB entry), but is still able to generate a prediction using its gshare component. However, it must wait until the branch instruction in the current block is decoded before the branch target can be determined. As a result, instruction fetch must be stalled for a cycle (assuming instruction decode is in the second stage of the pipeline) before the prediction can be used to determine the address of the next block to be fetched. When a BTB miss occurs for a block-structured ISA predictor, the predictor is still able to generate a prediction in the same manner as the conventional ISA predictor and must also wait a cycle until the current block's trap operation is decoded to determine the trap targets. However, the key difference

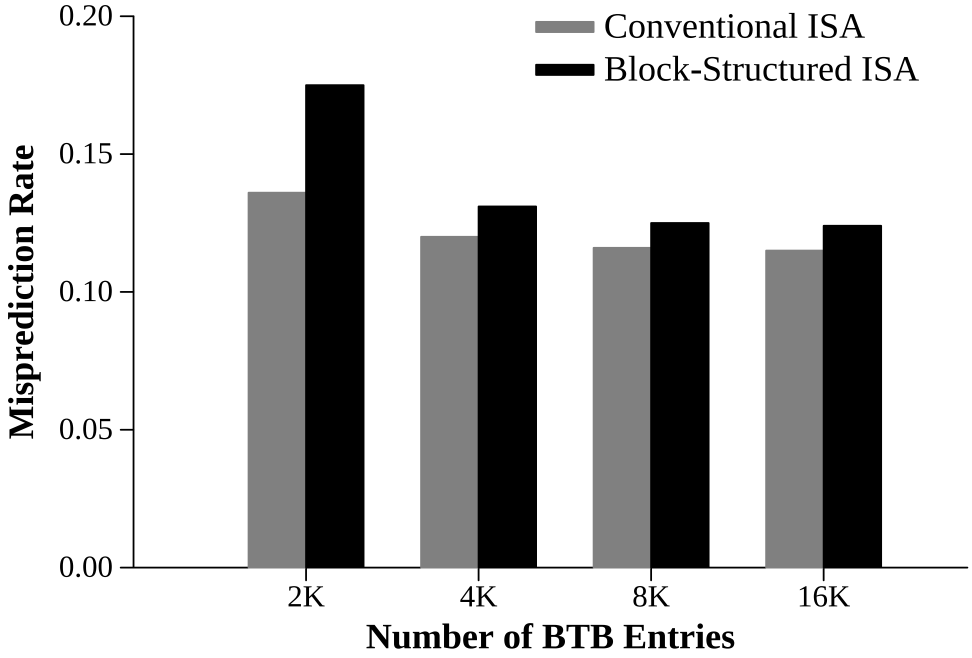

Figure 7.5: The effect of BTB size on misprediction rates for the go benchmark.

for the block-structured ISA predictor is that the trap operation only specifies two of the current block's successors (see chapter 4.2.2). If the target index generated by the predictor corresponds to a target that is not specified by the trap operation, the predictor has no way to generate the address for that target. As a result, the predictor must alter its prediction to be one of the targets specified by the trap operation, reducing the prediction accuracy of the predictor. Figure 7.5 compares the prediction accuracies of the block-structured ISA and conventional ISA predictors for the go benchmark as the BTB size is increased from 2K entries to 16K entries. The size of both predictors is 32KB. The difference in prediction accuracy decreases significantly as the BTB size is increased.

7.2.2 Branch Resolution Time

The second reason for the increase in misprediction stall cycles suffered by the block-structured ISA executables is that the branch resolution time also increases for the block-structured ISA executables. Figure 7.6 shows the average number of cycles required to resolve a mispredicted branch for both the block-structured ISA and conventional ISA executables. The branch resolution time is 12% longer for the block-structured ISA executables as compared to the conventional ISA executables. As discussed in chapter 5.1, by combining separate blocks into a single enlarged block, the block enlargement optimization increases the average length of time an instruction must wait for its dependencies to resolve. This phenomenon of increased mispredicted branch resolution time was also observed by Butler when studying the performance of a conventional ISA processor that fetched and issued

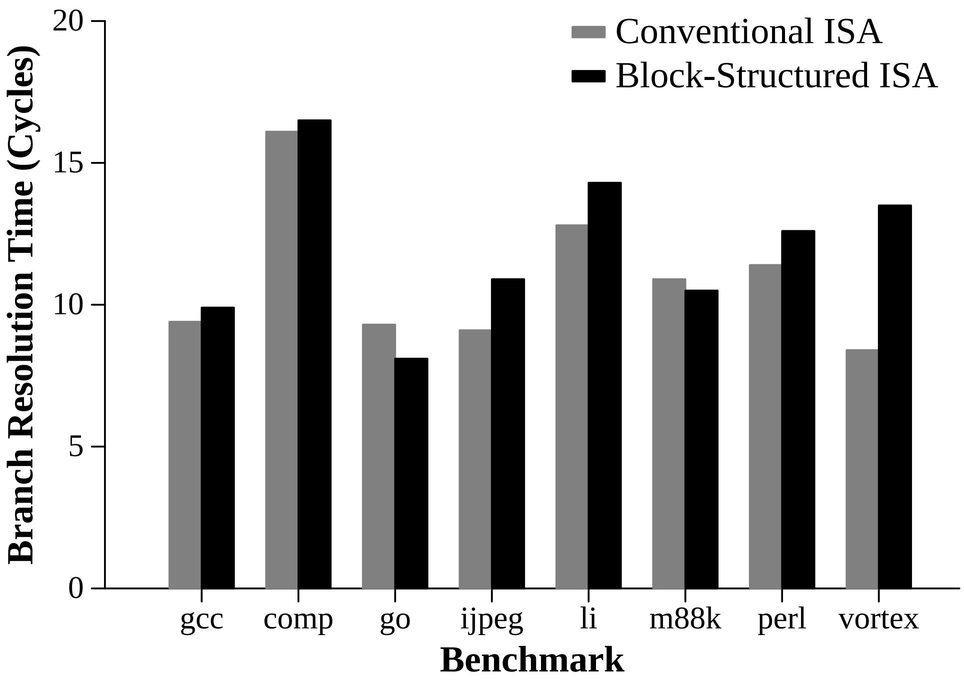

Figure 7.6: Mispredicted branch resolution times for the block-structured and conventional ISA predictors.

multiple basic blocks per cycle [3].

Unlike the other benchmarks, the branch resolution times for the go and m88ksim benchmark decreases (by 14% and 4%) for the block-structured ISA executables instead of increasing. Part of the decrease for the go benchmark was due to BTB misses and icache misses. As discussed above, the block-structured ISA branch predictor makes additional mispredictions due to BTB misses. The resolution time for such branches turns out to be faster than truly mispredicted branches. Icache misses decrease the branch resolution time if they occur while fetching a branch that will be mispredicted. Instructions that have already been issued into the machine continue to execute while the icache miss is serviced. As a result, when the branch is finally fetched and issued, the likelihood that the instructions upon which the branch depends have already been executed is greater than if the icache miss had not occurred. This, in turn, results in a shorter resolution time for the branch. Eliminating BTB and icache misses from the execution of the go benchmark reduces the difference in branch resolution time from 14% to 8%.

The remaining differences in branch resolution times for the go and m88ksim benchmarks remain unaccounted for. One possible explanation is the conventional ISA executables have a load balancing problem. When a block of instructions is issued into the processor, the first instruction in the block is sent to the first functional unit's node table, the second instruction is sent to the second functional unit's node table, and so on. This particular mapping of instructions to functional units results in a heavier workload for the functional units in the earlier positions and a lighter workload for the functional units in the later positions. Stark has shown that such load imbalances can increase the latencies of critical

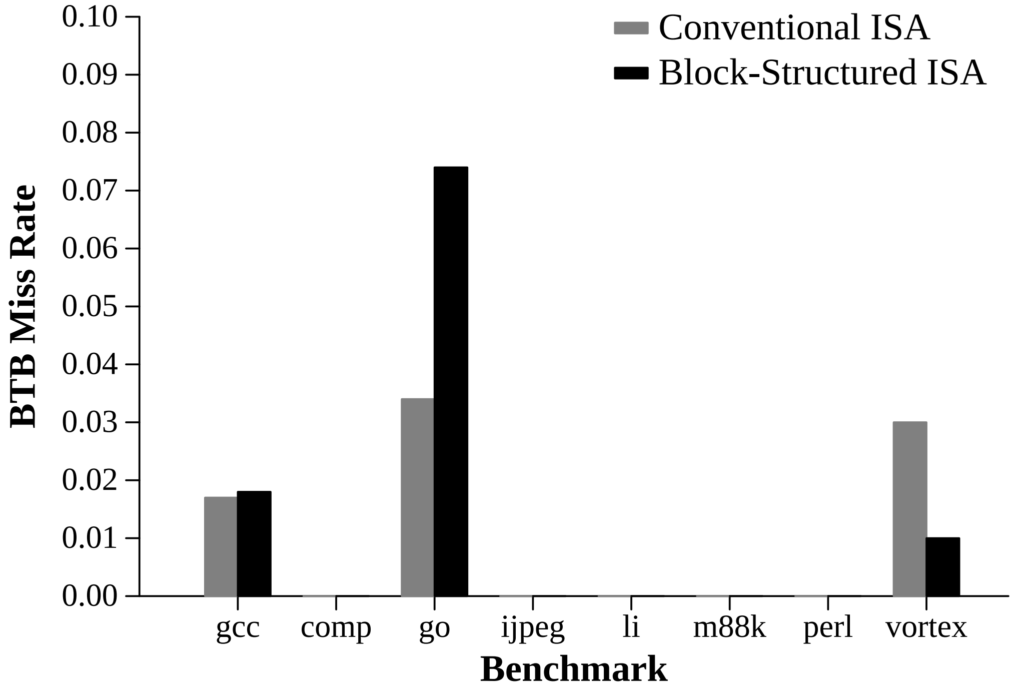

Figure 7.7: BTB miss rates for the block-structured ISA and the conventional ISA branch predictors.

paths in the program, lowering performance [50]. By merging multiple blocks into a single enlarged block, the block enlargement optimization creates a more even distribution of the workload which speeds up instruction resolution time and in particular, mispredicted branch resolution time.

7.3 BTB Performance

This section compares the BTB performance for the block-structured ISA branch predictor to that of the conventional ISA branch predictor. Figure 7.7 shows the BTB miss rates for a 2K entry BTB. Gcc, go, and vortex were the only benchmarks to have a large enough number of frequently executed branches to incur a noticeable number of BTB misses. For the gcc benchmark, the BTB miss rates for the block-structured and conventional ISA predictors were comparable. For the go benchmark, the block-structured ISA predictor's BTB miss rate was over twice that of the conventional ISA predictor. For the vortex benchmark, the block-structured ISA predictor's BTB miss rate was three times smaller than that of the conventional ISA predictor. These result are deceptive, however. Because the block-structured ISA combines multiple basic blocks into a single enlarged block enabling it to fetch the equivalent work of multiple basic blocks each cycle, the block-structured ISA processor does not need to access the BTB as many times as the conventional ISA processor to execute the same amount of work. As a result, the block-structured ISA processor may incur fewer BTB misses but still have a higher BTB miss rate. Table 7.1 lists the absolute

Benchmark	gcc	comp	go	ijpeg	li	m88k	perl	vortex
Conventional	462432	310	635518	6203	544	5666	1895	126489
Block-Structured	232831	303	633003	3810	457	981	1503	24640

Table 7.1: BTB miss counts for the block-structured and conventional ISA branch predictors.

number of BTB misses incurred while executing the block-structured and conventional ISA executables. In all cases, executing the block-structured ISA executables resulted in fewer BTB misses. For the gcc, go, and vortex benchmarks, the ratio of the absolute number of BTB misses for the conventional ISA predictor's to that of the block-structured ISA predictor's is twice that of the ratio of their BTB hit rates. This ratio indicates that each block in the block-structured ISA executables contains the equivalent work of two blocks in the conventional ISA executables, which is consistent with the increase in block size from 5.8 to 10.6 that was seen in chapter 5.

7.4 Increasing the Predictor Size

Figures 7.8–7.15 compares the misprediction rates of the block-structured ISA and conventional ISA predictors as the predictor sizes are increased from 32KB to 128KB. The configuration for the 32KB sized block-structured ISA predictor is gshare(13,1)/PAs(13,1). The configuration for the 32KB sized conventional ISA predictor is gshare(16,1)/PAs(16,1). To double the size of each predictor to the next cost level, an extra bit of history is added to each of the component predictor history registers. The average misprediction rate of the block-structured ISA predictor decreases from 5.3% at 32KB to 4.6% at 128KB. The average misprediction rate of the conventional ISA predictor decreases from 4.9% at 32KB to 4.0% at 128KB. These figures show two interesting trends. First, for the compress and ijpeg benchmarks, the conventional ISA predictor's misprediction rate decreases at a greater rate than the block-structured ISA predictor's misprediction rate. Second, for the m88ksim, perl, and vortex benchmarks, the misprediction rates for the block-structured ISA and conventional ISA branch predictors are almost identical.

For the majority of the benchmarks, the misprediction rates for both the block-structured ISA and conventional ISA predictors decreased at the same rate. However, for the compress and ijpeg benchmarks, the conventional ISA predictor misprediction rate decreases at a greater rate. This is due to the block-structured ISA predictor's inability to take full advantage of the hybrid predictor. The power of hybrid branch prediction is that it enables the branch predictor to choose the most appropriate prediction scheme to use for predicting each branch. Because the block-structured ISA predictor is predicting multiple branches simultaneously (i.e. the branches that are grouped together in an enlarged block) when it generates the target index of the next block to be fetched, the predictor is constrained to use the same

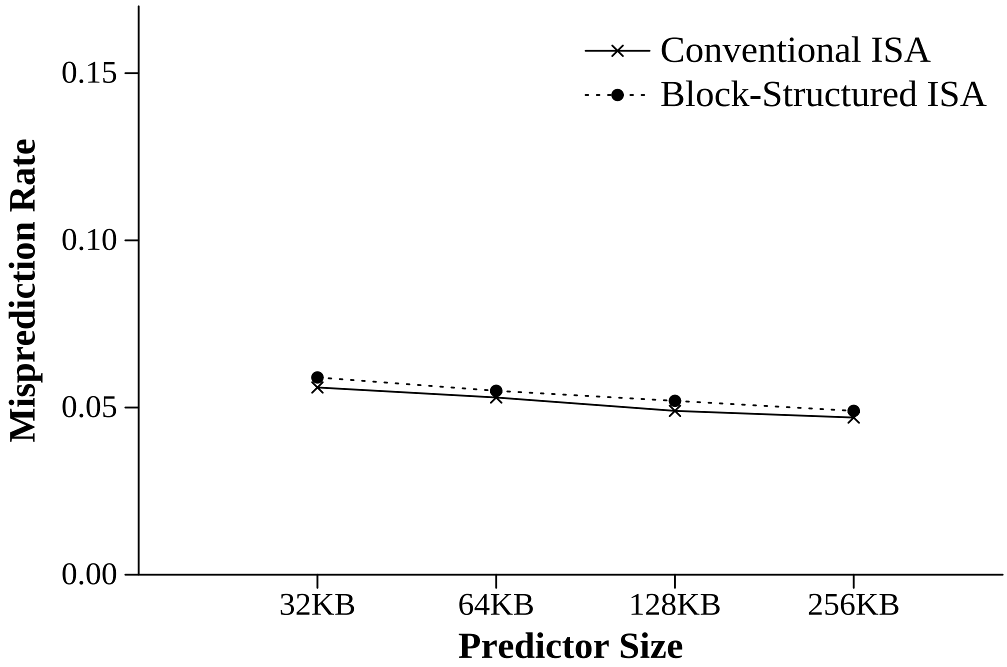

Figure 7.8: Misprediction rates for the gcc benchmark for block-structured ISA and conventional ISA branch predictors of size 32KB to 256KB.

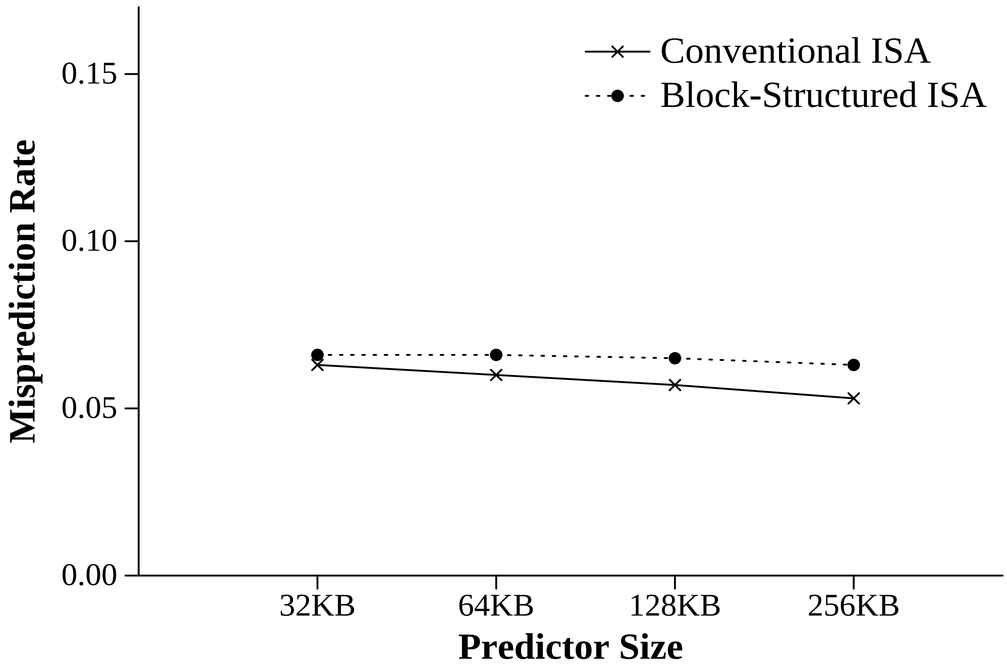

Figure 7.9: Misprediction rates for the compress benchmark for block-structured ISA and conventional ISA branch predictors of size 32KB to 256KB.

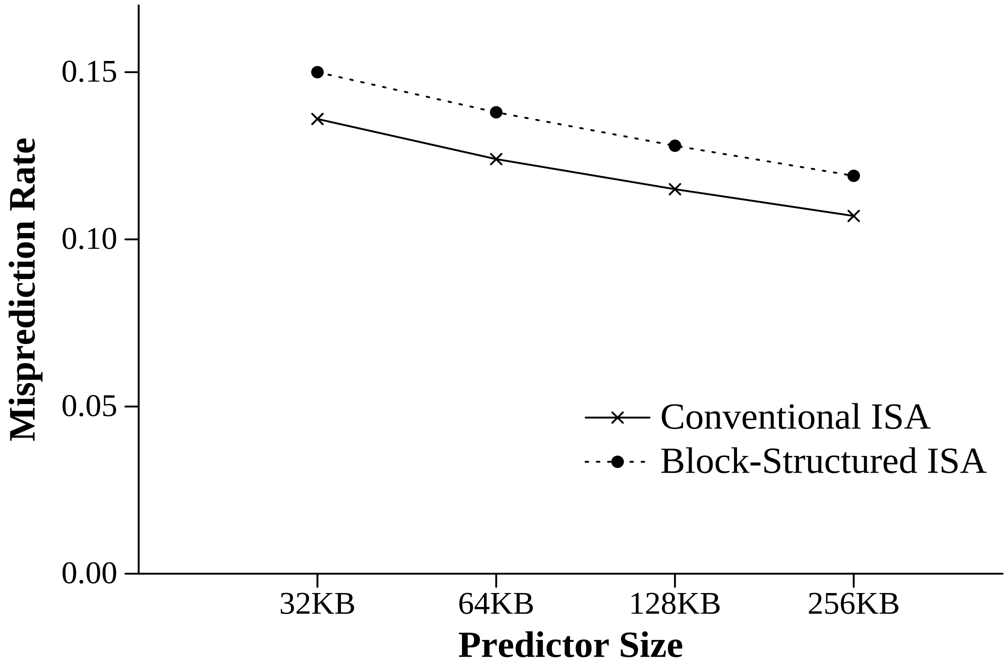

Figure 7.10: Misprediction rates for the go benchmark for block-structured ISA and conventional ISA branch predictors of size 32KB to 256KB.

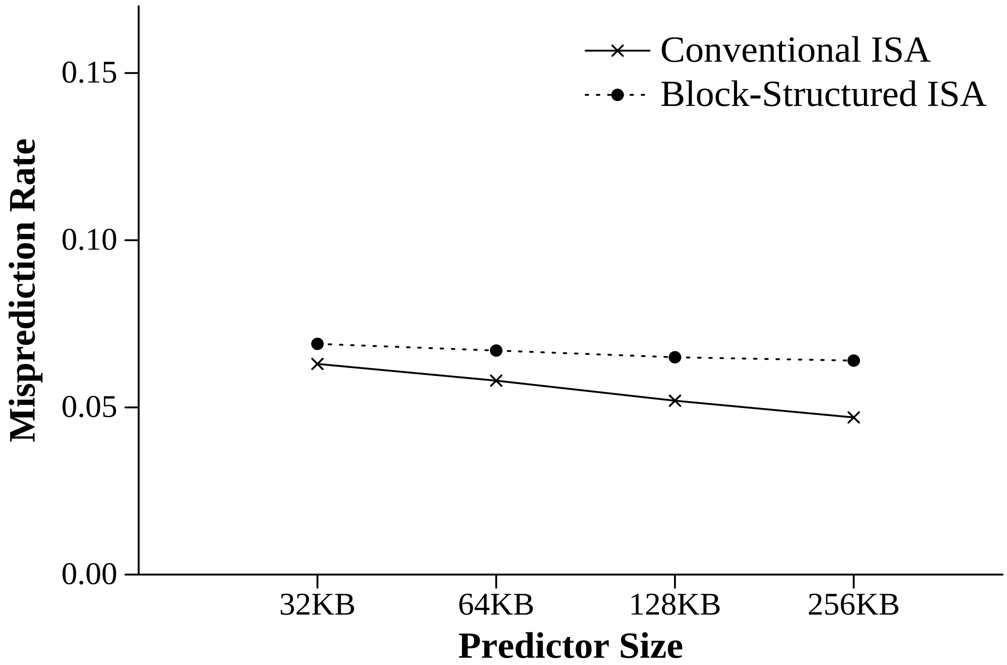

Figure 7.11: Misprediction rates for the ijpeg benchmark for block-structured ISA and conventional ISA branch predictors of size 32KB to 256KB.

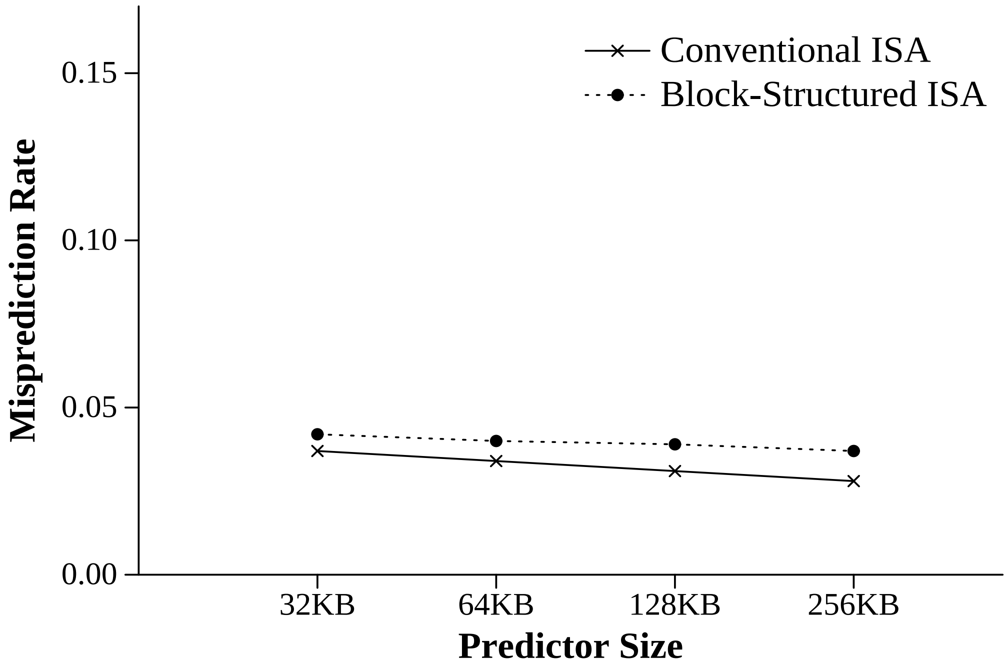

Figure 7.12: Misprediction rates for the xlisp benchmark for block-structured ISA and conventional ISA branch predictors of size 32KB to 256KB.

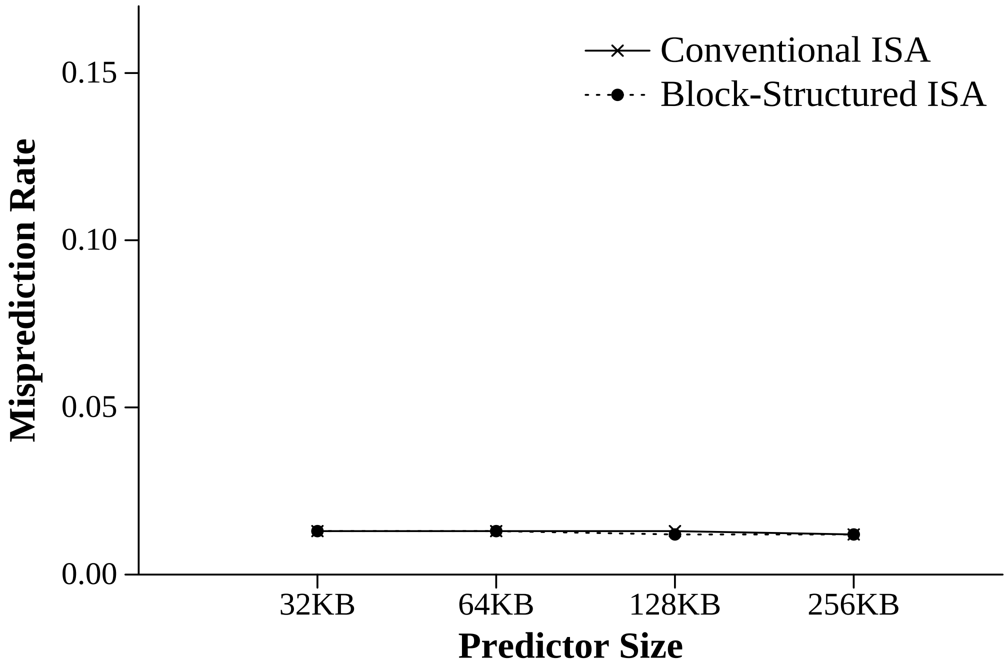

Figure 7.13: Misprediction rates for the m88ksim benchmark for block-structured ISA and conventional ISA branch predictors of size 32KB to 256KB.

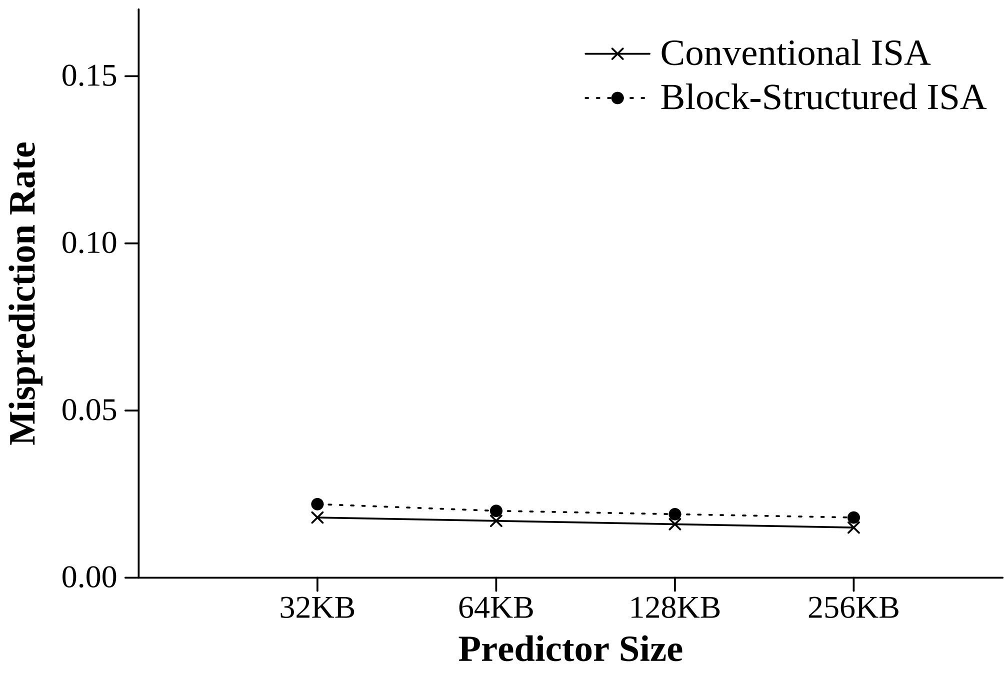

Figure 7.14: Misprediction rates for the perl benchmark for block-structured ISA and conventional ISA branch predictors of size 32KB to 256KB.

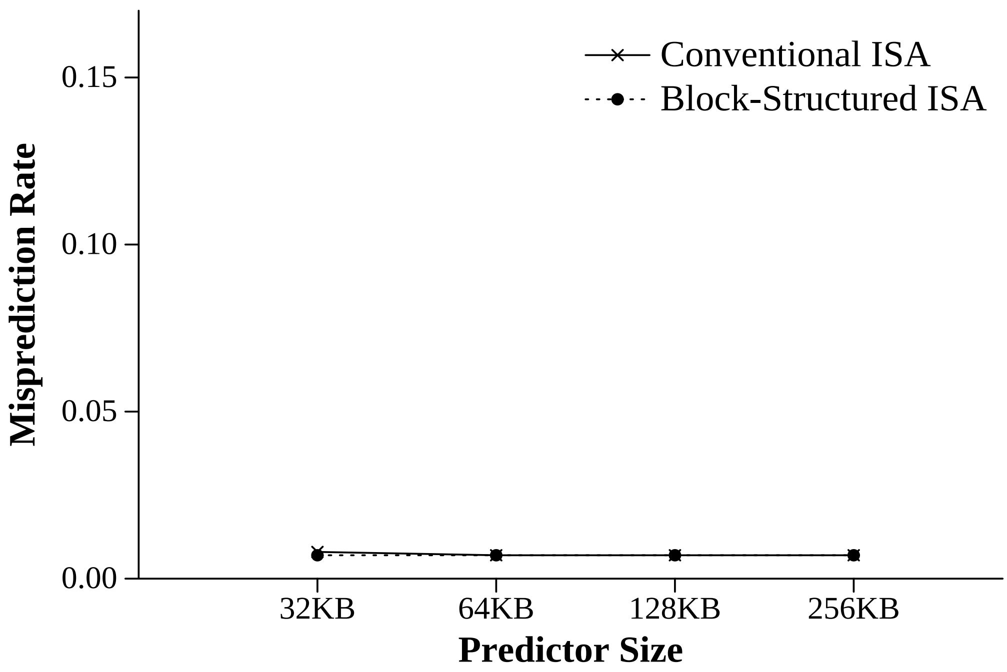

Figure 7.15: Misprediction rates for the vortex benchmark for block-structured ISA and conventional ISA branch predictors of size 32KB to 256KB.

predictor component for predicting these branches. The conventional ISA predictor has the freedom to use any predictor component for each branch it encounters because it predicts the branches one at a time. This difference is emphasized at the larger predictor sizes for compress and ijpeg. At the smaller predictor sizes, the gshare predictor component performs as well or better for many of the branches in compress and ijpeg so the hybrid predictor spends most of its time selecting the gshare component. However, as the predictor size is increased, the PAs predictor's accuracy increases significantly. As a result, the number of times for which it desirable for the hybrid predictor to switch from one component to the other as it proceeds from one branch to the next in the dynamic instruction stream increases. Because the conventional ISA predictor is able to take full advantage of this phenomenon, it is able to achieve a larger reduction in misprediction rate than the block-structured ISA predictor.

The misprediction rates for the block-structured ISA and conventional ISA branch predictors are almost identical for the m88ksim, perl, and vortex benchmarks because the branches within these benchmarks are very easy to predict. This is evidenced by the low misprediction rates achieved for all three benchmarks. The misprediction rate remains almost constant as the predictor size is increased indicating that the predictors require relatively short history lengths to capture enough information to accurately predict these branches. This negates the block-structured ISA predictor's disadvantage of having shorter history lengths than the conventional ISA predictor. Furthermore, both the gshare and PAs predictor components perform equally well in predicting these branches. As a result, the block-structured ISA predictor's reduced flexibility in switching from one component predictor to the other becomes less of a handicap. Because the major disadvantages of the block-structured ISA predictor do not come into play for these benchmarks, the block-structured ISA predictor is able to achieve almost the same prediction accuracy as that of the conventional ISA predictor.

7.5 Block Enlargement Effects

This section examines the impact on prediction accuracy of adding function inlining and the profile-based approach to enforcing the max successor restriction to the block enlargement optimization. Figure 7.16 shows the misprediction rates achieved by the gshare(13,1)/PAs(13,1) configuration of the block-structured ISA branch predictor for executables compiled with the three different block enlargement variations. The figure shows two interesting trends: function inlining provides a small improvement in prediction accuracy for half of the benchmarks and the profile-based approach to enforcing the max successor constraint significantly improves prediction accuracy for the gcc and go benchmarks.

The improvement in prediction accuracy provided by inlining may be due to the fact that the dynamic behavior of a branch within a function may vary according to the program location from which the function was called[1]. Suppose such a function with such a branch existed. If the function is not inlined, the branch predictor would have to make its predictions for that branch based on an average of all the different dynamic behavior exhibited by the branch at each of its function's call sites. By inlining the function at each of its call sites,

[1]Young and Smith observed this correlation between function call site and branch behavior and used it to improve static branch prediction accuracy [58].

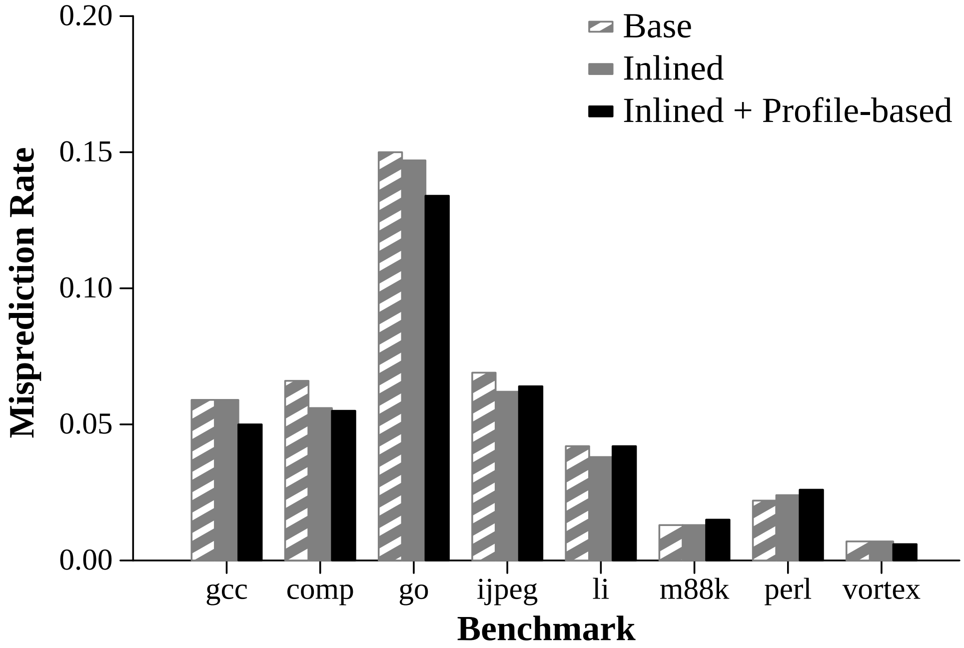

Figure 7.16: Branch misprediction rates for the three variations of the block enlargement optimization.

a separate copy of the branches would be created for each call site. This would enable the branch predictor to tailor its predictions to the specific dynamic behavior of each inlined copy of the branch.

The improvement in prediction accuracy for the gcc and go benchmarks provided by the profile-based approach to enforcing the max successor constraints was due to the profile-based approach reducing the impact of BTB misses on branch prediction accuracy. As discussed above, BTB misses reduce the accuracy of the block-structured ISA branch predictor because the predictor is constrained by a BTB miss to choosing its prediction from the two targets specified by the current block's trap operation instead of choosing its prediction from all the possible successors to the current block. The profile-based approach reduces the loss in prediction accuracy due to BTB misses by using the program profile to determine which successor on the taken side of a given block's trap operation will occur most frequently and which successor on the not taken side of the trap will occur most frequently. The profile-based approach then selects these two successors to be the targets of the trap. By setting the trap targets to be the most frequently occurring successors instead of selecting the targets at random, the profile-based approach is able to improve the accuracy of the predictions made by the block-structured ISA branch predictor during a BTB miss. This advantage led to improvements in prediction accuracies for only the gcc and go benchmarks because they were the only two benchmarks that had a significant number of BTB misses.

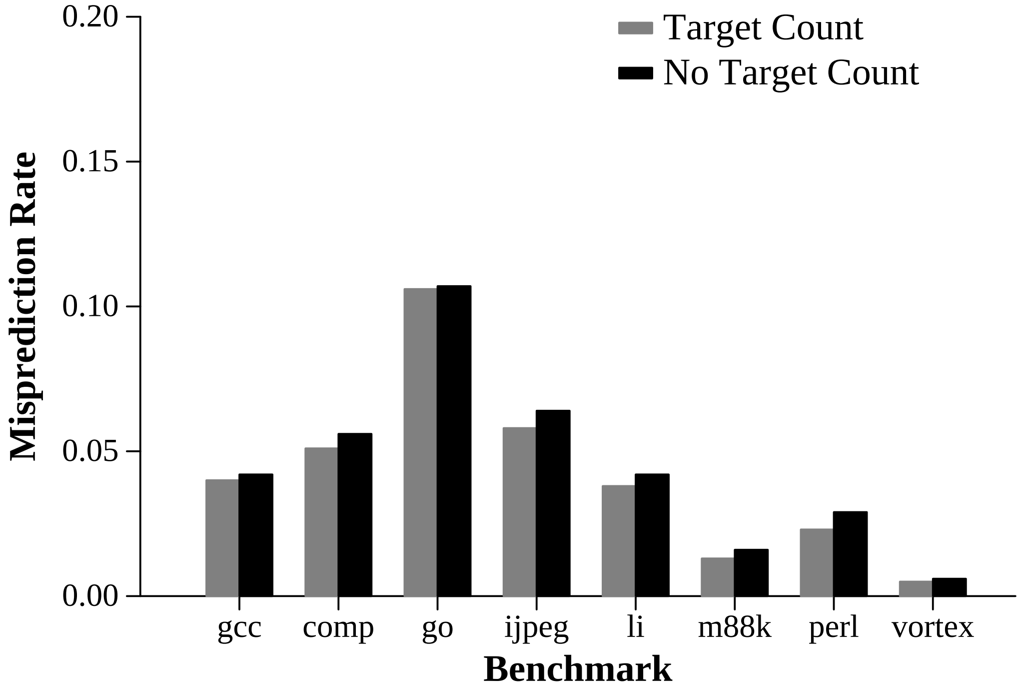

Figure 7.17: Misprediction rates for branch predictors with and without the target count extension.

7.6 Block-Structured ISA Extensions

This section examines the performance benefits of the target count and target mapping extensions to the block-structured ISA branch predictor.

7.6.1 Target Count

As discussed in chapter 6.2.2, each enlarged block is tagged with the minimum number of bits required to uniquely specify all the block's successors (i.e. the log of the number of targets for the block). The block-structured ISA branch predictor uses this information to determine how many bits to take from the predicted target index to shift into the branch history register. Figure 7.17 evaluates the performance benefit of using the block's target count to control the number of bits shifted into the branch history register. It compares the misprediction rate of a gshare(16,1)/PAs(16,1) block-structured ISA branch predictor that uses the target count extension to the misprediction rate of an identically configured predictor that always shifts in the complete target index. Ignoring the target count increased the misprediction rate by 13% which translates to a 2.2% increase in execution time.

7.6.2 Target Mapping

The target map associates each successor of a block to a unique target. This mapping is specified by the compiler and stored in the BTB. As discussed in chapter 6.3 this mapping

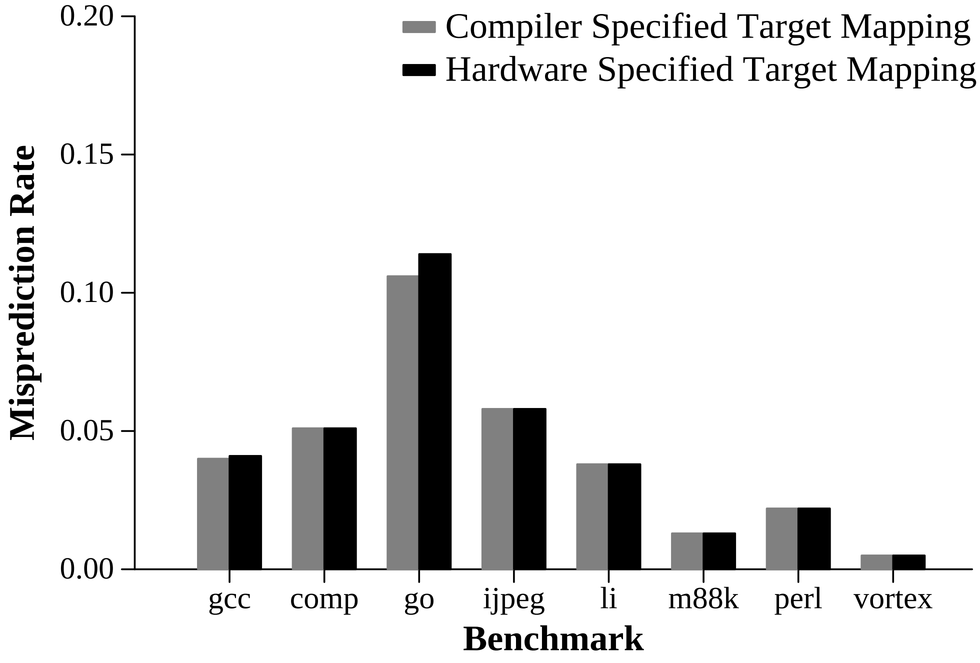

Figure 7.18: Mispredicton rates for branch predictors with and without compiler specified target mappings.

could also be generated dynamically by the hardware. The compiler was chosen to specify the mapping to guarantee that it remained constant even in the event of BTB misses. Figure 7.18 shows the performance benefit of using the compiler to specify the target mapping. It compares the misprediction rate of a gshare(16,1)/PAs(16,1) block-structured ISA branch predictor with a 2K entry BTB that uses a compiler specified target mapping to the misprediction rate of an identically configured predictor that uses a hardware specified target mapping. As expected, go and gcc are the only two benchmarks that show any change in prediction accuracy because all the other benchmarks have an insignificant number of BTB misses. The go benchmark had a 7.5% increase in misprediction rate that translated to a 3.4% increase in execution time. The gcc benchmark had a 2.5% increase in misprediction rate that translated to a .6% increase in execution time.

7.7 Overall Performance

The performance of the block-structured ISA executables compiled with inlining and the profile-based approach to enforcing max successor constraints is compared to that of the conventional ISA executables compiled with inlining. The execution times for these two sets of executables are shown in figure 7.19. Both sets of executables were the highest performing versions for their respective ISAs. The block-structured ISA branch predictor used the gshare(13,1)/PAs(13,1) configuration. The conventional ISA branch predictor used the gshare(16,1)/PAs(16,1) configuration. Both predictors are 32KB in size. The execution times for the block-structured ISA executables were on average 15% faster than those of the

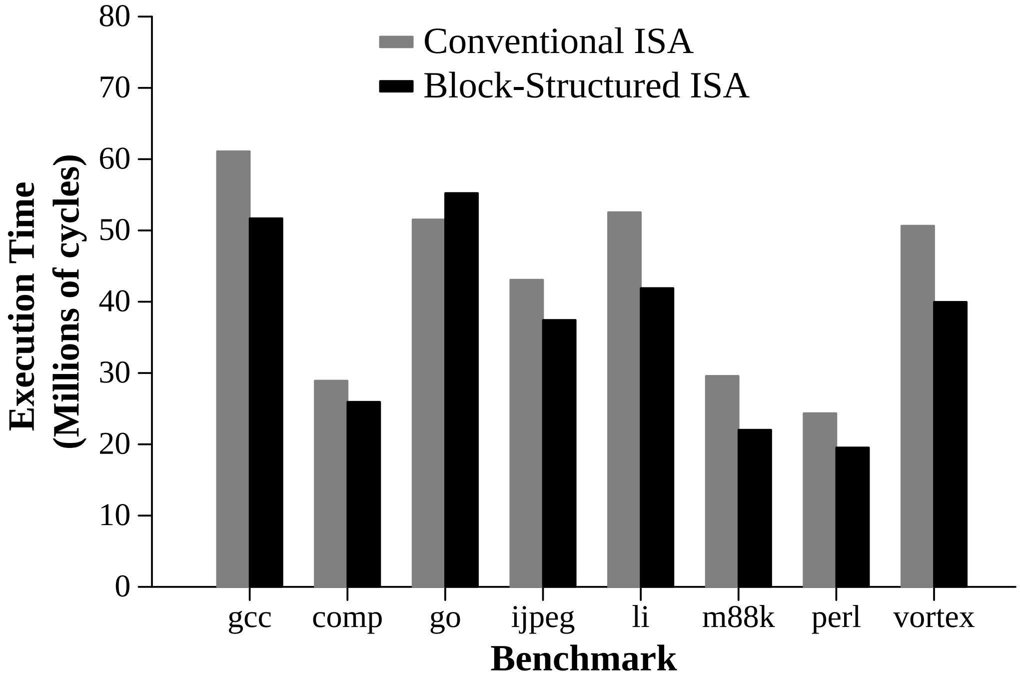

Figure 7.19: Execution times for block-structured and and conventional ISA executables when using a 32KB predictor.

conventional ISA executables.

In this comparison, as well as many others in this chapter, the go benchmark stands out as the block-structured ISA executable with the worst performance relative to its corresponding conventional ISA executable. The go benchmark performs so poorly because it includes many of the attributes that negate the performance benefits provided by block-structured ISAs and block enlargement. As was shown in previous sections, the go benchmark has a large number of icache misses and the code duplication incurred by the block enlargement optimization significantly increases that number. Furthermore, the block-structured ISA branch predictor fares the worst against the conventional ISA branch predictor for the go benchmark because of its shorter history length and the significant number of BTB misses. Because of the increased penalties for icache misses and branch mispredictions, the block-structured ISA executable performs worse than the conventional ISA executable. Further work needs to be done to address these issues.

7.8 Summary

The key results presented in this chapter are:

- The block-structured ISA branch predictor achieves a prediction accuracy that is comparable to that of the conventional ISA branch predictor. For the 32KB cost level, its misprediction rate is 11% higher than that of the conventional ISA branch predictor and that is mostly due to the shorter history register lengths that must be used by the block-structured ISA branch predictor to compensate for its larger PHT entries.

- Block-structured ISA executables do not incur more BTB misses than conventional ISA executables. In fact, because multiple blocks can be combined into a single enlarged block, the block-structured ISA executables often incur significantly fewer BTB misses than the conventional ISA executables.

- The mispredicted branch resolution time is 12% longer for block-structured ISA executables than for conventional ISA executables. As a result, a mispredicted branch incurs a larger penalty for a block-structured ISA executable than for a conventional ISA executable.

- For real branch predictors of size 32KB, block-structured ISAs show a performance improvement of 15% as compared to conventional ISAs.

CHAPTER 8

Block Enlargement for Scientific Code — Measurements and Analysis

This chapter examines the performance benefit of block enlargement optimizations for scientific code. A separate chapter is devoted to the performance of scientific code because scientific code differs in three ways from the integer code studied in the previous chapters that make scientific code more amenable to block enlargement. First, the basic blocks in scientific code are almost twice as large as those in integer code. As a result, the block enlargement optimization is able to create even larger enlarged blocks than those created for integer code. In fact, because the basic blocks for scientific code are so large (on the order of ten instructions), machines with issue widths of 32 instructions are considered in this chapter in addition to the 16 wide issue machines considered in the previous chapters. Second, the conditional branches in scientific code can be predicted with a very high degree of accuracy. This eliminates the negative effect on performance due to branch mispredictions which was discussed in chapter 7. Third, scientific programs are significantly smaller than their integer counterparts and as a result, have significantly smaller icache requirements. Because the icache requirements for scientific programs are so small, the code duplication incurred by block enlargement will have little effect on performance.

8.1 The Base Block Enlargement Optimization

Using the block enlargement optimization described in section 5.1, the compiler generated block-structured ISA executables for the SPECfp95 benchmarks for a 16 and a 32 wide issue machine [1]. Figure 8.1 compares the performance of the block-structured ISA executables to the performance of the corresponding conventional ISA executables on a 16 wide issue machine under the assumption of perfect branch prediction. The graph shows the total number of cycles required to execute each benchmark. Figure 8.2 compares the average block sizes for the two sets of executables.

The block-structured ISA executables for the 16 wide issue machine showed little performance improvement over the conventional ISA executables, because the block enlargement

[1]The executables generated for the 32 wide issue machine were compiled with the maximum block size set to 32.

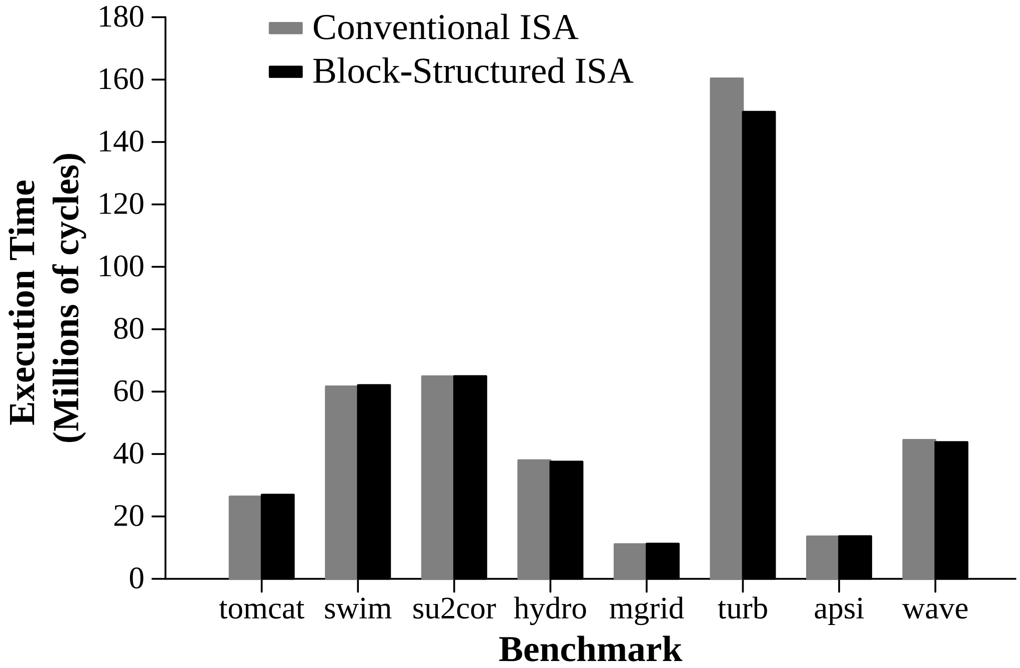

Figure 8.1: Performance comparison of block-structured ISA executables to conventional ISA executables for a 16 wide machine with perfect branch prediction.

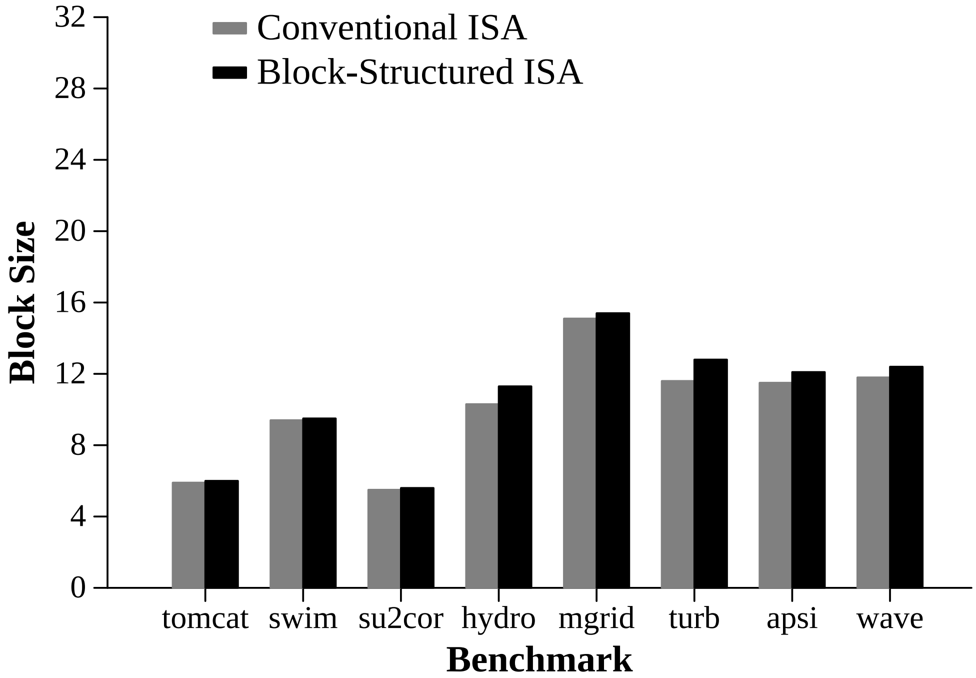

Figure 8.2: Average block sizes for block-structured and conventional ISA executables for a 16 wide machine.

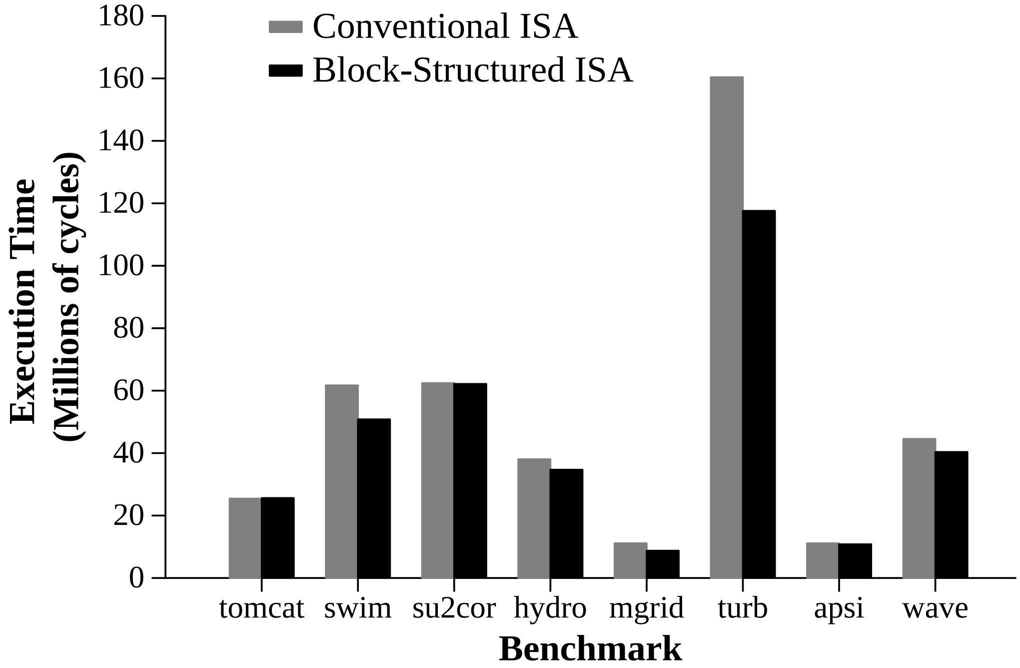

Figure 8.3: Performance comparison of block-structured ISA executables to conventional ISA executables for a 32 wide machine with perfect branch prediction.

optimization was unable to significantly increase the average block size. The average block size for the block-structured ISA executables is 10.6 as compared to 10.1 for the conventional ISA executables. The maximum block size constraint was the major reason why the block enlargement optimization was unable to significantly increase the block size. Given that the average block size before block enlargement was ten instructions, the majority of the blocks formed by block enlargement had more than sixteen instructions. The two exceptions to this observation were tomcatv and su2cor. Both benchmarks had average block sizes of less than six instructions for the conventional ISA executables, but showed little increase in block size for the block enlarged executables for the 16 wide machine. The major reason why the block enlargement optimization was unable to significantly increase the block size for these two benchmarks was the call/return constraint. The call/return constraint accounted for a third of the packet breaks in these two benchmarks.

Figure 8.3 compares the performance of the block-structured ISA executables to the performance of the corresponding conventional ISA executables on a 32 wide issue machine under the assumption of perfect branch prediction. Figure 8.4 compares the average block sizes for the two sets of executables.

The block-structured ISA executables for the 32 wide machine achieved an average reduction in execution time of 11%. The average block size for these executables is 15.8, a 34% increase in size over the conventional ISA block size of 11.7. The major reason why the block enlargement optimization was not able to further increase the block size was due to the occurrence of call/returns. Inlining may lead to further increases in block size. Because the

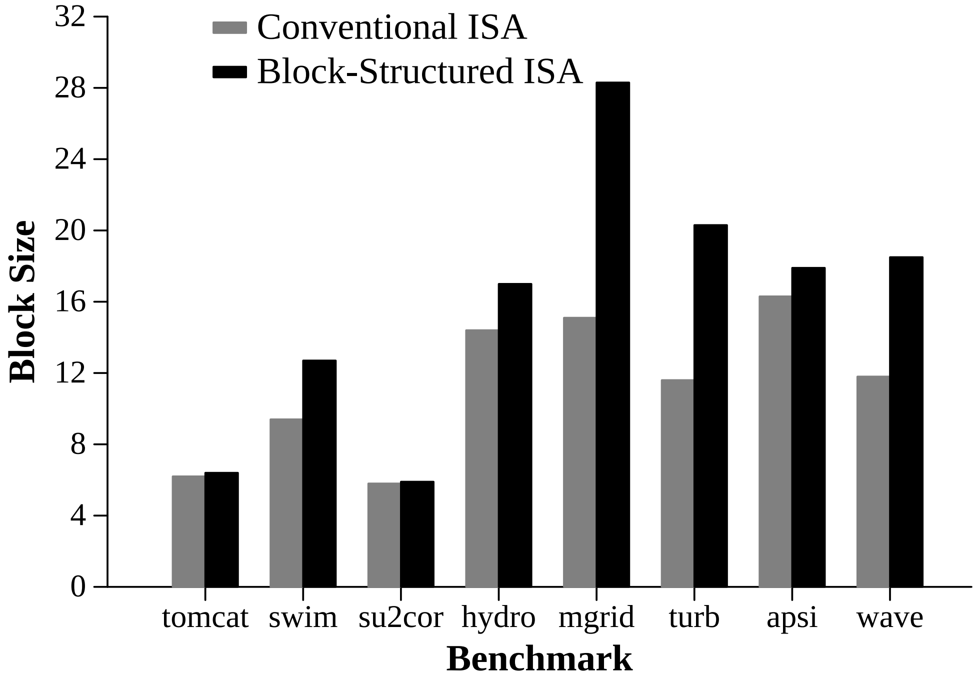

Figure 8.4: Average block sizes for block-structured and conventional ISA executables for a 32 wide machine.

SPECfp95 benchmarks are so small, very aggressive levels of inlining can be performed for these benchmarks without significantly affecting the icache hit rate, increasing the likelihood that inlining will lead to significant performance gains.

8.2 ICache Performance

The number of icache miss cycles was measured for both the block-structured and conventional ISA executables, running on both the 16 and 32 wide issue machines. In all cases, the number of icache miss cycles was less than 1% of the total execution time. The code duplication due to block enlargement had no significant effect on performance. This result is not surprising given the aforementioned observation that scientific programs have extremely low icache requirements.

8.3 Branch Prediction Performance

This section considers the effect of branch prediction on performance for the SPECfp95 benchmarks. The branch predictors used for the experiments in this section were the same 32KB configurations used in chapter 7 (gshare(13,1)/PAs(13,1) for the block-structured ISA predictor and gshare(16,1)/PAs(16,1) for the conventional ISA branch predictor). Figure 8.5 compares the prediction accuracy for the conventional ISA and the block-structured ISA predictor for the 32 wide machine. For most of the benchmarks, the misprediction rate for all

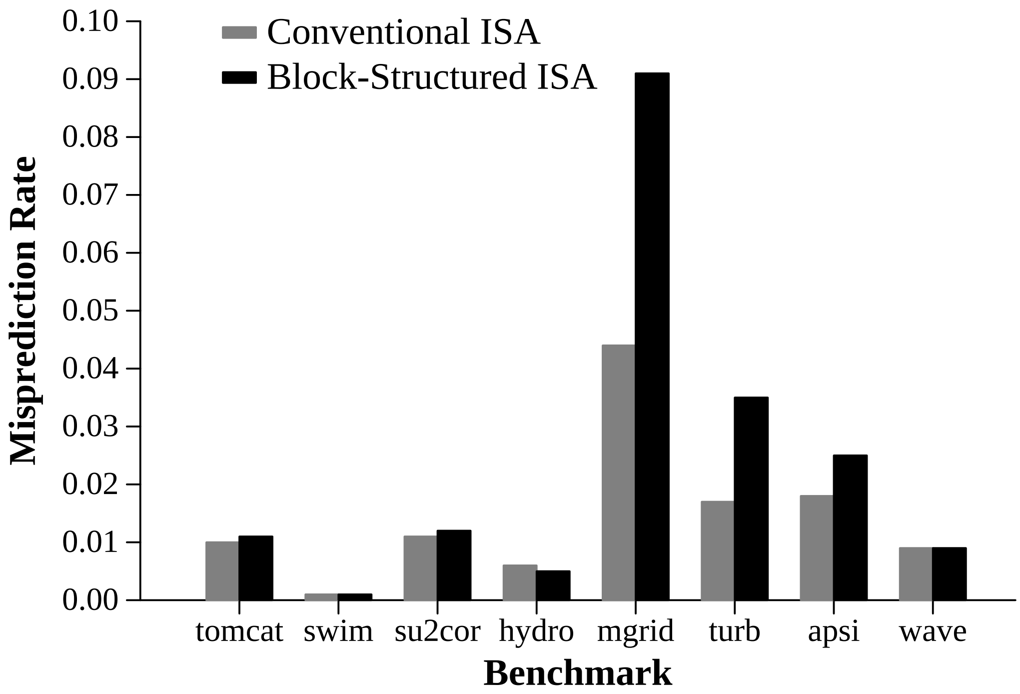

Figure 8.5: Misprediction rates for block-structured and conventional ISA executables.

three cases is very low. The notable exceptions to this rule are the mgrid and turb3d benchmarks. The block-structured ISA branch predictor show a notable increase in misprediction rate for these two benchmarks. The key reason for this increase in misprediction rate is that the mgrid and turb3d benchmarks both perform calculations on a three dimensional space of size 16x16x16. As a result, many of the loops in these benchmarks iterate for exactly sixteen iterations. The conventional ISA predictor's history register is sixteen bits long, which is just long enough to predict such loop branches with 100% accuracy. The block-structured ISA predictor's history register is thirteen bits long [2] which is too short to enable the predictor to catch the iteration in which the loop branch exits the loop.

Figure 8.6 shows the effect of real branch prediction on execution time for the block-structured ISA executables executing on a 32 wide machine. For most benchmarks, the execution time with real branch prediction is comparable to that with perfect branch prediction which is not surprising given the low misprediction rates achieved for the SPECfp95 benchmarks. The benchmarks that show a significant increase in execution time with real branch prediction are turb3d, tomcatv, and su2cor. The increase in execution time for the turb3d benchmark is due to the increase in misprediction rate discussed above. On the other hand, the tomcatv and su2cor benchmarks show a significant increase in execution time despite achieving a low branch misprediction rate for conditional branches because they have a significant number of indirect branches whose targets are frequently mispredicted. Using a target cache [5] to predict the targets of these indirect branches may significantly improve

[2] The shorter history register length is due to the increased size of the pattern history table entries.

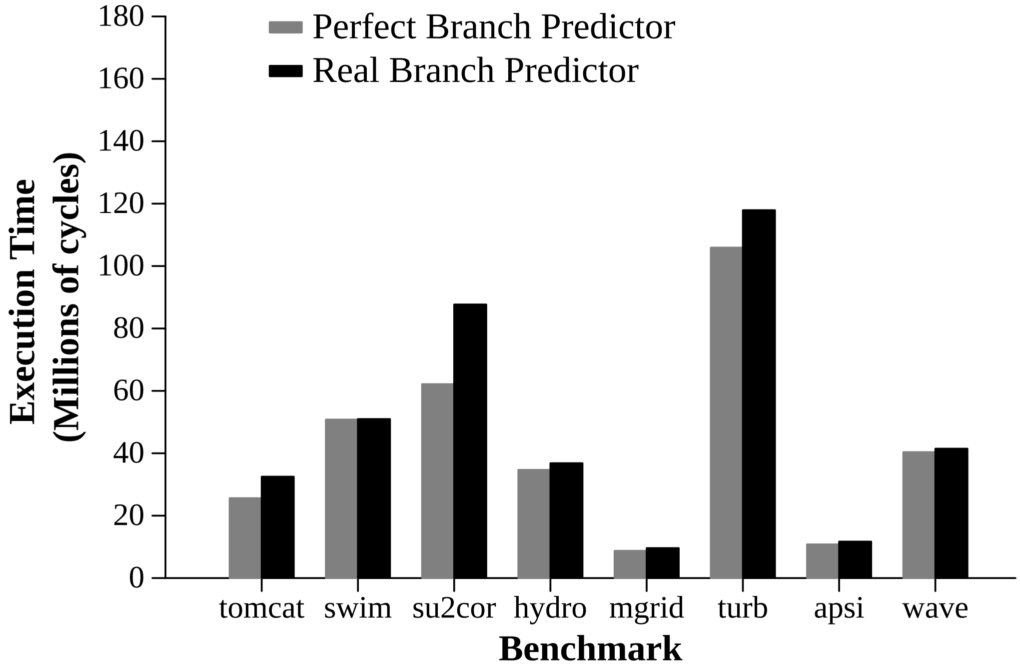

Figure 8.6: Execution times for block-structured ISA executables running on a 32 wide machine with and without perfect branch prediction.

the performance of these two benchmarks.

8.4 Future Directions

The performance improvement achieved by the block enlargement optimization for the SPECfp95 benchmarks was very disappointing. As discussed at the beginning of this chapter, given the large blocks, predictable branches, and small icache requirements that are characteristic of scientific code, the block enlargement optimization should be very effective for scientific code. However, these initial results point to three directions to investigate to significantly improve performance.

1. The problem size should be increased. The input data sets used in the experiments were modified to reduce the problem size so as to reduce the time required to simulate the benchmarks. Increasing the problem size will increase performance in two ways. First, increasing the problem size will increase the ratio between the parallel portion of the benchmark (i.e. the loops) to the sequential portion of the benchmark. The majority of the large blocks in a benchmark are found in its parallel portion. As a result, the block enlargement optimization is more effective on this portion of the benchmark. The majority of the procedure calls and small blocks in a benchmark are found in its sequential portion. As a result, the block enlargement optimization is less effective on this portion of the benchmark. By increasing this ratio, the fraction of executed instructions that are from the parallel portion is increased, increasing the

performance benefit of block enlargement. Second, increasing the problem size will increase the number of iterations executed for many loops in the benchmark, reducing impact of the branch prediction problems discussed above and as a result, increasing overall prediction accuracy.

2. One major constraint that prevented block enlargement from creating larger enlarged blocks was the max size constraint. Because the initial basic blocks were so large, the block enlargement optimization was unable to combine many of the blocks in the benchmarks because the resulting enlarged blocks from such combinations would have violated the max size constraint. One way to reduce the performance impact of this constraint would be to enable the block enlargement optimization to combine a block with only a part of its successor block rather than its entire successor block. With this partial block enlargement, the opportunities for applying the block enlargement optimization to the SPECfp95 benchmarks will be significantly increased, increasing the average size of the enlarged blocks for these benchmarks.

3. Another major constraint that prevented block enlargement from creating larger enlarged blocks was the occurrence of call and return instructions. As discussed above, inlining may be an effective solution to this problem. Because the SPECfp95 benchmarks are so small, more aggressive levels of inlining can be performed without affecting the icache hit rate than can be performed for the SPECint95 benchmarks, increasing the likelihood that inlining will lead to significant performance gains.

CHAPTER 9

Conclusion

9.1 Contributions

To achieve higher levels of instruction level parallelism, processors are being built with wider issue widths and larger numbers of functional units. To take full advantage of this increased execution bandwidth, such processors must be able to fetch instructions at a high enough rate to feed this bandwidth. However, the instruction fetch mechanisms for such processors continue to fetch only a single basic block per cycle. Because the average basic block size for integer programs is approximately five instructions, processors that aim to exploit aggressive levels of instruction level parallelism (on the order of sixteen instructions per cycle) must be able to fetch multiple basic blocks each cycle.

To meet the need for higher instruction fetch rates, this dissertation examined the performance benefit of using the block enlargement optimization to increase the atomic block size of a block-structured ISA. Through the block enlargement optimization, the atomic blocks of the block-structured ISA are combined to form larger atomic blocks. As a result, larger units of work can be fetched from the icache each cycle without having to fetch non-consecutive cache lines as is done in past hardware-based approaches to the instruction fetch problem. Furthermore, the block-structured ISA provides support for the use of dynamic branch prediction in selecting the successor block, rather than restricting the processor to static branch prediction as is required by past software-based approaches.

This dissertation defined one instance of a block-structured ISA and implemented a compiler targeted to that ISA and a simulator to model the performance of a sixteen wide issue, dynamically scheduled processor that implements that ISA. Using this compiler and simulator, the performance of programs executing on a block-structured ISA processor was compared to the performance of programs executing on a conventional ISA processor for the SPECint95 benchmarks. The block-structured ISA processors achieved a 28% performance improvement over conventional ISA processors when perfect branch prediction was assumed and a 15% performance improvement when real branch prediction was used. These performance improvements were due to the block enlargement optimization increasing the average block size from 5.8 instructions to 10.6 instructions.

This dissertation also presented the design of a dynamic branch predictor for block-structured ISAs that was shown to have a misprediction rate that was only 11% larger than that of an aggressive branch predictor of the same size for a conventional ISA. The majority of this difference in prediction accuracy was due to the block-structured ISA predictor having PHT entries that were larger than those of the conventional ISA predictor. As a result, when considering predictors of equal size, the conventional ISA predictor was able to record more branch history than the block-structured ISA predictor which enabled it to make more accurate predictions.

Finally, this dissertation examined the performance benefit of block enlargement for the SPECfp95 benchmarks. For a 32 wide issue HPS processor, the block-structured ISA executables achieved an 11% increase in performance over the conventional ISA executables and a 34% increase in block size (11.7 instructions to 15.8 instructions). The max size and call/return constraints were shown to be the major obstacles to higher levels of block enlargement.

This dissertation focused on the improvements in instruction fetch rate provided by block-structured ISAs and did not consider the implementation benefits provided by block-structured ISAs. By reducing the hardware complexity of various microarchitectural mechanisms, block-structured ISAs enable the implementation of extremely wide issue processors. The performance benefits of these features would further increase the performance advantage of block-structured ISA processors over conventional ISA processors. In addition, as new algorithms are developed that increase the instruction level parallelism in programs that use them, the ability to implement wide issue processors as well as increasing the instruction fetch rate for such processors will become even more important.

9.2 Future Directions

The dissertation was only a first step in studying the performance benefits of block-structured ISAs. Future directions for further increasing the performance of block-structured ISAs include:

- Block-structured ISAs still leave a significant fraction of the total fetch bandwidth unused. Further performance improvements can be achieved by extending the block enlargement optimization so that even larger blocks are formed. As shown in chapter 5.3, the major obstacle to higher levels of block enlargement is the occurrence of calls and returns in the program. Function inlining was shown to provide some performance improvement. A more effective solution might be to use a limited form of function inlining where the block enlargement optimization is given the freedom to combine a block that ends in a function call with the first block of the called function. A similar solution can be used for blocks that end in returns, but because such blocks may have an enormous number of control flow successors, the block enlargement performed for such blocks must be carefully controlled.

- This dissertation did not investigate the tradeoffs between using trap operations in place of fault operations. The key difference between using a trap operation instead of a fault operation is that a mispredicted trap operation does not cause the work in its associated block to be discarded. The advantage provided by this difference is that the work in the trap's block that is on the correct path does not have to be fetched, issued, and executed again when the trap is mispredicted. The disadvantage is that the work in the trap's block that is on the wrong path must somehow be suppressed when the trap is mispredicted. This could possibly be done by adding additional code to the trap's successor block that negates the effect of the included code from the wrong path as is done in trace scheduling and superblocks [14, 4, 22]. Given these tradeoffs, there may be situations in which it is better for the block enlargement optimization to insert a trap operation instead of a fault operation.

- Further increases in prediction accuracy for block-structured ISA branch predictors can be achieved by:

  1. Thoroughly exploring the design space to determine the best configuration for the branch predictor studied in this dissertation
  2. Searching for more efficient implementations of the branch predictor so that the length of its branch history registers are comparable to that of an equal-sized conventional ISA predictor
  3. Extending the predictor so that a different predictor component can be used for each branch that is being predicted in a given cycle

- This dissertation did not investigate the performance benefit of including predicated execution in a block-structured ISA. By eliminating branches (in particular, hard to predict branches), predicated execution will help both conventional and block-structured ISAs. However, because the average amount of time required to resolve a mispredicted branch tends to be higher for block-structured ISAs, predicated execution will provide a larger performance benefit for block-structured ISAs than for conventional ISAs.

- As discussed in chapter 8.4, the performance benefit achieved by the block enlargement optimization for the SPECfp95 benchmarks may be significantly improved by increasing the problem size simulated to increase the parallel portion of the benchmarks, using partial block enlargement to reduce the effect of the max block size constraint, and aggressively inlining function calls to reduce the effect of the call/return constraint..

APPENDICES

APPENDIX A

The Basic Block Fetch Bottleneck

This appendix illustrates the performance bottleneck created when a wide issue processor is constrained to fetching at most one basic block per cycle. The following figures illustrate the performance of a sixteen wide issue, dynamically scheduled processor with perfect branch prediction executing the eight SPECint95 benchmarks. The figures plot the execution time and average packet size achieved by the processor as the number of basic blocks that could be fetched each cycle was increased from one to four.

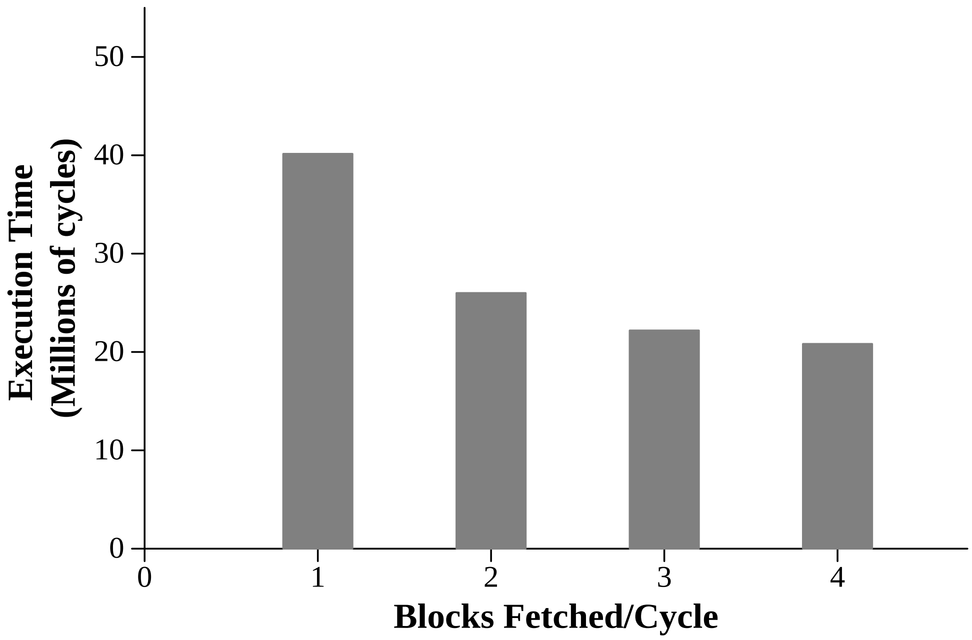

Figure A.1: Execution times for the gcc benchmark.

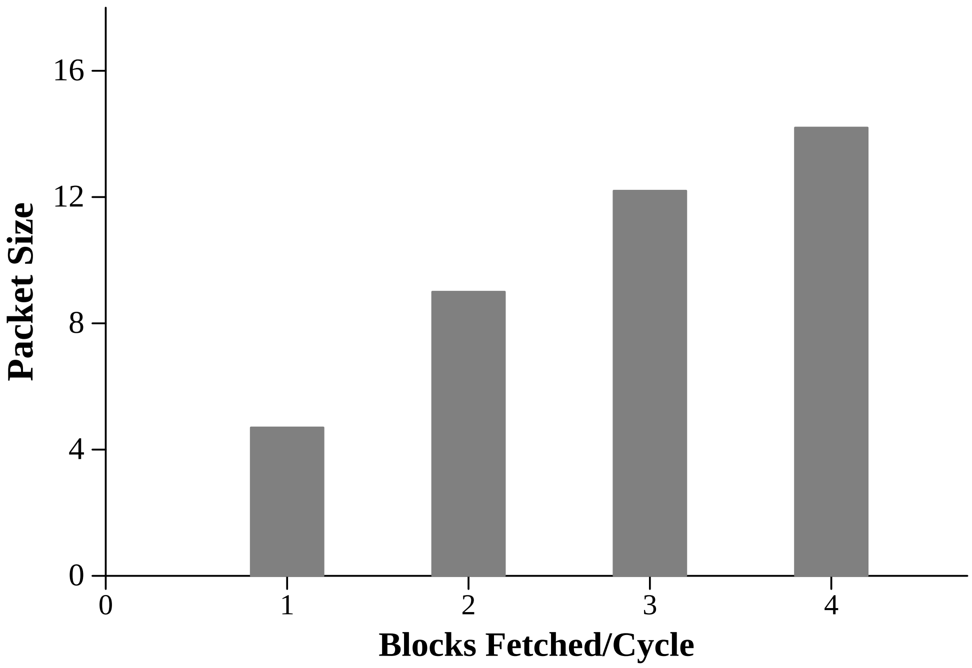

Figure A.2: Average packet sizes for the gcc benchmark.

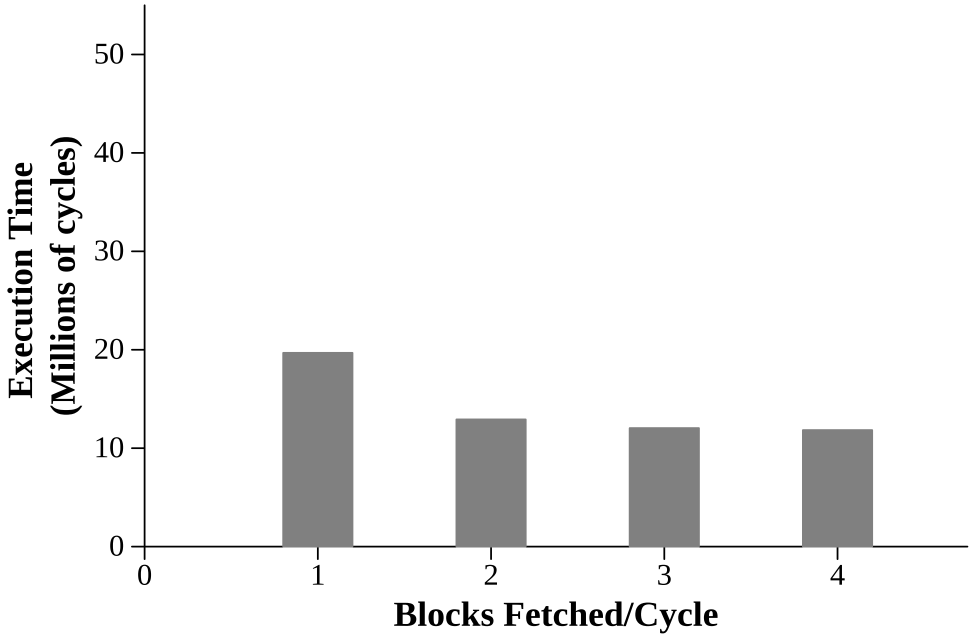

Figure A.3: Execution times for the compress95 benchmark.

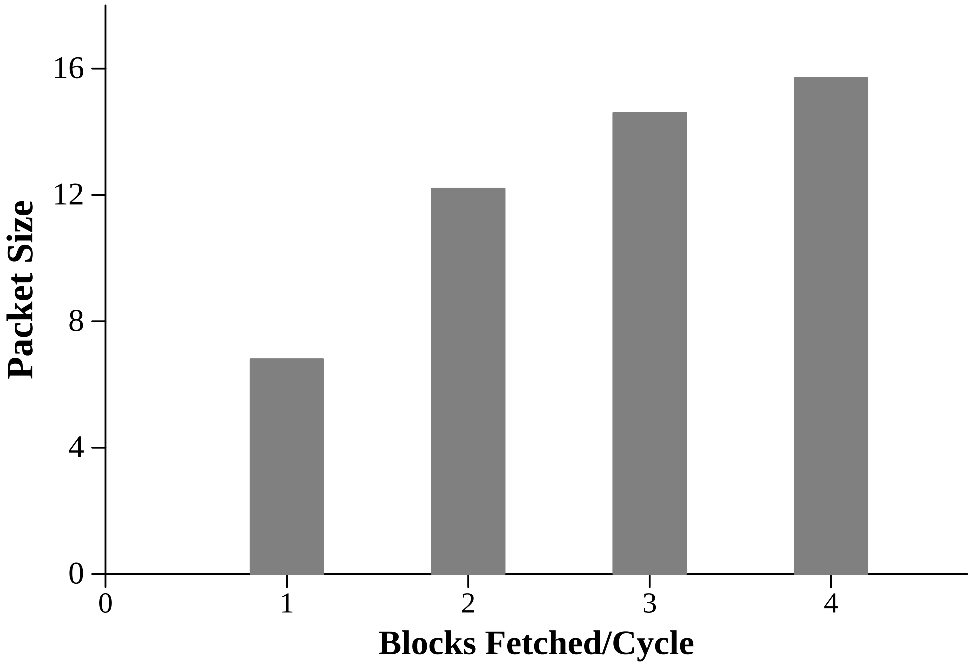

Figure A.4: Average packet sizes for the compress95 benchmark.

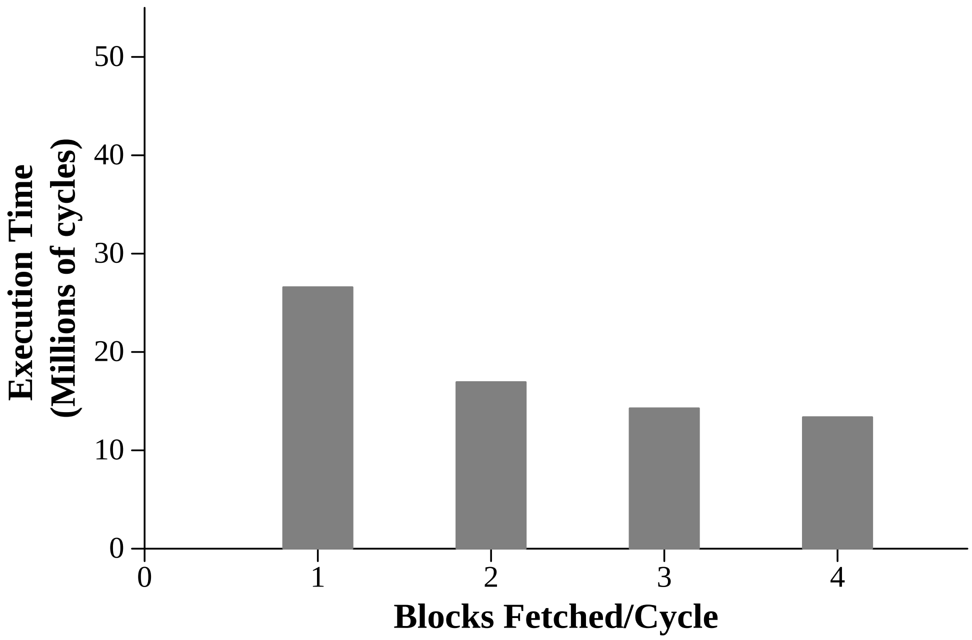

Figure A.5: Execution times for the go benchmark.

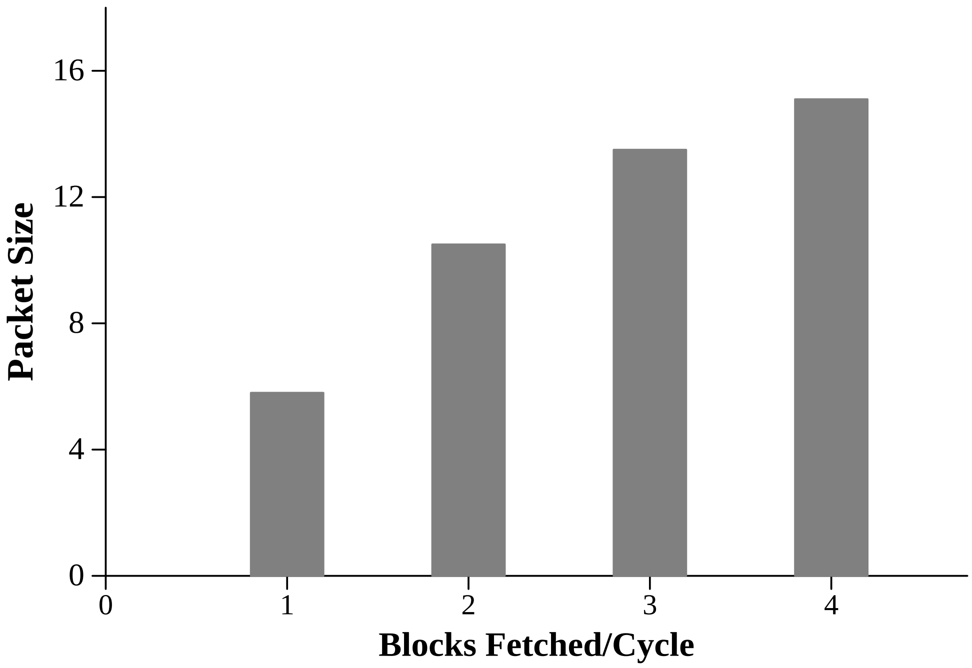

Figure A.6: Average packet sizes for the go benchmark.

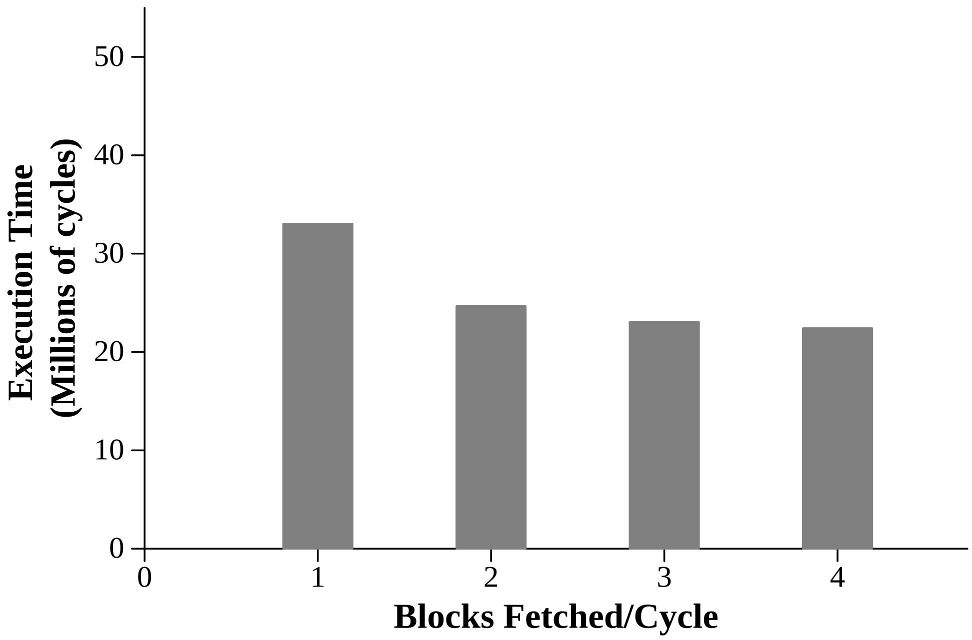

Figure A.7: Execution times for the ijpeg benchmark.

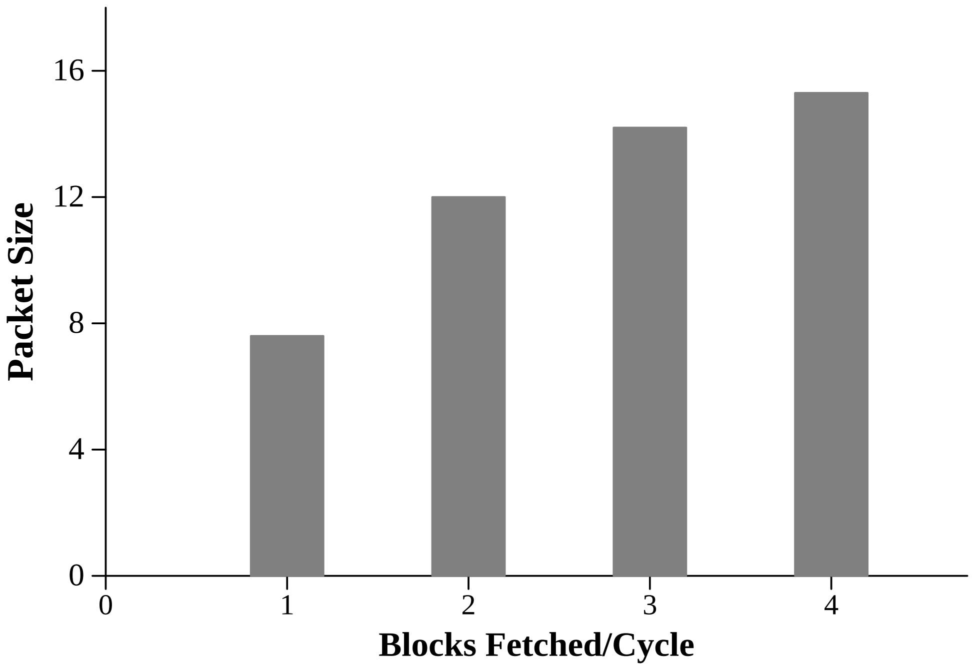

Figure A.8: Average packet sizes for the ijpeg benchmark.

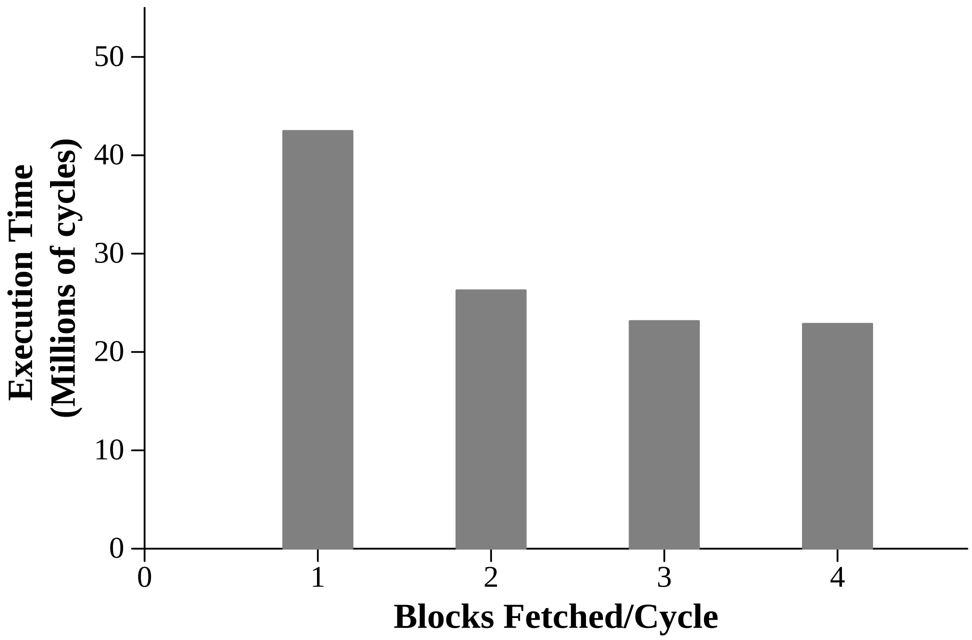

Figure A.9: Execution times for the xlisp benchmark.

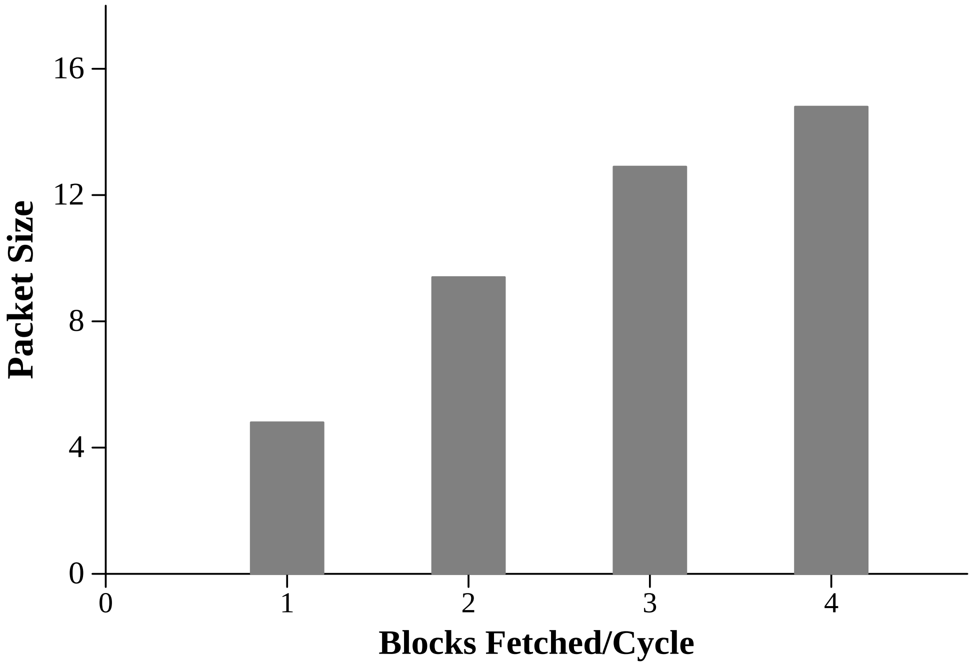

Figure A.10: Average packet sizes for the xlisp benchmark.

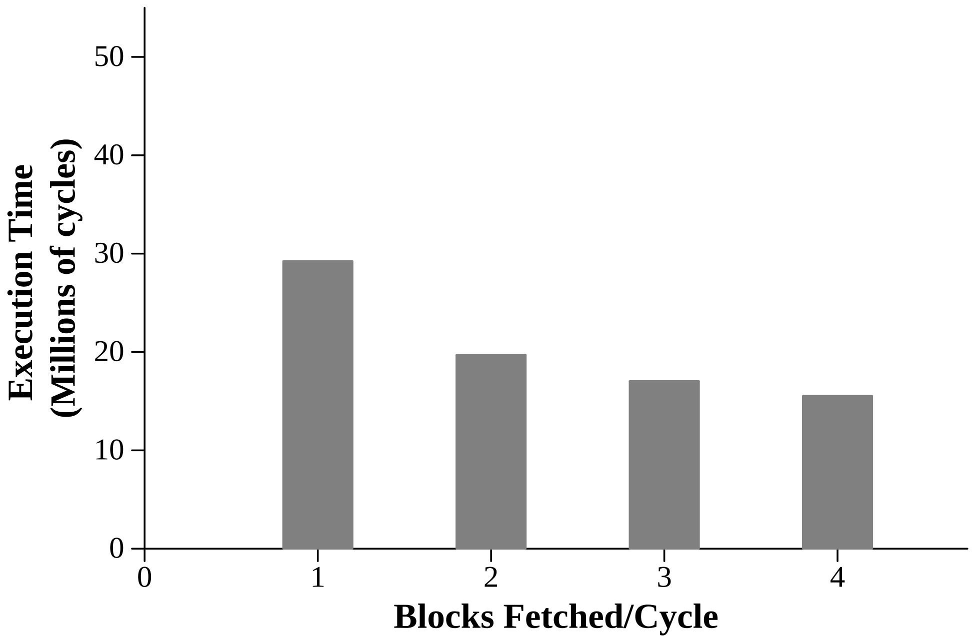

Figure A.11: Execution times for the m88ksim benchmark.

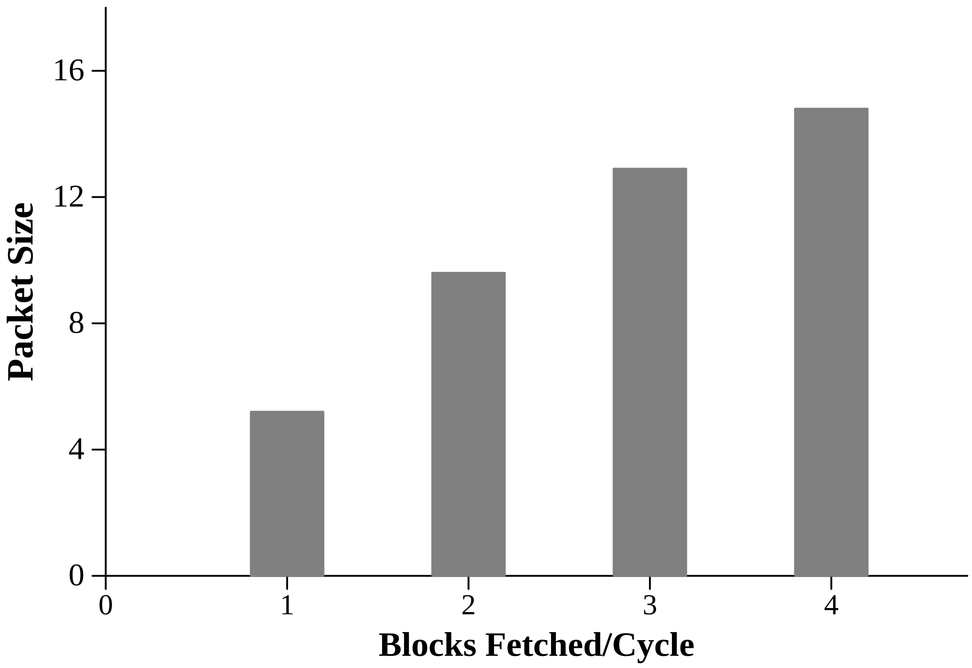

Figure A.12: Average packet sizes for the m88ksim benchmark.

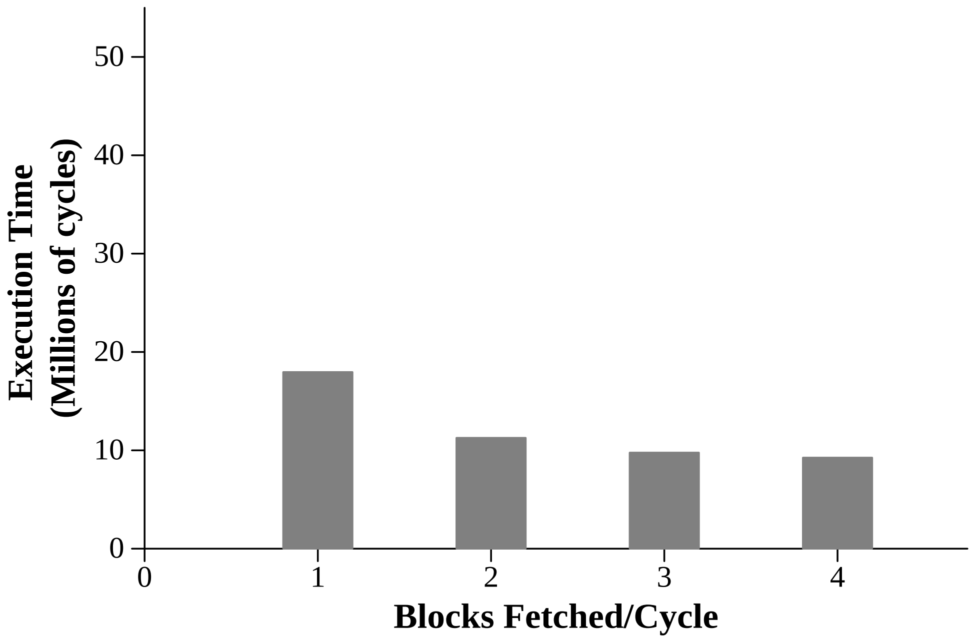

Figure A.13: Execution times for the perl benchmark.

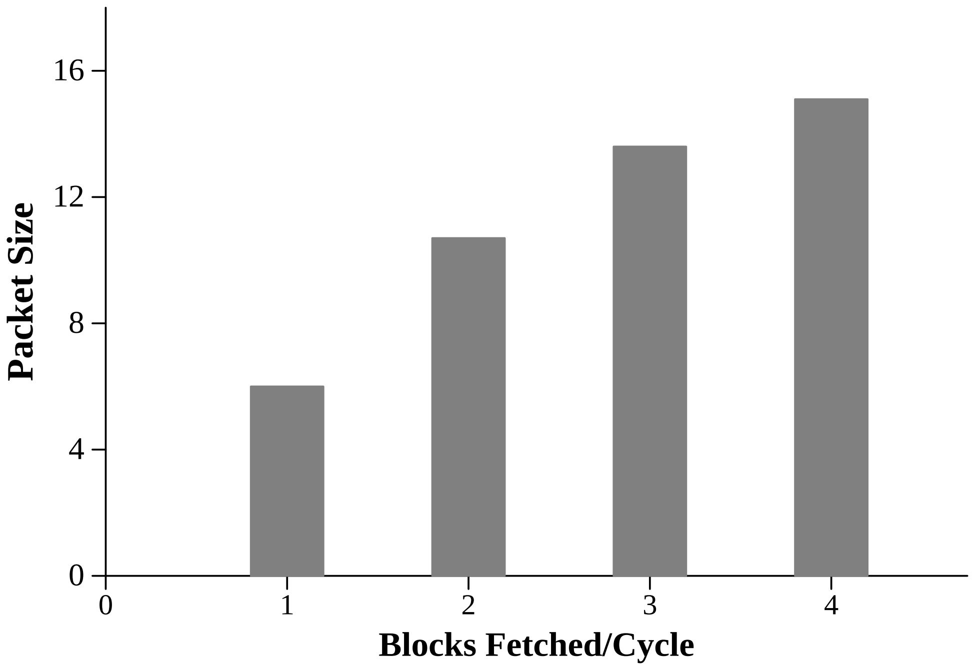

Figure A.14: Average packet sizes for the perl benchmark.

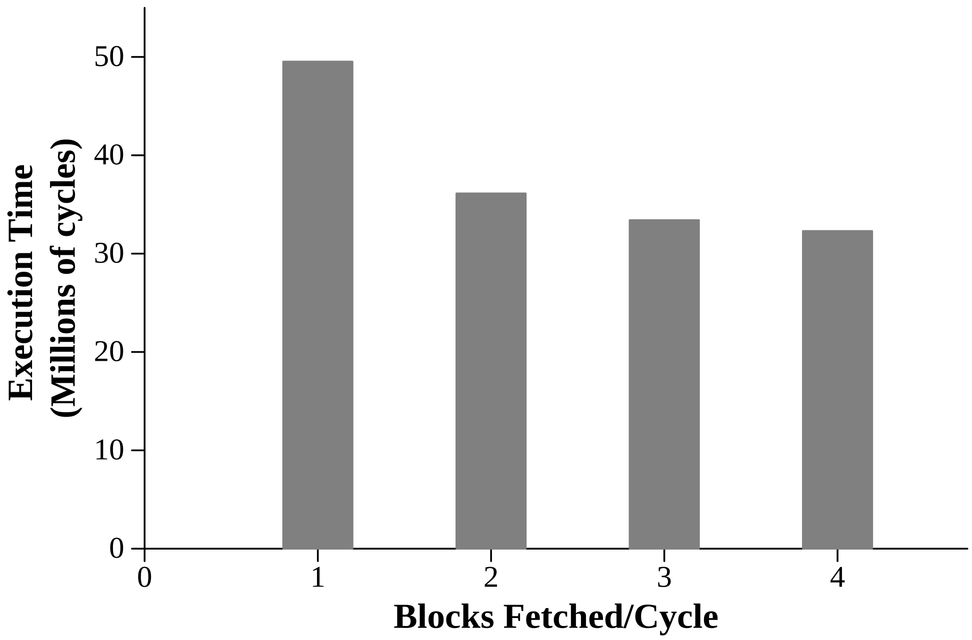

Figure A.15: Execution times for the vortex benchmark.

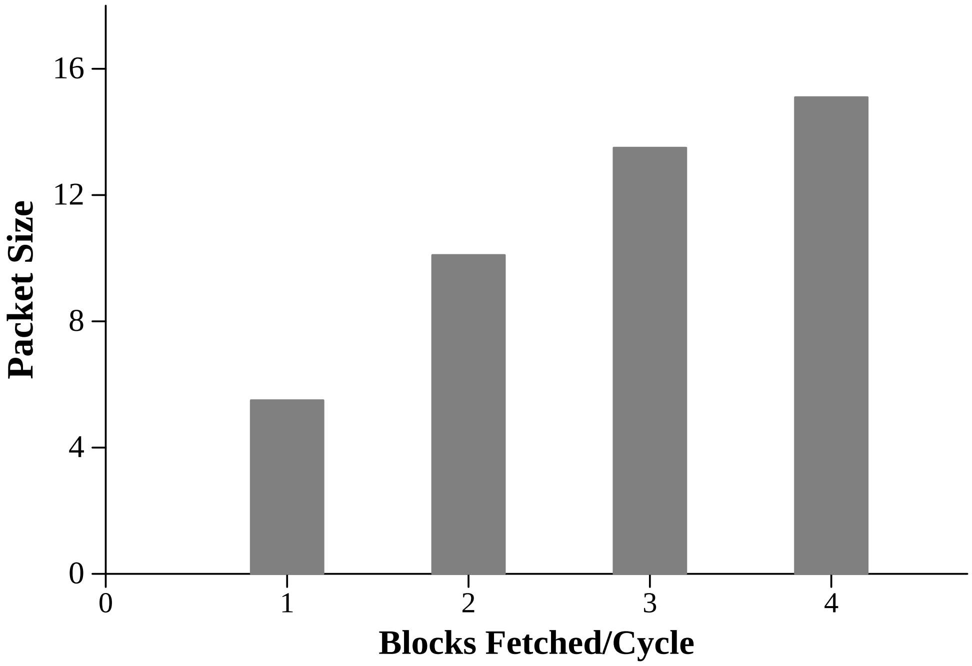

Figure A.16: Average packet sizes for the vortex benchmark.

APPENDIX B

The SPECint95 Experimental Data Sets

This section lists the test and training data sets used to execute the SPECint95 benchmarks that were modified from the data sets distributed with the benchmarks.

B.1 Compress Test Data Set — 30KB.in

The compress test data set was modified to generate a 30KB array of random data to be compressed. The seed for the random number generator was taken from one of the SPEC reference data sets. The following is a listing of 30KB.in:

```
30000   q 2131
```

B.2 Compress Test Data Set — 300KB.in

The compress test data set was modified to generate a 300B array of random data to be compressed. The seed for the random number generator was taken from one of the SPEC reference data set. The following is a listing of 300B.in:

```
300 e 2231
```

B.3 Vortex Test Data Set — test.in

The search parameters in the vortex test data set were modified to reduce the running time of the benchmark. The following is a listing of test.in:

```
MESSAGE_FILE   test.msg
OUTPUT_FILE    test.out
DISK_CACHE     bmt.dsk
RENV_FILE      bendian.rnv
WENV_FILE      bendian.wnv
PRIMAL_FILE    vortex.pml
PARTS_DB_FILE  parts.db
DRAW_DB_FILE   draw.db
```

```
EMP_DB_FILE    emp.db
PERSONS_FILE   persons.1k
PART_COUNT     100
OUTER_LOOP     2
INNER_LOOP     4
LOOKUPS        10
DELETES        10
STUFF_PARTS    10
PCT_NEWPARTS   50
PCT_LOOKUPS    25
PCT_DELETES    50
PCT_STUFFPARTS 100
TRAVERSE_DEPTH 5
FREEZE_GRP     1
ALLOC_CHUNKS   10000
EXTEND_CHUNKS  5000
DELETE_DRAWS   1
DELETE_PARTS   0
QUE_BUG        1000
VOID_BOUNDARY  67108864
VOID_RESERVE   1048576
```

B.4 Vortex Test Data Set — profile.in

The vortex training set was generated by modifying the the search parameters in the vortex test data set as well as changing the input persons file. The following is a listing of profile.in:

```
MESSAGE_FILE   profile.msg
OUTPUT_FILE    profile.out
DISK_CACHE     bmt.dsk
RENV_FILE      bendian.rnv
WENV_FILE      bendian.wnv
PRIMAL_FILE    vortex.pml
PARTS_DB_FILE  parts.db
DRAW_DB_FILE   draw.db
EMP_DB_FILE    emp.db
PERSONS_FILE   persons.15
PART_COUNT     10
OUTER_LOOP     1
INNER_LOOP     1
LOOKUPS        10
DELETES        10
STUFF_PARTS    10
PCT_NEWPARTS   50
PCT_LOOKUPS    25
PCT_DELETES    50
PCT_STUFFPARTS 100
TRAVERSE_DEPTH 1
FREEZE_GRP     1
ALLOC_CHUNKS   10000
EXTEND_CHUNKS  5000
```

```
DELETE_DRAWS   1
DELETE_PARTS   0
QUE_BUG        1000
VOID_BOUNDARY  67108864
VOID_RESERVE   1048576
```

APPENDIX C

The SPECfp95 Experimental Data Sets

This section describes the test data sets used to execute the SPECfp95 benchmarks. These data sets were based upon the reference data sets distributed with the benchmarks. The data sets were chosen so that a complete run of each benchmark lasted no more than a few hundred million instructions. Some of the benchmarks had the problem parameters hardcoded into their source code. In those cases, the benchmark source code was modified appropriately.

C.1 Tomcatv

The tomcatv test data set was modified so that the size of the mesh generated was reduced from 513x513 to 129x129. The following is a listing of the input file:

```
129,5,0.0d0,1.0d0,.1d0
```

C.2 Swim

The swim test data set was modified so that the size of the plane simulated was reduced from 512x512 to 128x128 and the iteration count was reduced from 10 to 1. The following is a listing of the input file:

```
20.
.25E5
.25E5
1.E6
.001
1
1
512
512
```

C.3 Su2cor

The su2cor test data set was modified so that the size of the grid simulated was reduced from 8x8x8x16 to 4x4x4x8. The following is a listing of the input file:

```
4 4 4 4 8
```

C.4 Hydro2d

The hydro2d source code was modified so that the size of the surface simulated was reduced from 402x160 to 122x48. To accomplish this, the `MP` parameter in the source code was modified from 402 to 122 and the `NP` parameter was modified from 160 to 48.

C.5 Mgrid

The mgrid test data set was modified so that the size of the grid simulated was reduced from 64x64x64 to 16x16x16 and the number of iterations to simulate was reduced from 40 to 20. The following is a listing of the input file:

```
4
20
1
  NEGATIVE CHARGES AT
 ( 65, 48, 27) ( 14, 31, 30) ( 41, 51, 59) ( 31, 64, 56) ( 34, 39,  2)
 ( 28, 27, 27) ( 27, 16, 38) ( 20, 47, 38) ( 52, 17, 25) ( 40, 62, 53)
  POSITIVE CHARGES AT
 ( 60, 17, 56) ( 60, 21, 64) ( 53, 48, 28) ( 10, 18, 50) ( 19, 45, 55)
 ( 25, 49, 59) ( 22, 34, 60) ( 47, 25, 51) ( 41,  2,  7) ( 29, 34, 47)
```

C.6 Turb3d

The turb3d source code was modified so that the size of the cube simulated was reduced from 64x64x64 to 16x16x16. To accomplish this, the `IX`, `IY`, and `IZ` parameters in the source code were reduced from 64 to 16.

C.7 Apsi

The apsi test data set was modified so that the size of the grid simulated was reduced from 128x1x32 to 32x1x8 and the number of timesteps to simulate was reduced from 720 to 100. The following is a listing of the input file:

```
-----------------------------------------------!                             00001000
  *** D A T A   F O R   M E S O - R U N ***    !                             00002000
-----------------------------------------------!                             00003000
              INTEGER CONSTANTS                !                             00004000
-----------------------------------------------!                             00005000
     L E G E N T           !  V A L U E        !                             00006000
-----------------------------------------------!                             00007000
GRID POINTS IN x DIRECTION   !        16       !                             00008000
GRID POINTS IN y DIRECTION   !         1       !                             00009000
GRID POINTS IN z DIRECTION   !         8       !                             00010000
MOMENTUM SMOOTHING RATIO     !         2       !                             00011000
SPECTRAL FILTERING (0=n,1=y) !         1       !                             00012000
NUMBER OF TIME STEPS         !       100       !                             00013000
EXCHANGE LEAP TO EULER FREQ  !        20       !                             00014000
VERTICAL SCALE (0=reg,1=map) !         1       !                             00015000
BIAS FOR HEATING FUNCTION    !         1       !                             00016000
HORIZONTAL FILTERING STEP    !      2000       !                             00017000
BATCH MODE RUN   (0=y,1=n)   !         0       !                             00018000
VERTICAL SCHEME (0=Pade 1=CN)!         0       !                             00019000
Z-SMOOTHING (1=YES,0=NO)     !         0       !                             00020000
RERUN ->SAVED DATA (0=n,1=y) !         0       !                             00021000
SALVAGE DATA PARAMETER STEP  !       100       !                             00022000
```

```
SAVE GRAPH DATA (0=N, 1=Y)   !        100       !                         00023000
SAVE GRAPH DATA EVERY OTHER  !        720       !                         00024000
NUMBER OF C SOURCES/SINKS    !          1       !                         00025000
NUMBER OF INITIAL CONDITIONS !          0       !                         00026000
SOURCE #1 (1=Gauss, 0=Delta) !          1       !                         00027000
I.C.   #1 (1=Gauss, 0=Delta) !          1       !                         00028000
SOURCE #2 (1=Gauss, 0=Delta) !          1       !                         00029000
I.C.   #2 (1=Gauss, 0=Delta) !          0       !                         00030000
x-LOCATION OF SOURCE 1 (I)   !          2       !                         00031000
x-LOCATION OF IN CON 1 (I)   !          1       !                         00032000
x-LOCATION OF SOURCE 2 (I)   !          6       !                         00033000
x-LOCATION OF IN CON 2 (I)   !          3       !                         00034000
y-LOCATION OF SOURCE 1 (J)   !          1       !                         00035000
y-LOCATION OF IN CON 1 (J)   !          2       !                         00036000
y-LOCATION OF SOURCE 2 (J)   !          1       !                         00037000
y-LOCATION OF IN CON 2 (J)   !          1       !                         00038000
z-LOCATION OF SOURCE 1 (K)   !          1       !                         00039000
z-LOCATION OF IN CON 1 (K)   !          1       !                         00040000
z-LOCATION OF SOURCE 2 (K)   !          1       !                         00041000
z-LOCATION OF IN CON 2 (K)   !          3       !                         00042000
STRENGTH  OF SOURCE 1        !          1       !                         00043000
STRENGTH  OF IN CON 1        !       5000       !                         00044000
STRENGTH  OF SOURCE 2        !          1       !                         00045000
STRENGTH  OF IN CON 2        !       2000       !                         00046000
x-SPREAD  OF SOURCE 1  (I-GP)!          1       !                         00047000
x-SPREAD  OF IN CON 1  (I-GP)!          2       !                         00048000
x-SPREAD  OF SOURCE 2  (I-GP)!          1       !                         00049000
x-SPREAD  OF IN CON 2  (I-GP)!          0       !                         00050000
y-SPREAD  OF SOURCE 1  (J-GP)!          1       !                         00051000
y-SPREAD  OF IN CON 1  (J-GP)!          1       !                         00052000
y-SPREAD  OF SOURCE 2  (J-GP)!          1       !                         00053000
y-SPREAD  OF IN CON 2  (J-GP)!          1       !                         00054000
z-SPREAD  OF SOURCE 1  (K-GP)!          1       !                         00055000
z-SPREAD  OF IN CON 1  (K-GP)!          4       !                         00056000
z-SPREAD  OF SOURCE 2  (K-GP)!          1       !                         00057000
z-SPREAD  OF IN CON 2  (K-GP)!          0       !                         00058000
VARIANCE  Sxy OF SOURCE 1    !         20       !                         00059000
VARIANCE  Sxy OF IN CON 1    !         20       !                         00060000
VARIANCE  Sxy OF SOURCE 2    !         20       !                         00061000
VARIANCE  Sxy OF IN CON 2    !          0       !                         00062000
STORE AS A FIRST  z-LEVEL    !          2       !                         00063000
STORE AS A SECOND z-LEVEL    !          3       !                         00064000
STORE AS A THIRD  z-LEVEL    !          5       !                         00065000
STORE AS A FOURTH z-LEVEL    !         30       !                         00066000
HOMOGENIOUS y (0=2-D, 1=3-D) !          1       !                         00067000
MEAN FIELD RECALCULATE STEPS !      10000       !                         00068000
UG RATE OF CHANGE (cm/hour)  !         10       !                         00069000
VG RATE OF CHANGE (cm/hour)  !         10       !                         00070000
T0 RATE OF CHANGE (K/10hour) !         10       !                         00071000
TN RATE OF CHANGE (K/10hour) !         10       !                         00072000
CPU INFORMATION FREQUENCY    !       1000       !                         00073000
------------------------------------------------!                         00074000
             REAL CONSTANTS                     !                         00075000
------------------------------------------------!                         00076000
```

```
     L E G E N T            !  V A L U E     !                       00077000
-------------------------------------------------!                      00078000
STARTING TIME (sec)         !        0.0     !                          00079000
DELTA t  (TIME-STEP) (sec)  !      120.0     !                          00080000
STARTING x    (m)           !        0.0     !                          00081000
DELTA x  (x-STEP)  (m)      !     8000.0     !                          00082000
DIFFUSSIVITY  Kx (m*m/sec)  !     7000.0     !                          00083000
STARTING y    (m)           !        0.0     !                          00084000
DELTA y  (y-STEP)  (m)      !     5000.0     !                          00085000
DIFFUSSIVITY  Ky (m*m/sec)  !     1000.0     !                          00086000
STARTING z    (m)           !        0.0     !                          00087000
DELTA z  (z-STEP)  (m)      !      100.0     !                          00088000
MIN DIFFUSSIVITY  Kz(m*m/sec)!       0.1     !                          00089000
SPREAD OF STRECHED VARIABLE a!       0.3     !                          00090000
AMPLITUDE OF STRECHED VBL  b !     120.0     !                          00091000
HEIGHT H OF OBSERVED DATA (m)!    1000.0     !                          00092000
Ug(H) GEOSTROPHIC WIND (m/s) !       7.0     !                          00093000
Vg(H) GEOSTROPHIC WIND (m/s) !       0.0     !                          00094000
POTENTIAL TEMPERATURE AT z=0 !     290.0     !                          00095000
POTENTIAL TEMPERATURE AT z=H !     295.0     !                          00096000
STEP FOR WIND ITERATION delt !      60.0     !                          00097000
TOLLERANCE FOR ITERATION     !    2000.00002 !                          00098000
UPWINDING FOR CRANK-NICOLSON !       0.5     !                          00099000
x-SYMMETRY OF LAKE          !        2.8     !                          00100000
x-SPREAD OF LAKE            !    45000.0     !                          00101000
y-SYMMETRY OF LAKE          !        2.0     !                          00102000
y-SPREAD OF LAKE            !    20000.0     !                          00103000
URBAN ROUGHNESS LENGTH      !        0.02    !                          00104000
HEAT AMPLITUDE IN DEGREES (K)!      11.00    !                          00105000
HEAT LAG FACTOR IN SECONDS  !    50400.00    !                          00106000
CENTRAL LATITUDE IN DEGREES !       40.00    !                          00107000
IMPLICIT HOR SMOOTHING LAMDA !       0.5     !                          00108000
X-SYMMETRY OF SPECTRAL FILTER!       2.0     !                          00109000
Y-SYMMETRY OF SPECTRAL FILTER!       2.0     !                          00110000
X-SPREAD OF SPECTRAL FILTER !    85000.0     !                          00111000
Y-SPREAD OF SPECTRAL FILTER !    50000.0     !                          00112000
FILTER BASE NUMBER          !         .1     !                          00113000
SURFACE LAYER FRACTION OF BL !        .05    !                          00114000
INITIAL MIXED LAYER HEGHT (m)!     200.0     !                          00115000
INITIAL ROUGHNES OVER WATER  !        .01    !                          00116000
ROUGHNES LENGTH OVER LAND (m)!        .1     !                          00117000
-------------------------------------------------                       00118000
```

C.8 Wave5

The wave5 test data set was modified so that the size of the surface simulated was reduced from 1250x40 to 125x4 and the particle distribution from 5000x100 to 500x20. The following is a listing of the input file:

```
number of steps
4
particle distribution
```

```
500 20
nplots
2
grid size
125 4
xmax ymax
62.5 2.0
```

APPENDIX D

Inlined Call Sites

Tables D.1– D.7 lists the call sites that were selected for inlining for each of the benchmarks.

Called Function	Call Site Module	Call Site Line Number	Call Site Frequency	Called Function Size (bytes)
getbyte	compress95.c	477	1.58%	80
putbyte	compress95.c	739	1.57%	48
putbyte	compress95.c	621	0.97%	48
getcode	compress95.c	703	0.88%	440
output	compress95.c	502	0.87%	540
readbytes	compress95.c	796	0.12%	132

Table D.1: Inlined call sites for the compress benchmark.

Called Function	Call Site Module	Call Site Line Number	Call Site Frequency	Called Function Size (bytes)
match2	g2shp.c	2331	0.32%	448
mrglist	g25.c	182	0.22%	576
match	g2shp.c	2328	0.22%	448
markspot	g25.c	249	0.15%	604
dellist	g2shp.c	2334	0.11%	536

Table D.2: Inlined call sites for the go benchmark.

Called Function	Call Site Module	Call Site Line Number	Call Site Frequency	Called Function Size (bytes)
emit_bits	jchuff.c	390	0.24%	312
emit_bits	jchuff.c	395	0.24%	312

Table D.3: Inlined call sites for the ijpeg benchmark.

Called Function	Call Site Module	Call Site Line Number	Call Site Frequency	Called Function Size (bytes)
xlarg	xlsubr.c	50	0.18%	80
xlarg	xlsubr.c	75	0.27%	80
xleval	xlsubr.c	78	0.27%	232
consd	xlsym.c	68	0.14%	48
cons	xlsym.c	72	0.14%	56
xlxgetvalue	xlsym.c	79	0.75%	164
xlobgetvalue	xlsym.c	91	0.75%	376
livecar	xldmem.c	294	0.87%	176
livecdr	xldmem.c	303	0.27%	132
livecdr	xldmem.c	324	0.59%	132
mark	xldmem.c	355	0.28%	344
xlgetvalue	xleval.c	34	0.75%	68
xlevlist	xleval.c	105	0.23%	196
consa	xleval.c	190	0.48%	48
iskeyword	xleval.c	256	0.14%	88
xlbind	xleval.c	259	0.14%	72
xlygetvalue	xlobj.c	90	0.75%	104
xlygetvalue	xlobj.c	91	0.75%	104

Table D.4: Inlined call sites for the xlisp benchmark.

Called Function	Call Site Module	Call Site Line Number	Call Site Frequency	Called Function Size (bytes)
display_trace	dpath.c	860	0.39%	752
check_scoreboard	simtime.c	75	0.39%	444
Data_path	go.c	120	0.39%	800
ckbrkpts	dpath.c	86	0.39%	284
getmemptr	dpath.c	96	0.39%	120
test_issue	dpath.c	105	0.39%	264
Statistics	dpath.c	105	0.39%	316
do_issue	dpath.c	105	0.39%	64
uext	dpath.c	105	0.39%	124
execute	dpath.c	105	0.39%	4404
killtime	dpath.c	105	0.39%	248
Pc	dpath.c	105	0.39%	628

Table D.5: Inlined call sites for the m88ksim benchmark.

Called Function	Call Site Module	Call Site Line Number	Call Site Frequency	Called Function Size (bytes)
str_numset	eval.c	1745	1.31%	80
str_numset	str.c	281	0.52%	80
str_free	cmd.c	676	0.26%	272
str_sset	eval.c	459	0.26%	524
str_mortal	eval.c	224	0.26%	228
str_inc	eval.c	226	0.26%	684
str_new	str.c	1262	0.26%	136
str_sset	str.c	1264	0.26%	524

Table D.6: Inlined call sites for the perl benchmark.

Called Function	Call Site Module	Call Site Line Number	Call Site Frequency	Called Function Size (bytes)
Chunk_ChkGetChunk	mem10.c	590	1.01%	324
Chunk_ChkGetChunk	mem10.c	749	0.44%	324
TmFetchCoreDb	tm.c	148	0.31%	532
Hm_FetchDbObject	tm.c	150	0.31%	180
TmGetObject	oa0.c	432	0.31%	208
Mem_GetAddr	hm.c	749	0.31%	200
TmFetchCoreDb	tm.c	133	0.18%	532
Mem_GetWord	tm.c	134	0.18%	228
Chunk_ChkGetChunk	mem10.c	416	0.15%	324
Mem_GetStackPtr	mem10.c	417	0.15%	160
Mem_GetWord	env1.c	965	0.14%	228
memcpy	core01.c	974	0.13%	196
Ut_StackTrack	core01.c	1014	0.13%	152

Table D.7: Inlined call sites for the vortex benchmark.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, 1986.

[2] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *10th Annual ACM Symposium on Principles of Programming Languages*, pp. 177–189, 1983.

[3] M. G. Butler, *Aggressive Execution Engines for Surpassing Single Basic Block Execution*, PhD thesis, University of Michigan, 1993.

[4] P. Chang, *Compiler Support for Multiple-Instruction-Issue Architectures*, PhD thesis, University of Illinois at Urbana-Champaign, 1991.

[5] P.-Y. Chang, E. Hao, and Y. N. Patt, "Target prediction for indirect jumps," To appear in the 24th Annual International Symposium on Computer Architecture.

[6] P.-Y. Chang, E. Hao, and Y. N. Patt, "Alternative implementations of hybrid branch predictors," in *Proceedings of the 28th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 252–257, 1995.

[7] P.-Y. Chang, E. Hao, Y. N. Patt, and P. P. Chang, "Using predicated execution to improve the performance of a dynamically scheduled machine with speculative execution," in *Proceedings of the IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '95*, pp. 99–108, 1995.

[8] P.-Y. Chang, E. Hao, T.-Y. Yeh, and Y. N. Patt, "Branch classification: A new mechanism for improving branch predictor performance," in *Proceedings of the 27th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 22–31, 1994.

[9] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "IMPACT: An architectural framework for multiple-instruction-issue processors," in *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pp. 266–275, 1991.

[10] T. M. Conte, K. N. Menezes, P. M. Mills, and B. Patel, "Optimization of instruction fetch mechanisms for high issue rates," in *Proceedings of the 22st Annual International Symposium on Computer Architecture*, pp. 333–344, 1995.

[11] S. Dutta and M. Franklin, "Control flow prediction with tree-like subgraphs for superscalar processors," in *Proceedings of the 28th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 258–263, 1995.

[12] K. Ebcioğlu, "Some design ideas for a VLIW architecture for sequential natured software," *Parallel Processing (Proceedings of IFIP WG 10.3 Working Conference on Parallel Processing*, pp. 3–21, April 1988.

[13] J. A. Fisher, "2^n-way jump microinstruction hardware and an effective instruction binding method," in *Proceedings of the 13th Annual Microprogramming Workshop*, pp. 64–75, 1980.

[14] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Transactions on Computers*, vol. C-30, no. 7, pp. 478–490, July 1981.

[15] M. Franklin and G. S. Sohi, "The expandable split window paradigm for exploiting fine-grain parallelism," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 58–67, 1992.

[16] L. Gwennap, "PA-8000 combines complexity and speed," *Microprocessor Report*, vol. 8, no. 15, , November 1994.

[17] L. Gwennap, "Intel's P6 uses decoupled superscalar design," *Microprocessor Report*, vol. 9, , February 1995.

[18] L. Gwennap, "Digital 21264 sets new standard," *Microprocessor Report*, vol. 10, no. 14, , October 1996.

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

[20] P. Hsu and E. Davidson, "Highly concurrent scalar processing," in *Proceedings of the 13th Annual International Symposium on Computer Architecture*, 1986.

[21] W. W. Hwu and P. P. Chang, "Inline function expansion for compiling c programs," in *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation*, 1989.

[22] W. 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*, vol. 7, no. 9-50, , 1993.

[23] W. W. Hwu and Y. N. Patt, "Checkpoint repair for high-performance out-of-order execution machines," *IEEE Transactions on Computers*, vol. C-36, no. 12, , December 1987.

[24] *Intel Reference C Compiler User's Guide for UNIX Systems*, Intel Corporation, 1993.

[25] S. Jourdan, T. Hsing, J. Stark, and Y. N. Patt, "The effects of mispredicted-path execution on branch prediction structures," in *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, pp. 58–67, 1996.

[26] K. Karplus and A. Nicolau, "Efficient hardware for multi-way jumps and prefetches," in *Proceedings of the 18th Annual Microprogramming Workshop*, pp. 11–18, 1985.

[27] J. K. F. Lee and A. J. Smith, "branch prediction strategies and branch target buffer design," *IEEE Computer*, pp. 6–22, January 1984.

[28] S. A. Mahlke, R. E. Hank, R. A. Bringmann, J. C. Gyllenhaal, D. M. Gallagher, and W. W. Hwu, "Characterizing the impact of predicated execution on branch prediction," in *Proceedings of the 27th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 217–227, 1994.

[29] 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 *Proceedings of the 25th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 45–54, 1992.

[30] S. McFarling, "Combining branch predictors," Technical Report TN-36, Digital Western Research Laboratory, June 1993.

[31] S. Melvin, *Performance Enhancement Through Dynamic Scheduling and Lareg Execution Atomic Units in Single Instruction Stream Processors*, PhD thesis, University of California, Berkeley, May 1991.

[32] S. Melvin and Y. Patt, "Enhancing instruction scheduling with a block-structured ISA," *International Journal on Parallel Processing*, vol. 23, no. 3, pp. 221–243, 1995.

[33] S. Melvin and Y. N. Patt, "Exploiting fine-grained parallelism through a combination of hardware and software techniques," in *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pp. 287–297, 1991.

[34] S. W. Melvin and Y. N. Patt, "Performance benefits of large execution atomic units in dynamically scheduled machines," in *Proceedings of Supercomputing '89*, pp. 427–432, 1989.

[35] S. W. Melvin, M. C. Shebanow, and Y. N. Patt, "Hardware support for large atomic units in dynamically scheduled machines," in *Proceedings of the 21st Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 60–63, 1988.

[36] S.-M. Moon and K. Ebcioğlu, "An efficient resource-constrained global scheduling technique for superscalar and VLIW processors," in *Proceedings of the 25th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 55–71, 1992.

[37] *MC88110 Second Generation RISC Microprocessor User's Manual*, Motorola, 1991.

[38] S.-T. Pan, K. So, and J. T. Rahmeh, "Improving the accuracy of dynamic branch prediction using branch correlation," in *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 76–84, 1992.

[39] S. J. Patel, S. W. Kim, D. H. Friendly, and Y. N. Patt, "Enhancing the trace cache fetch mechanism," To appear as a University of Michigan technical report, 1997.

[40] Y. Patt, W. Hwu, and M. Shebanow, "HPS, a new microarchitecture: Rationale and introduction," in *Proceedings of the 18th Annual Microprogramming Workshop*, pp. 103–107, 1985.

[41] Y. N. Patt, S. W. Melvin, W. Hwu, and M. C. Shebanow, "Critical issues regarding HPS, a high performance microarchitecture," in *Proceedings of the 18th Annual Microprogramming Workshop*, pp. 109–116, 1985.

[42] D. N. Pnevmatikatos and G. S. Sohi, "Guarded execution and dynamic branch prediction in dynamic ILP processors," in *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pp. 120–129, 1994.

[43] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle, "The Cydra 5 departmental supercomputer," *IEEE Computer*, vol. 22, pp. 12–35, January 1989.

[44] E. Rotenberg, S. Bennett, and J. E. Smith, "Trace cache: A low latency approach to high bandwidth instruction fetching," Technical Report 1310, University of Wisconsin - Madison, April 1996.

[45] A. Seznec, S. Jourdan, P. Sainrat, and P. Michaud, "Multiple-block ahead branch predictors," in *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[46] J. E. Smith, "A study of branch prediction strategies," in *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pp. 135–148, 1981.

[47] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, "Multiscalar processors," in *Proceedings of the 22st Annual International Symposium on Computer Architecture*, 1995.

[48] E. Sprangle and Y. Patt, "Facilitating superscalar processing via a combined static/dynamic register renaming scheme," in *Proceedings of the 27th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 143–147, 1994.

[49] *Welcome to SPEC*, The Standard Performance Evaluation Corporation. http://www.specbench.org/.

[50] J. W. Stark and Y. N. Patt, "The effects of memory disambiguation on the performance of wide-issue, dynamically-scheduled microprocessors," To appear as a University of Michigan technical report, 1995.

[51] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of Research and Development*, vol. 11, pp. 25–33, January 1967.

[52] J. E. Wilson, S. Melvin, M. Shebanow, W. mei Hwu, and Y. N. Patt, "On tuning the microarchitecture of an HPS implementation of the VAX," in *Proceedings of the 20th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 162–167, 1987.

[53] T.-Y. Yeh, *Two-Level Adaptive Branch Prediction and Instruction Fetch Mechanisms for High Performance Superscalar Processors*, PhD thesis, University of Michigan, 1993.

[54] T.-Y. Yeh, D. Marr, and Y. N. Patt, "Increasing the instruction fetch rate via multiple branch prediction and branch address cache," in *Proceedings of the International Conference on Supercomputing*, pp. 67–76, 1993.

[55] T.-Y. Yeh and Y. N. Patt, "Two-level adaptive branch prediction," in *Proceedings of the 24th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 51–61, 1991.

[56] T.-Y. Yeh and Y. N. Patt, "Alternative implementations of two-level adaptive branch prediction," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 124–134, 1992.

[57] T.-Y. Yeh and Y. N. Patt, "A comparison of dynamic branch predictors that use two levels of branch history," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pp. 257–266, 1993.

[58] C. Young and M. D. Smith, "Improving the accuracy of static branch prediction using branch correlation," in *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 232–241, 1994.