BLOCK-AWARE INSTRUCTION SET ARCHITECTURE

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Ahmad Darweesh Zmily

June 2007

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Christoforos Kozyrakis) Principal Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Oyekunle Olukotun)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Mark Horowitz)

Approved for the University Committee on Graduate Studies.

Abstract

This dissertation examines the use of a block-aware instruction set architecture (BLISS) to address the front-end challenges of modern processors. The theme of BLISS is to allow software to assist the front-end hardware by providing architecture support for control-flow prediction and instruction delivery. BLISS defines basic block descriptors in addition to and separately from the actual instructions in each program. A descriptor describes the type of the control-flow operation that terminates the block, its potential target, and the number of instructions in the basic block. This information is sufficient for fast and accurate control-flow prediction without accessing or parsing the instruction stream. The architecture also provides a flexible mechanism for communicating compiler-generated hints at basic block granularity.

The BLISS ISA suggests a decoupled front-end organization that fetches the descriptors and the associated instructions in a decoupled manner. The front-end uses the information available in descriptors to improve control-flow accuracy, implement guided instruction prefetching, and reduce the energy used for control-flow prediction and instruction delivery. We demonstrate that the new architecture improves upon conventional superscalar designs by 20% in performance and 16% in energy. We also show that it outperforms hardware-only approach for decoupled front-ends by 13% and 7% for performance and energy respectively. These benefits are robust across a wide range of architectural parameters.

We also evaluate the use of BLISS for embedded processor designs. We develop a set of code size optimizations that utilize the ISA mechanism to provide code size reduction

of 40% while maintaining a 10% performance and 21% energy advantages. We also develop and evaluate a set of hardware and software techniques for low cost front-ends for embedded systems. The optimization techniques target the size and power consumption of instruction caches and predictor tables. We show that the decoupling features of BLISS and the ability to provide software hints allow for embedded designs that use minimally sized, power efficient caching and predictor structures, without sacrificing performance.

Acknowledgements

The work on this thesis has been an inspiring and enjoyable journey for me. During which, I have been accompanied and supported by many people. It is a pleasant that I have now the opportunity to express my gratitude for all of them.

First, I wish to express my sincere gratitude to my thesis adviser, Professor Christos Kozyrakis. I could not have imagined having a better adviser for my thesis. Without his guidance, patience, and constant support, I would never have achieved this. I would like also to thank my Ph.D. oral examiners and committee members Professor Kunle Olukotun, Professor Mark Horowitz, and Professor Krishna Saraswat.

Special thanks to my friends and to my colleagues at Intel for their friendship and support during my Ph.D. studies. I would like also to acknowledge Earl Kilian for his valuable input. This work was partially supported by the Stanford University, Office of Technology Licensing.

Finally, I am forever indebted to my father Darweesh and my mother Suzan for their constant support and endless love and care. My father, who passed away almost ten years ago, still has deep influence in every aspect of my life. I wish he were here with us celebrating this precious moment; he would be so happy and proud. No words can express my gratitude to my mother who always believed in me. Her persistence and determination inspire me and help me in surviving many challenges in my life. My thanks and love to her always. I am very fortunate to have a big, wonderful family. I would like to express my

deepest gratitude to my three lovely sisters Eman, Asma, and Amal and to my two brothers Misbah and Hammam for their constant support, love, and encouragement when it was most required.

I dedicate this thesis to the memory of my great father and to my lovely, wonderful mother.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Effective instruction delivery is vital for superscalar processors operating at high clock frequencies [79, 93]. The rate and accuracy at which instructions enter the processor pipeline set an upper limit to sustained performance. Consequently, modern processor designs place increased demands on the *front-end*, the engine responsible for control-flow prediction and instruction fetching. Conservative instruction delivery can severely limit the performance potential of the processor by unnecessarily gating instruction level parallelism. On the other hand, overly aggressive instruction delivery wastes energy on the execution of misspeculated instructions (over-speculation). Aggressive speculation can also reduce performance by frequently causing expensive pipeline flushes on mispredicted branches.

In addition to high application performance, energy efficiency, code size, power consumption, and die area are also critical design metrics. Energy efficiency is essential for both high-end and embedded processors. High energy consumption can severely limit the server scalability, its operational cost, and its reliability [28]. Energy consumption dictates if an embedded processor can be used in portable or deeply embedded systems for which battery size and lifetime are vital parameters. Code size determines the amount and cost of on-chip or off-chip memory necessary for program storage. For embedded applications,

instruction memory is often as expensive as the processor itself. Finally, power consumption and die area determine the cost to manufacture, package, and cool the chip. For most modern processors, the front-end design must strike a balance between multiple of these efficiency metrics. However, improving one metric often comes at the expense of another as they introduce conflicting tradeoffs.

In its effort to provide a balance between the efficiency metrics, the front-end must mitigate three basic detractors: instruction cache misses that cause long instruction delivery stalls; target and direction mispredictions for control-flow instructions that send erroneous instructions to the execution core and expose the pipeline depth; and multi-cycle instruction cache accesses in high-frequency designs that introduce additional uncertainty about the existence and direction of branches within the instruction stream. Conventional approaches to attack the front-end detractors typically rely on hardware-only techniques working below the ISA level.

The theme of this work is to allow software to assist the hardware in dealing with the front-end challenges by providing architecture support for control-flow prediction and instruction delivery. The new architecture provides efficient ISA-level support for front-end optimizations that target all of the processor efficiency metrics (performance, energy, power, code size, and die area). The architecture also provides a flexible communication mechanism that software can use to provide hardware with critical information about instruction fetching and control-flow.

Specifically, this dissertation examines the use of a block-aware instruction set architecture (BLISS) to address the front-end challenges. BLISS defines basic block descriptors in addition to and separately from the actual instructions in each program. A descriptor provides sufficient information for fast and accurate control-flow prediction without accessing or parsing the conventional instruction stream. It describes the type of the control-flow operation that terminates the basic block, its potential target, and the

number of instructions it contains. The architecturally visible basic block descriptors enable a wide-range of performance and energy optimizations and provide a flexible mechanism for communicating compiler-generated hints at the granularity of basic blocks without modifying the conventional instruction stream or affecting its instruction code footprint. BLISS allows for significant reorganization of the front-end for both high-end and embedded processors. Unlike techniques that rely on hardware-only or software-only features and typically improve one metric at the cost of another, BLISS strikes a balance between hardware and software features to provide benefits across all important metrics.

1.1 Research Contribution

The primary contributions of this dissertation are:

- We define the block-aware ISA that provides basic block descriptors in addition to and separately from the actual instructions in each program. The BLISS provides accurate information for control-flow prediction and instruction prefetching without fetching and parsing the actual instruction stream. BLISS also provides a versatile mechanism for conveying compiler-generated hints at basic block granularity without modifying the conventional instruction stream or affecting its instruction code footprint.

- We propose a decoupled front-end organization based on the BLISS ISA. The new front-end replaces the BTB with a descriptor's cache. It uses the information available in descriptors to improve control-flow accuracy, implement guided instruction prefetching, and reduce the energy used for control-flow prediction and instruction delivery. We demonstrate that the new architecture improves upon conventional superscalar designs by 20% in performance and 16% in energy. We also show that it

outperforms hardware-only approach for decoupled front-ends by 13% and 7% for performance and energy respectively. These benefits are robust across a wide range of architectural parameters.

- We evaluate the use of BLISS for embedded processor designs. We develop a set of code size optimizations that utilize the ISA mechanism to provide code size reduction of 40%. Unlike alternative proposals that tradeoff performance or energy consumption for code density, we show that BLISS-based embedded designs provide 10% performance and 21% energy advantages in addition to the improved code size.

- We develop and evaluate a set of hardware and software techniques for low cost front-ends for embedded systems. The optimization techniques target the size and power consumption of instruction caches and predictor tables. We show that the decoupling features of BLISS and the ability to provide software hints allow for embedded designs that use minimally sized, power efficient caching and predictor structures, without sacrificing performance.

The results in this thesis have been presented in several conference and journal publications [115, 116, 117, 118].

1.2 Organization of this Dissertation

The outline of the rest of this thesis is as follows.

Chapter 2 provides an overview of the front-end operation and discusses the major challenges to sustain high instruction bandwidth in an energy efficient manner. It also presents the motivation for this thesis by summarizing the potential gains in overall performance and energy consumption that can be achieved by improving the processor front-end.

In Chapter 3, we introduce the block-aware instruction set (BLISS) that provides additional information about the control-flow of the program at the granularity of basic blocks. We explain the descriptor format, provide an example, discuss the benefits of the ISA, and explain the software hints mechanism and their possible usages. The chapter also provides an ISA-level characterization of BLISS using applications from the SPEC benchmark suite.

Chapter 4 describes the decoupled front-end that exploits the basic block information available in the BLISS ISA. The chapter describes the architecture and operation of the BLISS-based front-end. It also describes and compares to BLISS a similar decoupled front-end design that forms extended basic blocks using hardware-only techniques.

A comprehensive evaluation of the performance advantage of BLISS for high-end superscalar processors is presented in Chapter 5. We compare the BLISS design to both a conventional front-end processor and a decoupled front-end processor with no software support. We also present a detailed performance analysis when key architectural parameters of the front-end are varied.

Chapter 6 focuses on the energy efficiency of the BLISS-based front-end. It explains how the BLISS front-end reduces the energy wasted by mispredicted instructions and reduces the energy used for control-flow prediction and instruction delivery. Similar to performance, we also perform a comprehensive evaluation for the BLISS energy efficiency.

In Chapter 7, we evaluate the use of BLISS for embedded designs. We explain the code size optimizations enabled with the BLISS ISA and analyze their effect on performance and energy efficiency. We also compare BLISS with the code optimizations to alternative approaches that build similar optimizations on top of conventional ISAs.

In Chapter 8, we show that the flexible semantics of BLISS allow us to implement a wide range of software and hardware optimizations without modifying the software model. We explain each optimization and evaluate its effect on performance, power, and energy efficiency. We also evaluate combinations of these optimizations for embedded processors.

Finally, the dissertation concludes in Chapter 9 which provides a summary and highlights future work.

Chapter 2

Background and Motivation

The front-end portion of the processor is responsible for predicting the next set of instructions, fetching them from memory, and delivering them to the back-end for execution. Since the back-end cannot execute instructions faster than they are being delivered, the instruction bandwidth sustained by the front-end sets an upper limit to the overall processor performance.

In this chapter, we provide an overview of the front-end operation and discuss the major challenges to sustain high instruction bandwidth in an energy efficient manner. Section 2.1 provides an overview of the front-end in a modern processor. In Section 2.2, we discuss the key front-end detractors and their impact on performance. In Section 2.3, we analyze the impact of the front-end challenges on the processor energy consumption. Section 2.4 summarizes the related work in front-end architecture and design. Section 2.5 provides the motivation for this thesis by summarizing the potential gains in overall performance and energy consumption that can be achieved by improving the processor front-end.

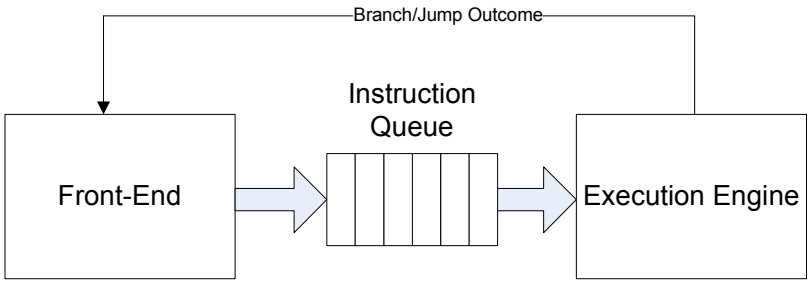


Figure 2.1: A typical pipelined processor: the front-end fetches instructions from memory and the execution engine executes them.

2.1 Front-End Overview

A processor mainly consists of a front-end and an execution engine. Figure 2.1 shows a typical pipelined processor which consists of a front-end that fetches instructions and an execution core that consumes the instructions and provides feedback to the front-end. The two components are separated by an instruction queue for decoupling purposes. In order to maintain high bandwidth, the front-end must be able to predict the execution order of instructions in the program. To accomplish this task, the front-end speculatively follows the execution path independently of the execution core and places the instructions in the queue. The execution engine reads the instructions from the buffer, generates the execution results, and provides feedback to the front-end regarding the actual outcome of the branch instructions. In case of a mismatch between the speculative and the actual execution paths, the misspeculated instructions are flushed from the processor pipeline and the front-end starts fetching at the correct address of the first mispredicted branch.

Instruction fetch is a critical component for the processor with respect to performance, energy consumption, power, and complexity [93]. The front-end sets an upper limit on performance as the execution core cannot execute instructions faster than they are being delivered. Conservative instruction delivery can severely limit the performance potential

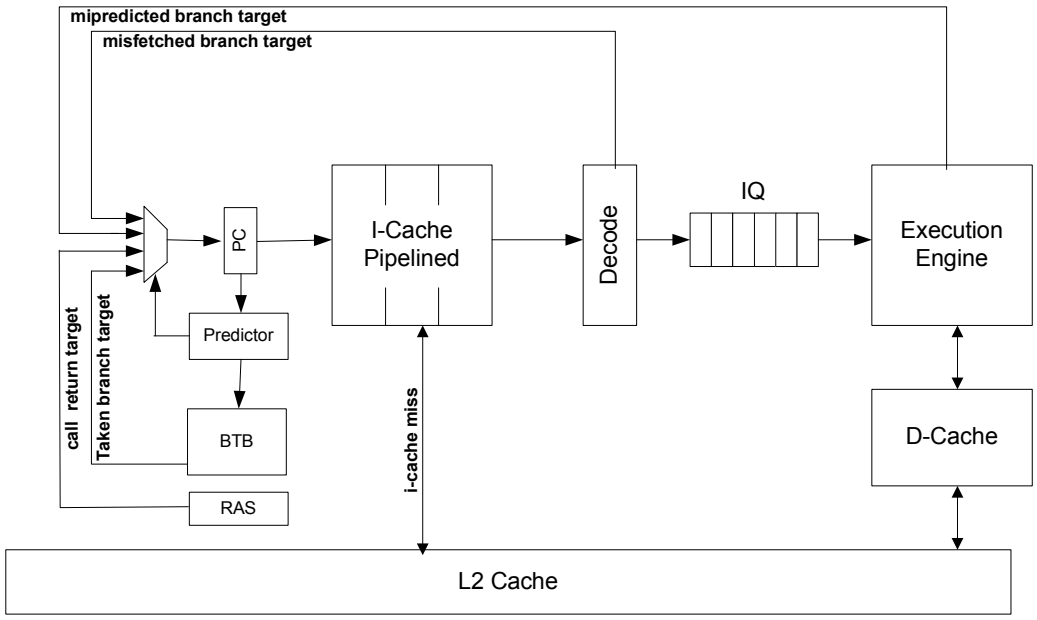


Figure 2.2: A superscalar front-end processor architecture.

of the processor. On the other hand, overly aggressive instruction delivery can reduce performance because of the overhead of frequent misspeculations. The front-end also influences the energy consumption of the processor as it determines how often the processor is executing useful instructions, mispredicted instructions, or no instructions at all. The processor wastes energy when it is executing mispredicted instructions or when the pipeline is empty.

Figure 2.2 presents the basic components in a typical front-end for a superscalar processor. The program counter (**PC**) points to the address of a set of sequential instructions that need to be fetched from the main memory. The instruction cache is a smaller, faster memory that buffers frequently used instructions to reduce the average time to access the main memory. The instruction cache is able to provide a set of sequential instructions each cycle. The branch predictor is a tagless array of counters that predict if a sequential set of instructions contains a taken branch or jump instruction. The branch target buffer (**BTB**)

is a tagged array that predicts the target of a taken control-flow instruction before it is executed. The return address stack (**RAS**) is a small memory structure that predicts the return address for functions by matching returns with corresponding calls. The branch predictor, BTB, and RAS allow the front-end of the processor to fetch instructions without waiting for the instructions to be decoded or the branches to be resolved.

A conventional front-end operates in the following manner. On every cycle, the instruction cache, predictor, and BTB are accessed using the PC. When the instructions are not available in the instruction cache, the cache generates a miss and the front-end stalls until the missing instructions are retrieved from lower memory hierarchy (L2-cache and main memory). On a cache hit, when the instructions are available in the cache, they are fetched and pushed into the instruction queue. The predictor and BTB outcomes are used for determining the next PC. If the predictor predicts a taken branch within the current set of fetched instructions, the next PC would be the outcome of the BTB or RAS. Otherwise, the next sequential address is used for accessing the instruction cache in the following cycle. The branch prediction is verified once the instructions are decoded (existence of control-flow instruction verified) and after the branches are executed (direction and target of control-flow instruction verified). In a case of a mismatch, the mispredicted instructions are flushed from the pipeline and fetching starts at the correct address.

With the increasing frequency gap between the processor and main memory, the performance of the instruction cache is critical to sustain high instruction fetch bandwidth. As a result, an increasing amount of resources are used to improve instruction cache performance. It is typical for advanced microprocessors today to use an instruction cache size of 16 to 64 KBytes with two- to three-cycle access time [93]. Today's modern processors are also designed with deep pipelines as a result of increased frequency. Although deep pipelines improve overall performance by allowing aggressive clock rates, they also lead to increasing misprediction penalties. Therefore, accurate branch prediction is crucial to

sustain high instruction fetch bandwidth. Modern processors typically use advanced hybrid branch predictors with a 512 to 4K-entries BTB to better predict the correct execution path.

The challenge with the front-end is to achieve high instruction fetch bandwidth in an energy-effective manner. Increasing the front-end resources and using advanced speculation techniques usually come at a cost of increased power, energy consumption, and complexity. In addition, the front-end itself consumes a significant percentage of the processor energy as it contains large memory arrays that are accessed nearly every cycle. On average, 13% of the total energy is consumed in the front-end itself alone for a 4-way superscalar processor [115].

2.2 Front-End Performance Detractors

In its effort to provide high instruction fetch bandwidth, the front-end engine must handle several detractors: instruction cache misses that cause instruction delivery stalls; multi-cycle instruction cache accesses that cause uncertainty about the existence and direction of branches within the instruction stream; and target and direction mispredictions for branches that send erroneous instructions to the execution core. These problems have a negative impact on both performance and energy consumption. For a 4-way superscalar processor running the SPEC benchmark, the cost is up to 33% in performance and 22% in energy consumption. In this section, we will look at each of these problems and its impact on performance in more details. The next section focuses on their energy impact.

2.2.1 Instruction Cache Detractors

The instruction cache introduces two challenges to instruction delivery: cache misses that interrupt instruction delivery and multi-cycle cache hits that introduce uncertainty about the existence of branches in the instruction stream.

Instruction Cache Misses

An instruction cache miss refers to a failed attempt to read the needed instructions from the instruction cache, which results in a long latency access to lower levels of the memory hierarchy. Instruction cache misses cause the front-end to stall until the missing instructions are available, leading to significant performance loss and energy waste due to leakage.

The design of the cache largely affects the number of misses. For example, larger cache size or higher cache associativity results in a reduction in the number of capacity and conflict misses. Nevertheless, this is not always desirable as it increases the cache access time. Fast cache access is important because it often determines the processor clock rate and the number of pipeline stages necessary for instruction access. Another effective technique to reduce the number of cache misses is to prefetch the instructions and bring them into the cache before they are even needed. The scheme used to predict the prefetch address is usually a challenge. The commonly used scheme is a sequential or stream-based prefetcher where one or more sequential cache lines that follow the current fetched line are prefetched [103, 77]. Ideally, we would like to prefetch instructions on the path of execution regardless of the code layout and the existence of taken branches.

Figure 2.3 quantifies the performance penalty due to instruction cache misses for a typical 4-way superscalar processor with a 32 KByte, 4-way set-associative instruction cache running the SPEC CPU benchmarks. The results are compared to similarly configured processor with a perfect instruction cache that does not suffer from any misses. The figure shows the loss for the benchmark with the maximum loss from the integer (INT) applications and the benchmark with the maximum loss from the floating-point (FP) applications (`vortex` and `apsi`). It also reports the average loss for INT applications and FP applications. Instruction cache misses alone cost 13% in performance for INT applications because they tend to have large instruction footprints and somewhat irregular code access patterns.

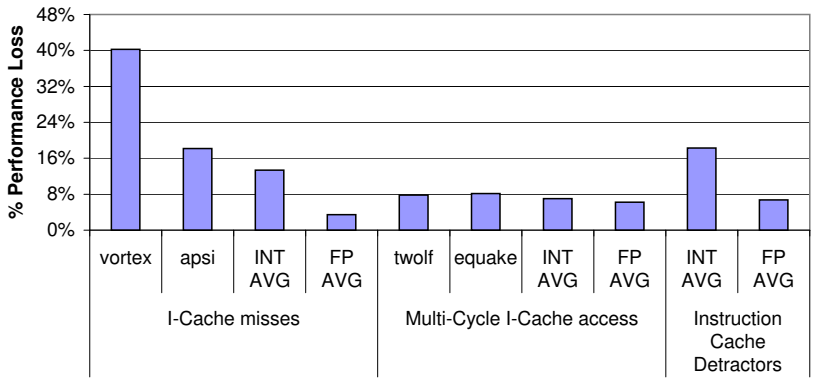


Figure 2.3: The percentage of performance loss for a 4-way superscalar processor running SPEC CPU benchmarks due to instruction cache misses and access latency. The instruction cache is 32 KBytes, 4-way set-associative with 2-cycle access time. (See 5.1 for methodology and configuration.)

On the other hand, the average loss for FP applications is 3% as these benchmarks are dominated by tight code loops with high temporal locality. Losses will be even more dramatic if we use a smaller cache size or a lower associativity because of the additional misses.

Multi-Cycle Instruction Cache Access

As processor frequencies get higher and the code footprint of important applications gets larger, it becomes difficult to access a large enough instruction cache within a single cycle [1]. It is common for modern processors to have instruction caches that need two to three cycles to access. This implies that the front-end cannot detect if there is any branch or jump in the current set of instructions to determine the program counter (PC) to use in the following cycle. To avoid a reduction in effective instruction bandwidth, modern superscalar processors use predictors and branch-target-buffers (BTB) to determine control-flow in a single cycle without knowing the exact control-flow instructions being fetched by pending instruction cache accesses [71]. Once the cache access completes in a later pipeline stage, the fetch unit verifies if the prediction it made earlier is appropriate given the type of the identified control-flow instruction.

The problem with this approach is that the prediction in the first pipeline stage lacks crucial information. First, it is not certain if the PC used to access the predictors points at or close enough to a control-flow instruction. As a result, the predictor is often accessed and updated with PCs that do not correspond to control-flow operations, which leads to reduced prediction accuracy due to interference. Second, until the instruction cache access completes, we cannot use the type of the control-flow operation in order to select between the outputs of different predictors in an accurate manner (e.g. BTB vs. RAS). As a result, even when accessing predictors in parallel with the instruction cache, the cache access latency must be relatively short in order to maintain high prediction accuracy.

Ideally, the fetch unit would first identify the control-flow operation that potentially terminates the current block and then predict the next basic block in execution order. In other words, the fetch unit would wait for the instruction cache access to complete and then use parsing and alignment logic to detect the first control-flow operation within the group of fetched instructions (if any). Given the type of the control-flow instruction and the outputs of various prediction structures (predictors, BTB, RAS), the fetch unit would make a very accurate prediction about the program counter (PC) to use in the following cycle. In cases where multiple branches exist within the set of fetched instructions, the front-end ideally needs to find the first taken branch ignoring any non-taken branches in front of the taken branch.

Figure 2.3 also reports the performance penalty due to instruction cache multi-cycle accesses for a 4-way superscalar processor with a 32 KByte, 4-way set-associative, 2-cycle access instruction cache and a hybrid predictor (4K-counter selector, 4K-counter Gshare, and 1K-entry L1, 1K-counter L2 PAg) running the SPEC CPU benchmarks. The results are compared to a similarly configured processor with a single-cycle access instruction cache. Multi-cycle instruction cache accesses cost 7% and 5% in performance for INT and FP applications respectively. For processors with higher instruction cache latency or with smaller predictor tables, the penalty would be even higher.

2.2.2 Branch Prediction Detractors

Independently of the instruction cache, branch prediction introduces two important detractors: branch direction mispredictions and branch target mispredictions.

Branch Direction Mispredictions

The direction predictor notifies the front-end if there is a control-flow change in the program or not. If not, the front-end will continue fetching using the next sequential address. As explained in Section 2.2.1, the direction predictor is accessed simultaneously with the instruction cache. Therefore, it lacks information about the instructions that are being fetched. This means that the predictor needs to be trained for instructions (program counter values) that do not correspond to control-flow operations and for unconditional control-flow instructions. This leads to loss in prediction accuracy due to slower training and more interference on the limited entries in the table. The predictor is also accessed using the PC which points to the block address of the current fetched instructions instead of the branch address itself. This negatively affects the predictor accuracy. For example, if a biased non-taken branch in the middle of the fetched block changes behavior, the predictor entry that is used for the control-flow operation that terminates the original block is now used for the biased branch. This could be avoided if the exact branch address is used to access the predictor instead of the current PC address.

Figure 2.4 quantifies the performance penalty due to branch direction mispredictions for a 4-way superscalar processor running the SPEC CPU benchmarks. The branch predictor is configured as a hybrid predictor with a two-level, correlating predictor with global history (Gshare) and a two-level, correlating predictor with per-address history (PAg). The results are compared to a similarly configured processor with a perfect direction predictor that always predicts the correct branch direction. Branch direction mispredictions alone

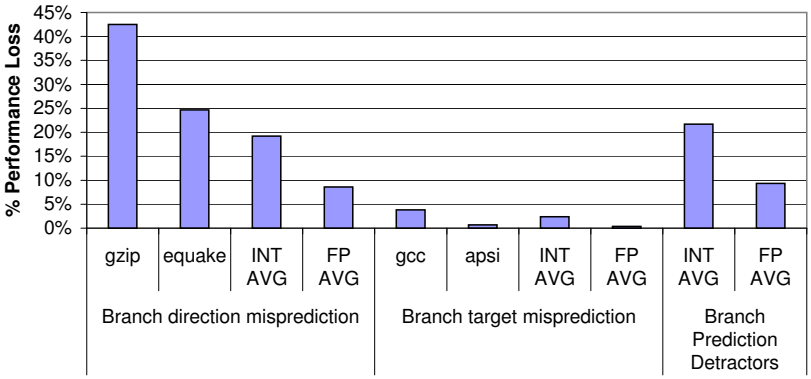


Figure 2.4: The percentage of performance loss for a 4-way superscalar processor running SPEC CPU benchmarks due to branch direction and target mispredictions. The BTB is configured 4-way with 1K-entries. The hybrid predictor has a 4K-counter selector, 4K-counter Gshare, and 1K-entry L1, 1K-counter L2 PAg.

cost 19% in performance for INT applications and 9% for FP applications. FP applications are affected less as they have larger basic blocks than INT applications (fewer predictions/executed instructions) and a smaller number of static branches (fewer addresses used to train/access predictor).

Branch Target Misprediction

The branch target buffer (BTB) stores the target addresses for previously executed taken branches and jumps. Once the direction predictor guesses a change in control-flow, the BTB provides the target address of the control-flow operation. The front-end uses this address to fetch instructions in the following cycle. If no entry is found for the control-flow operation in the BTB, the front-end stalls until the control-flow instruction is decoded (for direct branches) or executed (for indirect branches). Such misses occur the first time a taken branch is encountered. The BTB also suffers from misses due to its finite capacity. In some cases, the entries in the BTB are evicted to make room for new data. Unfortunately, once an entry is evicted from the BTB, the information it contains is lost and needs to be recreated from scratch if needed. Indirect branches with multi-branch targets also cause

BTB misses whenever their target address change as each BTB entry only holds a single target address.

Figure 2.4 quantifies the performance penalty due to branch target mispredictions for a 4-way superscalar processor running the SPEC CPU benchmarks. The BTB is 4-way configured with 1K entries. The results are compared to a similarly configured processor with a perfect target predictor that always predicts the correct branch target. The cost of branch target mispredictions for INT applications is 3%. For FP applications, the cost is negligible. For most applications in the SPEC benchmarks suite, the size of the BTB simulated is large enough to hold most of the taken branches. Combined, the branch prediction detractors cost 22% and 9% in performance for INT and FP applications respectively.

2.3 Front-End and Energy Consumption

Energy efficiency is a critical design concern for modern processors as it dictates if the processor can be used in portable or deeply embedded systems for which battery size and lifetime are vital parameters. Energy efficiency is also essential for dense server systems (e.g. blades), where thousands of processors may be packed in a single colocation site. High energy consumption can severely limit the server scalability, its operational cost, and its reliability [28].

Instruction cache misses have a negative effect on the processor energy efficiency. First, the front-end and possibly the pipeline are idle wasting energy due to leakage during an instruction cache miss. Second, serving a cache miss requires access to large structures at the lower levels of the hierarchy with high dynamic energy consumption. Multi-cycle instruction cache accesses also have a negative effect on the processor energy efficiency as executing mispredicted instructions wastes energy. Figure 2.5 presents the total energy increase due to instruction cache misses and multi-cycle instruction cache accesses. The reported *Total Energy* includes all of the processor components (front-end, execution core,

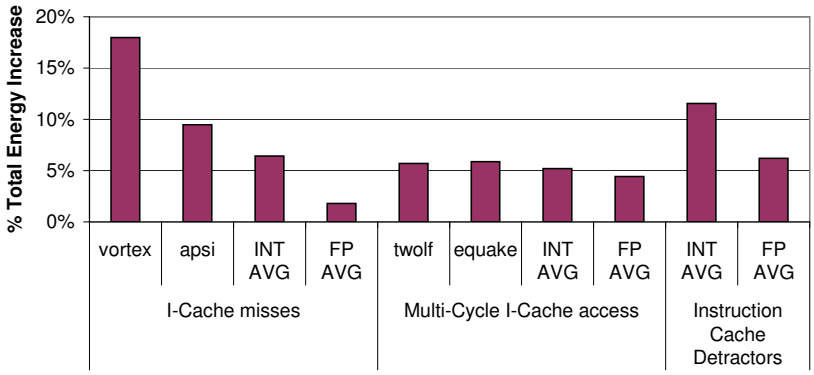


Figure 2.5: The percentage of total energy increase for a 4-way superscalar processor running SPEC CPU benchmarks due to instruction cache misses and access latency. The instruction cache is configured as 32 KByte, 4-way set-associative with 2-cycle access time.

and all caches). For INT applications, 12% of the processor total energy is wasted due to instruction cache misses and multi-cycle access time. For FP applications, 6% is wasted due to the instruction cache detractors.

Prediction accuracy also affects the processor energy efficiency. Anytime the front-end misspeculates, energy is wasted by fetching and executing erroneous instruction from the wrong execution path. Moreover, recovering from a branch misprediction is a costly operation in terms of energy, as the erroneous instruction must be removed from the pipeline. Figure 2.6 presents the energy wasted due to branch direction and target mispredictions. On average, 14% and 7% of the processor total energy are wasted by branch direction and target mispredictions for INT and FP applications respectively.

The front-end itself consumes a significant percentage of the processor total energy as it contains large memory arrays (instruction cache, predictor, BTB) that are accessed nearly every cycle. On average, 13% of the total energy is consumed in the front-end itself alone for a 4-way superscalar processor [115]. The front-end arrays are typically designed for high performance, which may not be very energy efficient. As an example, all of the components of the hybrid predictor are accessed in parallel to achieve a single

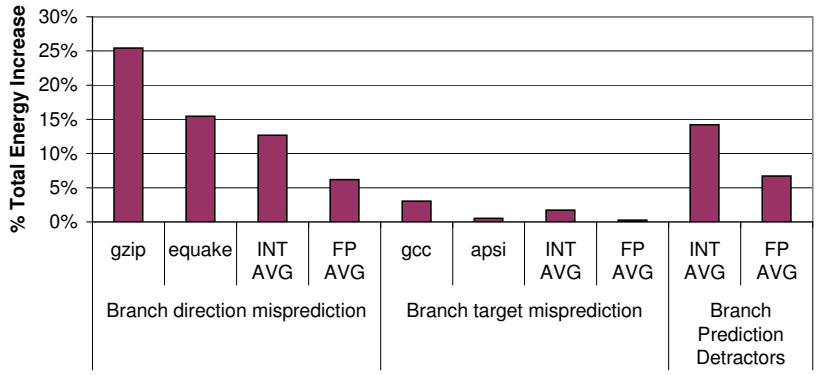


Figure 2.6: The percentage of total energy increase for a 4-way superscalar processor running SPEC CPU benchmarks due to branch direction and target mispredictions. The BTB is configured 4-way with 1K entries. The hybrid predictor has a 4K-counter selector, 4K-counter Gshare, and 1K-entry L1, 1K-counter L2 PAg.

cycle prediction time. This is not efficient as only one component of the predictor has the required data (e.g., the Gshare predictor for global correlating type of branches). Similarly, for a set-associative instruction cache, all of the data arrays are accessed in parallel to reduce the cache access time. Ideally, only the data array for the way that hits is needed to get the required data. Additionally, reading the complete cache line is not energy efficient as only a subset of the instructions in the line are typically used (e.g. 4 out of 16). A jump or a taken branch in the middle of the cache line would possibly force the front-end to start fetching from a different cache line. Instructions in the early part of a cache line, right before the target of a taken branch, are also less likely to be required. Significant energy could be saved if we only read the required words from the cache line.

Figure 2.7 compares the energy of ideal instruction cache, predictor, and BTB to the conventional structures. For the ideal instruction cache, only the required instructions are read from the cache line and only the data array for the way that hits is accessed. Similarly, for the ideal branch direction predictor, only one of the hybrid predictor components is accessed and trained. The ideal BTB is only accessed and trained for taken control-flow instructions. The instruction cache is configured as a 32 KByte, 4-way set associative,

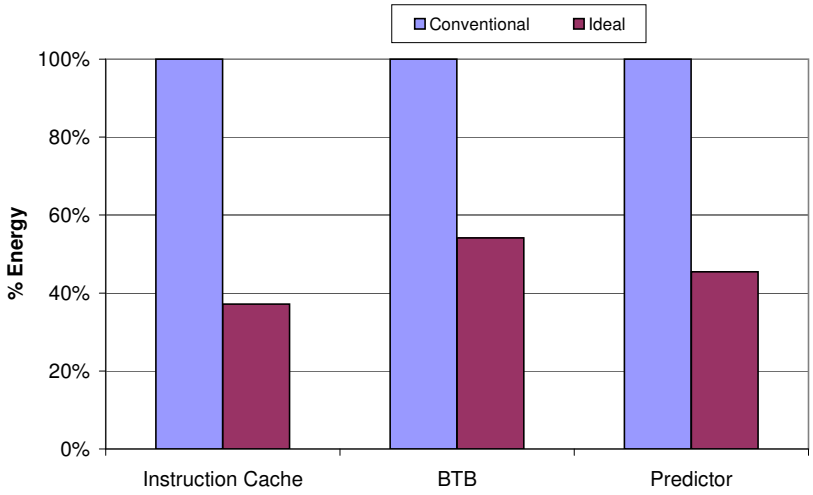


Figure 2.7: A comparison of energy consumption for an ideal instruction cache, predictor, and BTB for a 4-way superscalar processor running SPEC CPU benchmarks over conventional base design. The simulated processor is configured with a 32 KByte, 4-way, 2-cycle access instruction cache, 4-way, 1K-entry BTB, and a 4K-counter selector, 4K-counter Gshare, 1K-entry L1, 1K-counter L2 PAg hybrid predictor.

2-cycle access, 32 Byte line cache. The hybrid predictor has a 4K-counter selector, 4K-counter Gshare, and 1K-entry L1, 1K-counter L2 PAg. 4-way, 1K-entry BTB is also used. On average, 63% of the 4-way set-associative instruction cache energy can be saved using an efficient instruction cache. Similarly, 46% and 55% of the BTB and the predictor energy respectively could be saved if we access them ideally. The potentials of energy savings are significant; however, the challenge is to achieve energy efficiency without any compromise to neither complexity nor performance.

2.4 Related Work Overview

Given the effect of instruction delivery on the overall processor performance and energy consumption, there has been a significant volume of research targeting front-end detractors.

Several researchers have targeted reducing the number of instruction cache misses through either cache layout optimizations or prefetching. Cache optimizations include

mapping code into different cache areas to reduce conflict misses [11], code reordering and alignment to increase cache utilization [68], preventing infrequent executed code from polluting the instruction cache [35], using reconfigurable caches [24], and using software-assisted cache replacements techniques [46]. Many prefetching techniques have also been suggested to hide the latency of cache misses. They typically differ in the scheme used to predict prefetch addresses. Some of the techniques include sequential prefetching [103, 77], history-based schemes [104, 39, 48], wrong path prefetching [82], software cooperative approaches [61], and execution-based schemes [89, 21].

Moreover, many techniques have been proposed to reduce the branch misprediction penalty and improve the prediction accuracy. They include delayed branches [33], branch bypassing and multiple branch prefetching [92], branch folding [25], early resolution of branch decision [3], using multiple independent instruction streams in a shared pipeline [101], and the prepare-to-branch instruction [109]. Some of the techniques that target improving branch prediction accuracy include branch alignment optimizations [18], branch classification [19], and minimizing aliasing in predictors [72]. Of course, there is also significant research on using advanced dynamic predictors [98, 47].

There are also techniques that improve the front-end efficiency by predicting and fetching multiple basic blocks in a single cycle. Some of the techniques include block-based front-end engines [111], decoupled front-ends [88, 89], multi-block prediction [100, 99], control-flow prediction [83, 26], and parallel fetching of no contiguous instruction streams [74, 95]. Significant amount of front-end research has also focused on trace caches [94, 29], trace predictors [45], and trace construction [78].

Finally, there is also research that focuses on improving the front-end energy efficiency. Some techniques target improving the instruction cache energy consumption by way prediction [84], selective cache way access [2], sub-banking [30], tag comparison elimination [75, 114], and reconfigurable caches [87, 113]. Other techniques target improving the energy efficiency of the predictors using sub-banking [76], front-end gating [64], eliminating

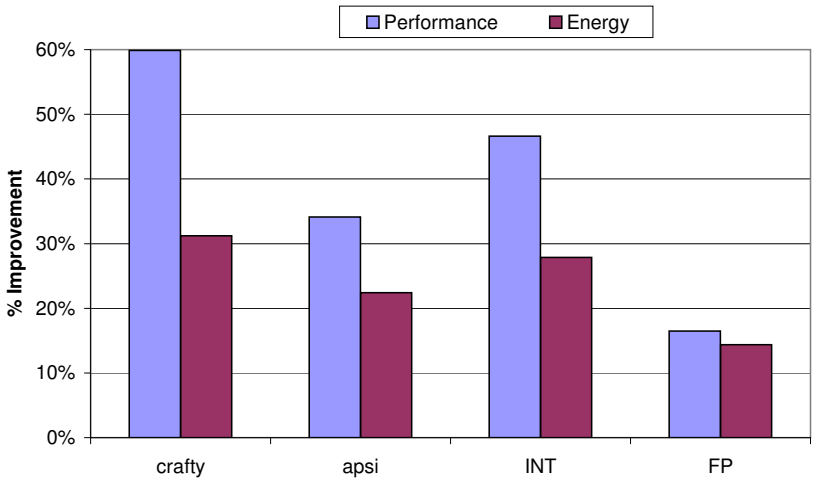


Figure 2.8: The percentage of performance and energy improvements for a 4-way superscalar processor running SPEC CPU benchmarks with a front-end that suffers no detractors and has optimal, energy-efficiency structures.

predictor and BTB accesses for non-branch instructions [76], using profile data to eliminate meta predictor [27] or to switch off part of the predictor [20], and selective predictor accesses to avoid using the predictor for well-behaved branches [8].

2.5 Ideal Front-End

While most of the techniques that are presented in Section 2.4 improve the front-end efficiency in some way or another, they typically provide partial solutions as each only tackles a single front-end issue. The goal of this work is to develop a unified framework that provides a complete solution to the front-end challenges and allows the front-end to achieve close to the optimal results for all of the efficiency metrics (performance, energy, power, code size, and die area). The differentiating factor of our approach is that we use an integrated, balanced software-hardware approach that provides capability to deal with all of the front-end challenges at once. Our approach builds upon previous research but introduces a novel instruction set architecture that allows software to provide hardware with

the information necessary for front-end optimizations. In the following chapters, we document this approach and demonstrate its significant advantages over the hardware-only or software-only approaches listed in Section 2.4.

Figure 2.8 quantifies the maximum performance and energy improvements we can achieve in a 4-way superscalar processor by providing an ideal front-end without any detractors. Up to 47% in performance improvement and 28% in energy saving could be achieved for INT applications. Even for FP applications, which are less susceptible to front-end detractors, 16% performance and 14% total energy could be improved. Similar gains can be also achieved by the front-end in other processor configurations such as narrow, in-order designs that are typically used in embedded systems. Of course, it is not possible to fully eliminate all of the front-end challenges. Nevertheless, we will demonstrate that a significant percentage of the ideal front-end benefits can be achieved by using software support to target multiple detractors.

Chapter 3

Block-Aware Instruction Set

Conventional ISAs provide no information to assist with the front-end challenges presented in the previous chapter. The ISA is structured to describe what needs to take place in the back-end only. Hence, the front-end hardware has to detect basic block boundaries, determine branch directions and targets, discover patterns (potentially repeatedly), and figure out how to balance over- and under-speculation.

The thesis of this work is to allow the software to assist the hardware in dealing with the front-end challenges. We modify the conventional instruction set architecture to provide additional information about the control-flow of the program at the granularity of basic blocks. The block-aware instruction set architecture (**BLISS**) defines basic block descriptors in addition to and separately from the actual instructions in each program. A descriptor provides sufficient information for fast and accurate control-flow prediction without accessing or parsing the instruction stream. It describes the type of the control-flow operation that terminates the block, its potential target, and the number of instructions in the basic block. This information is sufficient to tolerate the latency of instruction accesses, regulate the use of prediction structures, and direct instruction prefetching. Block descriptors also provide a general mechanism for compilers for passing software hints that can assist with a wide range of challenges at the hardware level.

Section 3.1 presents the block-aware instruction set architecture. In Section 3.2, we discuss the software hints and their possible usages. Section 3.3 overviews the tools and the experimental methodology. Section 3.4 provides an ISA-level characterization of BLISS using applications from the SPEC benchmark suite. In Section 3.5, we discuss the related research that this work is based on.

3.1 Instruction Set Architecture

The instruction set architecture (ISA) specifies the processor functionality and serves as the interface between hardware and software. A typical ISA defines the machine state (registers and memory) and a series of instructions that can operate on it. The block-aware instruction set architecture extends conventional ISAs with additional information about the program control-flow. The control-flow information is specified at the granularity of basic block (*BB*), which is a sequence of instructions starting at the target or fall-through of a control-flow instruction and ending with the next control-flow instruction or before the next potential branch target.

3.1.1 Basic Block Descriptors

BLISS stores the definitions for basic blocks in addition to and separately from the ordinary instructions they include. The code segment for a program is divided in two distinct sections. The first section contains descriptors that define the type and boundaries of blocks, while the second section lists the actual instructions in each block.

Figure 3.1 presents the format of a basic block descriptor (*B*BD). Each BBD defines the type of the control-flow operation that terminates the block such as a conditional branch or an unconditional jump. For fall-through basic blocks, we use the `FT` type. The `LOOP` type is a zero-overhead loop construct similar to that in the PowerPC ISA [67]. The BBD

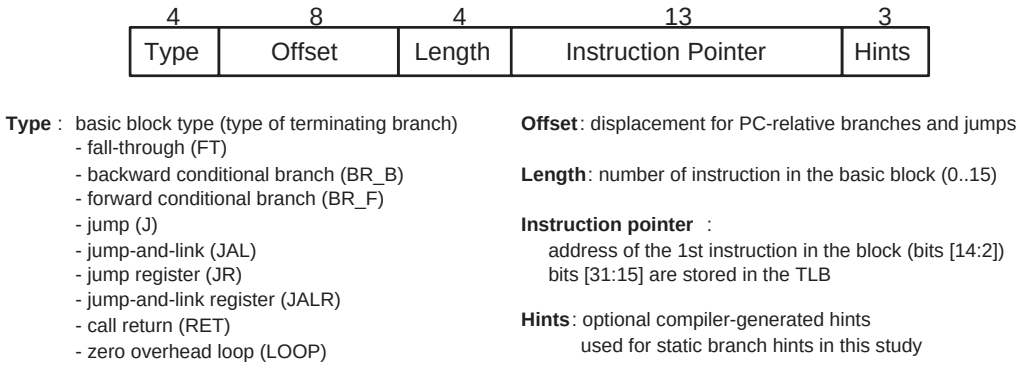


Figure 3.1: The 32-bit basic block descriptor format in BLISS.

also includes an offset field to be used for blocks ending with a branch or a jump with PC-relative addressing. The actual instructions in the basic block are identified by the pointer to the first instruction and the length field. Each BBD can point to up to 15 instructions at most. Basic blocks that are larger than 15 instructions use multiple BLISS descriptors. FT blocks can be larger than 15 instructions as their offset field is not required and is used to extend the length field instead. The BBD only provides the lower bits [14:2] of the instruction pointer, bits [31:15] are stored in the TLB. Section 4.1.1 explains in details the organization and operation of the TLB. The last BBD field contains optional compiler-generated hints. We discuss hints in details in Section 3.2. The overall BBD length is 32 bits.

With the BLISS ISA, there is only a single program counter (**PC**) that only points within the code segment for basic block descriptors. The PC does not point to the instructions themselves at any time. Every cycle, the PC is used to fetch basic block descriptors. The instruction pointer and the length fields available in the fetched descriptor are used to fetch the associated instructions in the next cycle. When all of the associated instructions are fetched, the BBD for the next basic block in the program order (PC+4 or PC+offset) is used for instruction fetching.

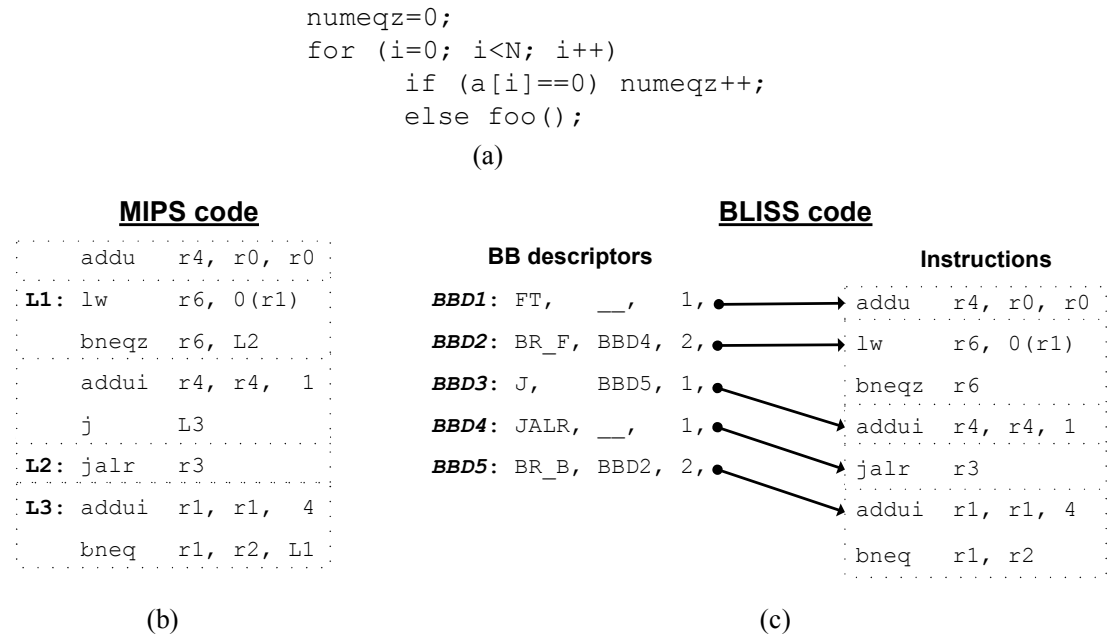


Figure 3.2: Example program in (a) C source code, (b) MIPS assembly, and (c) BLISS assembly. In (b) and (c), the instructions in each basic block are identified with dotted-line boxes. Register `r3` contains the address for the first instruction (b) or first basic block descriptor (c) of function `foo`. For illustration purposes, the instruction pointers in basic block descriptors are represented with arrows.

3.1.2 BLISS Code Example

To better understand how BLISS operates, we study the example program in Figure 3.2 that presents an example program that counts the number of zeros in array `a` and calls `foo()` for each non-zero element. With a RISC ISA like MIPS, the program requires 8 instructions (Figure 3.2.b). The 4 control-flow operations define 5 basic blocks. All branch conditions and targets are defined by the branch and jump instructions. With the BLISS equivalent of MIPS (Figure 3.2.c), the program requires 5 basic block descriptors and 7 instructions. All PC-relative offsets for branch and jump operations are available in BBDs. Compared to the original code, we have eliminated the `j` instruction. The corresponding descriptor (BBD3) defines both the control-flow type (`J`) and the offset; hence, the jump instruction itself is redundant. However, we cannot eliminate either of the two conditional branches (`bneqz`,

`bne`). The corresponding BBDs provide the offsets but not the branch conditions, which are still specified by the regular instructions. However, the regular branch instructions no longer need an offset field, which frees a large number of instruction bits. Similarly, we have preserved the `jalr` instruction because it allows reading the jump target from register `r3` and writing the return address in register `r31`.

3.1.3 Detailed Issues

The redistribution of control-flow information in BLISS between basic block descriptors and regular instructions does not change which programming constructs can be implemented with this ISA. Function pointers, virtual methods, jump tables, and dynamic linking are implemented in BLISS using jump-register BBDs and instructions in an identical manner to how they are implemented with conventional instruction sets. For example, the target register (`r3`) for the `jr` instruction in Figure 3.2 could be the destination register of a previous load instruction.

BLISS compresses the control-flow address space for programs as each BBD corresponds to 4 to 8 instructions on average. This implies that the BBD offset field requires fewer bits compared to the regular ISA. For cases where the offset field is not enough to encode the PC-relative address, an extra BBD is required to extend the address (see Table 3.5). The dense PCs used for branches (BBDs) in BLISS also lead to different interference patterns in the predictors than what we see with the PCs in the original ISA. For the benchmarks and processors studied, a 1% improvement in prediction accuracy is achieved with BLISS compared to the regular ISA due to the dense PCs.

Similar to the block-structured ISA [37, 70], BLISS treats each basic block as an atomic unit of execution. When a basic block is executed, either every instruction in the block is retired or none of the instructions in the block is retired. After any misprediction, the processor resumes execution at a basic block boundary and there is no need to handle

partially committed basic blocks. Atomic execution is not a fundamental requirement, but it leads to several software and hardware simplifications. For instance, it allows for a single program counter that only points within the code segment for basic block descriptors. The execution of all the instructions associated with each descriptor updates the PC so that it points to the descriptor for the next basic block in the program order (PC+4 or PC+offset). The PC does not point to the instructions themselves at any time.

Atomic basic block execution requires that the processor has sufficient physical registers for a whole basic block (15 in this case). For architectures with software handling of TLB misses, the associativity of the data TLB must be at least as high as the maximum number of loads or stores allowed per block. For the applications studied in Chapter 5, a limit of 8 load/stores per block does not cause a noticeable change in code size or performance. To handle precise exceptions, up to 16 additional physical registers and an 8-entry store buffer are required to allow register and memory writes to by undone if necessary in a case of an exception [70]. The cost of this backup hardware is minimal in many cases because these mechanisms are already present to support out-of-order, speculative execution.

3.2 Software Hints

The use of compiler-generated hints is a popular method for overcoming bottlenecks with modern processors [96]. The hope is that, given the higher level of understanding of program behavior or profiling information, the compiler can help the processor with selecting the optimal policies and with using the minimal amount of hardware in order to achieve the highest possible performance at the lowest power consumption or implementation cost. A compiler could attach hints to executable code at various levels of granularity: with every instruction, basic block, loop, function call, etc. Specifying hints at the basic block granularity allows for fine-grain information without increasing the length of all instruction encodings.

3.2.1 Potential Uses for BLISS Hints

BLISS provides a flexible mechanism for communicating compiler-generated hints at the basic block granularity using the last field in each basic block descriptor in Figure 3.1. Since descriptors are fetched early in the processor pipeline, the hints can be useful with tasks and decisions at any part of the processor (control-flow prediction, instruction fetch, instruction scheduling, etc.). The following is a non-exhaustive list of potential uses of the hints mechanism.

- **Code density**: The hints field can be used to aggressively interleave 16-bit and 32-bit instructions at basic block granularity without the overhead of additional instructions for switching between 16-bit and 32-bit modes [36]. The block descriptor identifies if the associated instructions use the short or long instruction format. No new instructions are required to specify the switch between the 16-bit and 32-bit modes. Hence, frequent switches between the two modes incur no additional runtime penalty. Since interleaving is supported at basic block granularity, any infrequently used basic block within a function or loop can use the short encoding without negative side effects. Chapter 7 evaluates using this type of hints for embedded processors.

- **Power savings**: The hints field specifies if the instructions for the basic block use a hardware resource such as the floating-point unit. This allows early detection of the processor components necessary to execute a code segment. Clock and power distribution can be regulated aggressively without suffering stalls during reactivation.

- **VLIW issue**: The hints field is used as a bit mask that identifies the existence of dependencies between consecutive instructions in the basic block. This allows for simpler logic for dependence checks within each basic block and instruction scheduling.

- **Extensive predication**: The hints field specifies one or two predicate registers used by the instructions in the basic block. This allows for a large number of predicate registers in the ISA without expanding every single instruction by 4 to 5 bits.

- **Simpler renaming**: The hints field specifies the live-in, live-out, and temporary registers for the instructions in the basic block. This allows for simpler renaming within and across basic blocks [70].

- **Cluster selection**: For a clustered processor, the hints field specifies how to distribute the instructions in this basic block across clusters given the dependencies they exhibit. Alternatively, the hints field can specify if this basic block marks the beginning of a new iteration of a parallel loop, so that a new cluster assignment can be initiated [73].

- **Selective pipeline flushing**: The hints can specify reconvergence points for if-then-else and switch statements so that the hardware can apply selective pipeline flushing on branch mispredictions.

Some of the examples above require a hints field that is longer than the 3 bits allocated in the format in Figure 3.1. Hence, there can be a tradeoff between the benefits from using the hints and the drawbacks from increasing the BBD length. For the hints usage studies in this thesis (prediction hints and code density hints), 3-bit fields were sufficient.

It is interesting to note that attaching hints to BBDs has no effect on the structure and code density of the instruction section of each program and its footprint and miss rate in the instruction cache. One could even distribute executable code with multiple versions of hints. The different versions can either represent different uses of the mechanism or hints specialized to the characteristics of a specific microarchitecture. Merging the proper version of hints with the block descriptors can be done at load time or dynamically, as pages of descriptors are brought into main memory.

3.2.2 Case Study: Branch Prediction Hints

To illustrate the usefulness of the hints mechanism, we use it to implement software hints for branch prediction [85]. The compiler uses 3 bits to provide a static or profile-based indication on the predictability of the control-flow operation at the end of the basic block. Two bits select one of the four predictability patterns:

- **Statically predictable**: fall-through basic blocks, unconditional jumps, or branches that are rarely executed or highly biased. For such descriptors, static prediction is as good as dynamic.

- **Dynamically predictable**: conditional branches that require dynamic prediction but do not benefit from correlation. A simple bimodal predictor is sufficient.

- **Locally predictable**: conditional branches that exhibit local correlation. A two-level, correlating predictor with per-address history is most appropriate for such branches (e.g. PAg [112]).

- **Globally predictable**: branches that exhibit global correlation. A two-level, correlating predictor with global history is most appropriate for such branches (e.g. Gshare or GAg [69, 112]).

We use the third bit to provide a default taken or not-taken static prediction outcome. With non-statically predictable descriptors, the static outcome can only be useful with estimating confidence or initializing the predictor. For statically predictable basic blocks, the hints allow us for accurate prediction without accessing prediction tables. Hence, there is reduced energy consumption and less interference. For dynamically predictable basic blocks, the hints allow us to use a subset of the hybrid predictor and calculate confidence.

3.3 Tools and Methodology

To experiment with the BLISS ISA, we developed a set of tools to generate the binary programs and simulate their execution on modern processors. This section reviews the tools necessary for the ISA-level characterization of BLISS. We discuss tools for performance and energy analysis in Section 5.1.2.

We generate BLISS executables using a static binary translator, which can handle arbitrary programs from high-level languages like C or Fortran. Figure 3.3 presents the flow diagram for the translator. The translator consists of three passes. The first pass parses the binary executable for the MIPS architecture, finds all symbols and relocations, identifies all basic blocks, and creates the initial basic block descriptors (type field only). The second pass optimizes the code by removing redundant jump instructions, transforming loops to use the LOOP construct, and eliminating redundant branch instructions that perform a simple test (equal/not equal to zero) to a register value produced within the same basic block by a simple arithmetic or logical operation. This pass also implements the code optimizations for code density which are presented in Chapter 7. The final pass assigns the length, target, IP, and hints to descriptors, fixes relocations, generates the new binary and section headers, and finally outputs the new BLISS executable. The generation of BLISS executable could also be done using a transparent, dynamic compilation framework [6].

Our simulation framework is based on the Simplescalar/PISA 3.0 toolset [16] which we modified for the BLISS ISA. For ISA evaluation, we study benchmarks from the SPEC CPU2000 suite using their reference datasets [38]. For benchmarks with multiple datasets, we run all of them and calculate the average. The benchmarks are compiled with gcc at the -O3 optimization level. With all benchmarks, we skip the first billion instructions in each dataset and simulate another billion instructions for detailed analysis.

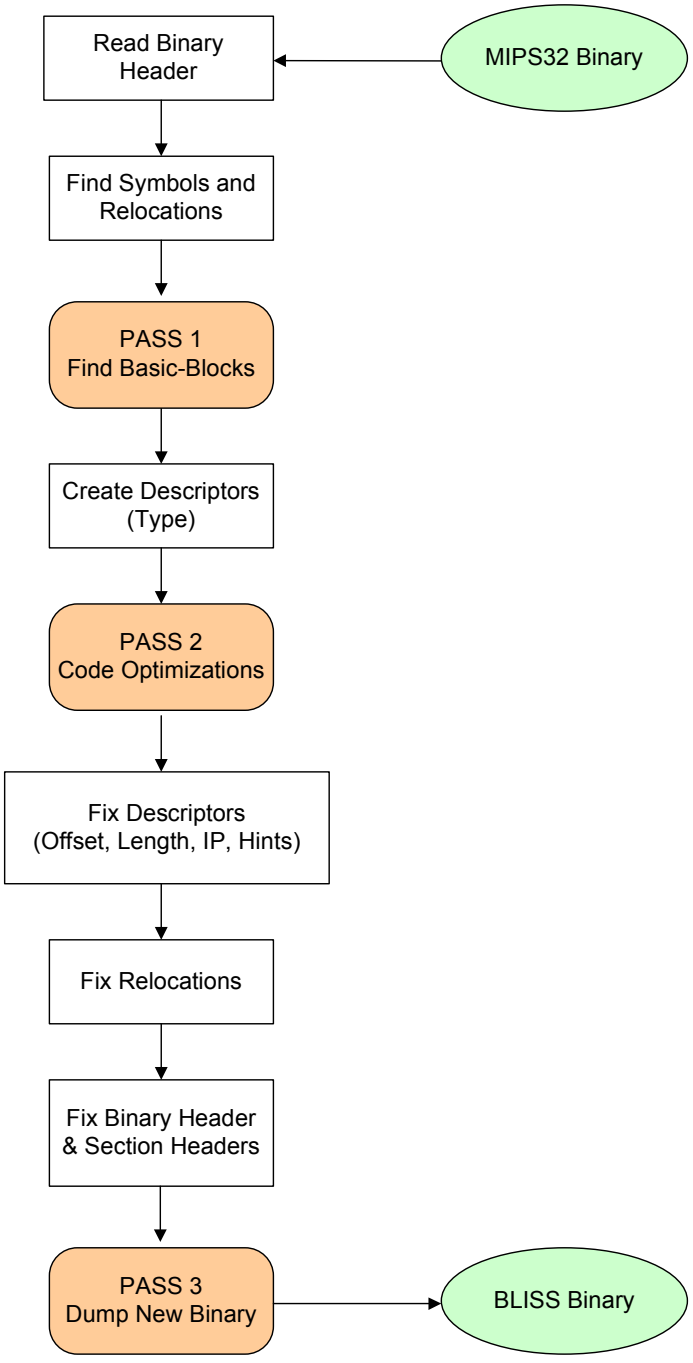


Figure 3.3: Flow diagram for the BLISS static binary translator.

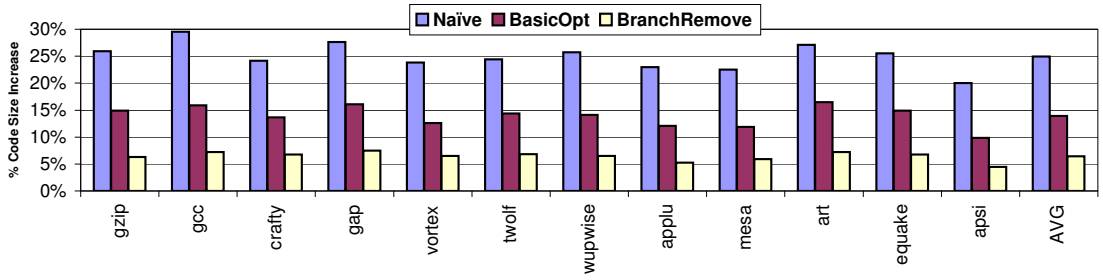


Figure 3.4: Static code size increase for BLISS over the MIPS-32 ISA. Positive increase means that the BLISS executables are larger.

3.4 ISA-Level Characterization

In this section, we present an ISA-level evaluation for BLISS. We study first the effect of the additional descriptors on the static code size. We also present statistics about the static and dynamic distribution of descriptor types and lengths.

3.4.1 Static Code Size

Figure 3.4 presents the percentage of code size increase for BLISS over the MIPS ISA that it is based on. Direct translation (*Naïve* bar) of MIPS code introduces one basic block descriptor every four instructions and leads to an average code size increase of 25%. Basic optimization (*BasicOpt* bar) reduces the average code size increase to 14%. The basic optimization targets the removal of redundant jump instructions (see example in Figure 3.2).

The *BranchRemove* bar in Figure 3.4 shows that the BLISS handicap can be reduced to 6.5% by removing conditional branch instructions that perform a simple test (equal/not equal to zero) to a register value produced within the same basic block by a simple arithmetic or logical operation. An extra bit is required for annotating the instruction producing the required register. This extra bit is readily available in most of the MIPS instructions using the register format. For instructions using the immediate format, additional opcodes are used for few of the most commonly used instructions.

	MIPS-32 Code Size (KBytes)	Number of Basic Blocks	Number of Descriptors	Jump Instructions Removed	Branch Instructions Removed
gzip	100	6478	6630	2704	2197
gcc	1229	91716	93053	42400	27196
crafty	217	12849	13404	5484	3842
gap	485	33754	34287	14286	10616
vortex	484	28388	29507	13472	7515
twolf	214	12941	13401	5236	4141
wupwise	100	6337	6577	2736	1949
applu	116	6277	6795	2743	2020
mesa	512	27705	29518	12912	7830
art	59	4043	4117	1519	1405
equake	61	3848	3982	1531	1275
apsi	188	8632	9657	4190	2593

Table 3.1: Statistics for the BLISS code size.

Table 3.1 presents additional statistics about the size, number of basic blocks, number of BBDs, number of jump instructions removed, and number of branch instructions eliminated for the studied applications. All applications require more BBDs than the number of basic blocks they have. On average, 4% more BBDs are used than the number of basic blocks. This is due to two main reasons. First, basic blocks with 16 instructions or more in the original MIPS code use multiple BLISS descriptors because each BBD can point to up to 15 instructions. Second, any BBD requiring more than eight bits for its offset field needs an extra BBD to extend its offset field (see Table 3.5).

Finally, BLISS facilitates two more optimizations that allow the BLISS code size to be even smaller than the original MIPS code. We study them in details in Chapter 7.

3.4.2 ISA Statistics

Table 3.2 presents the static distribution of descriptor types and Table 3.3 presents the dynamic distribution of descriptor types for BLISS. Most programs include a significant

	FT	BR_F	BR_B	J-JR	RET	JAL-JALR	LOOP
gzip	20.5%	39.3%	6.1%	13.0%	3.7%	15.2%	2.2%
gcc	13.3%	42.0%	4.0%	12.0%	2.4%	25.2%	1.1%
crafty	18.7%	39.6%	5.3%	11.6%	2.0%	21.4%	1.5%
gap	20.2%	39.4%	4.3%	14.4%	2.9%	16.8%	1.9%
vortex	14.3%	40.4%	1.5%	9.3%	3.7%	30.5%	0.3%
twolf	21.8%	38.7%	7.0%	10.6%	2.3%	17.6%	2.1%
wupwise	20.4%	36.8%	5.7%	14.8%	3.8%	17.4%	1.3%
applu	21.4%	36.0%	6.6%	14.2%	3.6%	16.2%	2.1%
mesa	19.1%	38.1%	2.3%	20.8%	4.1%	13.2%	2.4%
art	22.0%	40.2%	5.9%	13.0%	3.4%	13.0%	2.5%
equake	21.7%	38.6%	5.9%	12.6%	3.4%	15.5%	2.3%
apsi	21.7%	31.5%	5.2%	11.3%	3.5%	24.4%	2.4%
Average	19.6%	38.4%	5.0%	13.1%	3.2%	18.9%	1.8%

Table 3.2: Static distribution of BBD types for BLISS code.

	FT	BR_F	BR_B	J-JR	RET	JAL-JALR	LOOP
gzip	19.8%	45.5%	5.0%	5.8%	4.4%	4.4%	15.1%
gcc	54.9%	21.3%	11.3%	3.1%	2.3%	2.3%	4.9%
crafty	8.2%	25.0%	2.0%	2.9%	4.4%	4.4%	53.1%
gap	29.1%	34.0%	5.8%	4.4%	4.2%	4.2%	18.3%
vortex	18.9%	54.3%	0.5%	1.7%	10.3%	10.3%	3.8%
twolf	14.7%	41.5%	15.4%	2.0%	7.1%	7.1%	12.1%
wupwise	22.2%	38.1%	5.3%	3.9%	13.2%	13.2%	4.2%
applu	30.7%	8.6%	14.5%	0.0%	0.0%	0.0%	46.1%
mesa	21.4%	46.5%	2.1%	5.9%	8.1%	8.1%	7.9%
art	8.8%	33.8%	2.0%	0.0%	0.0%	0.0%	55.4%
equake	20.0%	23.7%	20.5%	4.2%	1.1%	1.1%	29.3%
apsi	22.8%	31.4%	5.3%	3.8%	6.1%	6.1%	24.5%
Average	22.6%	33.6%	7.5%	3.1%	5.1%	5.1%	22.9%

Table 3.3: Dynamic distribution of BBD types for BLISS code.

	0-3	4-7	8-11	12-15
gzip	58.3%	25.6%	12.2%	3.8%
gcc	74.2%	17.0%	6.9%	1.9%
crafty	29.7%	62.0%	3.8%	4.5%
gap	52.7%	21.3%	4.1%	21.9%
vortex	63.7%	11.8%	14.2%	10.3%
twolf	59.3%	19.9%	18.3%	2.6%
wupwise	63.2%	11.0%	6.2%	19.6%
applu	23.2%	18.6%	11.3%	46.9%
mesa	54.9%	23.7%	7.3%	14.1%
art	52.4%	20.7%	2.7%	24.2%
equake	30.0%	40.9%	18.2%	10.9%
apsi	46.5%	28.4%	8.9%	16.2%
Average (INT)	56.3%	26.3%	9.9%	7.5%
Average (FP)	45.0%	23.9%	9.1%	22.0%
Average	50.7%	25.1%	9.5%	14.7%

Table 3.4: Dynamic distribution of BBD lengths for BLISS code.

number of fall-through descriptors. For integer applications, this is mostly due to the large number of labels in the original MIPS code (potential targets of control-flow operations). For floating-point applications, this is mostly due to the many basic blocks with 16 instructions or more in the original MIPS code. Such basic blocks use multiple BLISS descriptors because each BBD can point to up to 15 instructions. On average, 1.9% and 2.8% of the static basic blocks contain more than 15 instructions for INT applications and FP applications respectively. Even though 1.8% of static BBDs on average use the LOOP construct, they account for almost 23.0% of executed BBDs.

Table 3.4 shows the dynamic distribution of descriptor lengths. It is interesting to notice that, even for INT applications, an average of 40% of the executed basic blocks include more than 4 instructions. This implies that making one prediction for every 4 instructions fetched from the instruction cache is often wasteful. On average, 94% of the executed basic blocks contain 15 instructions or less. Therefore, restricting the BBD length to 15

	0-2	3-5	6-8	9-11	11 or more
gzip	72.9%	11.4%	5.7%	10.0%	0.0%
gcc	56.7%	15.1%	6.1%	5.4%	16.6%
crafty	63.5%	12.4%	4.5%	16.9%	2.7%
gap	74.9%	10.7%	3.5%	3.5%	7.5%
vortex	57.7%	11.7%	2.9%	7.8%	20.0%
twolf	70.5%	11.0%	4.3%	7.4%	6.8%
wupwise	72.3%	12.7%	6.0%	9.0%	0.0%
applu	75.4%	11.0%	4.3%	9.2%	0.0%
mesa	69.5%	11.8%	9.2%	2.9%	6.6%
art	74.2%	11.2%	7.5%	7.1%	0.0%
apsi	69.3%	9.2%	3.7%	16.9%	0.9%
Average	68.8%	11.6%	5.2%	8.7%	5.6%

Table 3.5: Static distribution of the number of offset bits required for the BBDs in the BLISS code.

instructions is a good compromise between the descriptor length and the basic block length. Overall, the average dynamic basic block length is 7.7 instructions (5.8 for integer, 9.7 for floating-point), while the static average length is 3.7 instructions (3.5 for integer, 3.9 for floating-point).

Table 3.5 shows the number of offset bits required for BBDs. On average, 80% of BBDs require less than 3 bits (FT BBDs require 0 bits). BBDs that require offset field larger than 8 bits can be converted to use indirect addressing mode or extra BBDs are used to extend the address. As discussed in the previous section, this partially explains the difference between the number of basic blocks and the number BBDs for the applications shown in Table 3.1.

3.5 Related Work

Many researchers have observed that conditional branches and other control-flow operations serve multiple functions. They define basic block boundaries, provide the target address, and compute the branch condition. Several instruction sets define *prepare-to-branch*

instructions that define the first two functions and can be scheduled early in the instruction stream [51, 96]. The basic block descriptors in BLISS are basically prepare-to-branch instructions moved into a separate memory region. Hence, there is no need to access the instruction cache and parse the instruction stream in order to locate them.

The Block-structured ISA (*BSA*) [37, 70] defines basic blocks as atomic execution units but leaves their definitions within the regular instruction stream as well. BSA exploits atomic BB execution in order to reverse the ordering of instructions within each BB to simplify renaming.

The decoupled control-execute (DCE) architectures use a separate instruction set with distinct architectural state for control-flow calculation [107, 65]. DCE instruction sets allow the front-end to become an independent processor that can resolve control-flow and prefetch instructions tens to hundreds of cycles ahead of the execution core. However, DCE architectures are susceptible to deadlocks and have complicated exception handling. The basic block descriptors in BLISS are not a stand-alone ISA and do not define or modify any architectural state, hence eliminating the deadlock scenarios with decoupled control-execute ISAs. While most DCE ISAs have been developed for and evaluated with single-issue, in-order processors, BLISS works also for wide-issue speculative designs.

Several instruction sets allow for compiler-generated hints with individual branch and load/store instructions [96]. BLISS, on the other hand, provides a general mechanism for software hints at basic block granularity. The mechanism can support a variety of uses as explained in Section 3.2.

3.6 Summary

In this chapter, we presented the block-aware instruction set architecture that defines basic block descriptors in addition to and separately from the actual instructions in each program. A descriptor provides sufficient information for fast and accurate control-flow prediction

without accessing or parsing the instruction stream. It describes the type of the control-flow operation that terminates the block, its potential target, and the number of instructions in the basic block. This information is sufficient to tolerate the latency of instruction accesses, regulate the use of prediction structures, and direct instruction prefetching. Finally, block descriptors also provide a general mechanism for compilers for passing software hints that can assist with a wide range of challenges at the hardware level.

The instruction set level analysis demonstrated that basic block descriptors could be provided at a minimal impact to application code density (approximately 5%). The average basic block describes 5.8 to 9.7 instructions, which shows that limiting the length of basic blocks to 15 instructions provides a good compromise between the descriptor overhead and the block length. It also verifies that making prediction for every 4 instructions fetched with conventional ISAs is often wasteful.

In the following chapter, we will discuss the processor microarchitecture that implements the BLISS ISA. In Chapter 5, we will quantify the performance benefit of the BLISS ISA and microarchitecture.

Chapter 4

Front-End Architecture for the Block-Aware ISA

The BLISS instruction set allows the processor front-end to decouple branch prediction from instruction fetching. The software-defined basic block descriptors provide sufficient information for fast and accurate control-flow prediction without accessing or parsing the instruction stream. Decoupling allows us to remove the instruction cache access from the critical path of accurate prediction. Hence, instruction cache latency no longer affects the prediction accuracy. The descriptors also provide an early, yet accurate view into the instruction address stream and can be used for instruction prefetching to reduce the impact of instruction cache misses. Furthermore, the control-flow information available in descriptors allows for judicious use of branch predictors, which reduces interference and training time and improves overall prediction accuracy.

This chapter presents the decoupled front-end for the BLISS ISA and its benefits over a conventional processor front-end. Section 4.1 describes the architecture and operation of the BLISS-based front-end. Certainly, a decoupled front-end can be implemented without the ISA support provided by BLISS. Section 4.2 describes a state-of-the-art decoupled front-end without the software support and explains the advantages that BLISS introduces

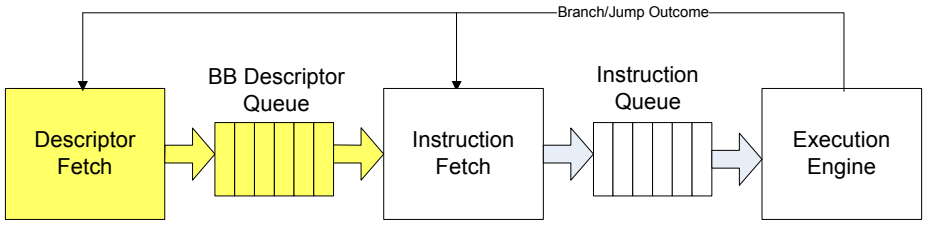


Figure 4.1: A simplified view of the BLISS decoupled processor.

over such designs. Section 4.3 discusses related work. While the chapter primarily focuses on architectural details and issues, Chapter 5 provides the evaluation results that clearly demonstrate the advantages of BLISS over both a conventional front-end and a state-of-the-art decoupled front-end without the ISA support.

4.1 Block-Aware Front-End Design

The BLISS ISA suggests a superscalar front-end that fetches the basic block descriptors (**BBDs**) and the associated instructions in a decoupled manner. The basic block queue (**BBQ**) in Figure 4.1 decouples control-flow predictions from instruction cache accesses. Each cycle, a BBD is fetched from the descriptor cache and pushed into the BBQ. The fetched BBD provides sufficient information to accurately predict the next BBD to be fetched in the following cycle. The content of the BBQ is used to fetch the required instructions form the instruction cache and deliver them to the back-end of the processor for execution.

4.1.1 Microarchitecture and Operation

Figure 4.2 presents the microarchitecture of a BLISS-based decoupled front-end. Compared to a conventional front-end, we replace the branch target buffer (BTB) with a *BB-cache* that caches the block descriptors in programs. The basic block descriptors fetched from the BB-cache provide the front-end with the architectural information necessary for

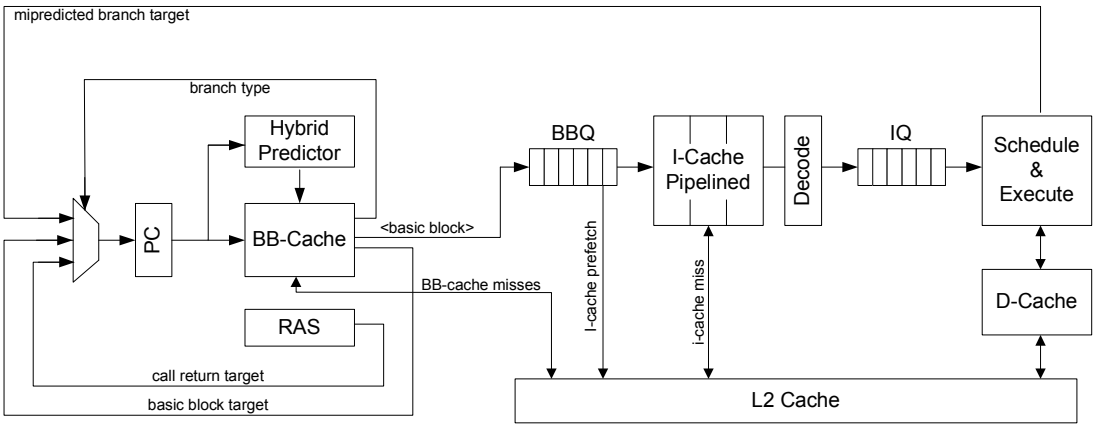


Figure 4.2: A decoupled front-end for a superscalar processor based on the BLISS ISA.

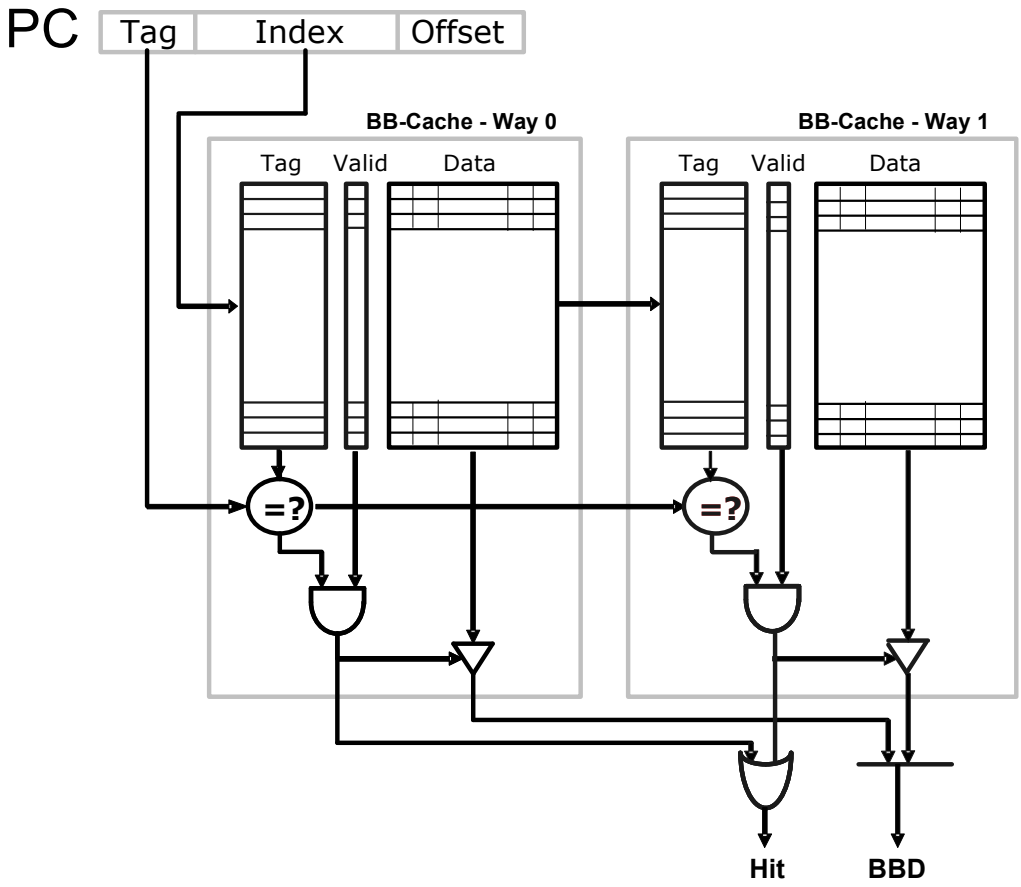
control-flow prediction in a compressed and accurate manner. Since descriptors are stored separately from ordinary instructions, their information is available for front-end tasks before instructions are fetched and decoded. The sequential target of a basic block is always at address PC+4, regardless of the number of instructions in the block. The non-sequential target (PC+offset) is also available through the offset field for all blocks terminating with PC-relative control-flow instructions. For register-based jumps, the non-sequential target is provided by the last regular instruction in the basic block through a register specifier. Basic block descriptors provide the branch condition when it is statically determined (all jumps, return, and fall-through blocks). For conditional branches, the descriptor provides type information (forward, backward, loop) and hints which can assist with dynamic prediction. The actual branch conditional is provided by the last regular instruction in the basic block.

The BLISS front-end operation is simple. On every cycle, the BB-cache is accessed using the PC. On a miss, the front-end stalls until the missing descriptor is retrieved from the memory hierarchy (L2-cache). On a hit, the BBD and its predicted direction/target are pushed in the *basic block queue (BBQ)*. The predicted PC is used to access the BB-cache in the following cycle. Instruction cache accesses use the instruction pointer and length fields in the descriptors available in the BBQ.

Figure 4.3 shows the BLISS descriptor cache. The offset field in each descriptor is stored in the BB-cache in an expanded form that identifies the full target of the terminating branch. For PC-relative branches and jumps, the expansion takes place on BB-cache refills from lower levels of the memory hierarchy, which eliminates target mispredictions even for the first time the branch is executed. For register-based jumps, the offset field is available after the first execution of the basic block. The BB-cache also integrates a simple bimodal predictor. The predictor provides a quick direction prediction along with the target prediction. The simple prediction is verified one cycle later by a large, tagless, hybrid predictor. In the case of a mismatch, the front-end experiences a one cycle stall.

The BB-cache stores multiple sequential BBDs. Long BB-cache lines exploit spatial locality in descriptor accesses and reduce the storage overhead for tags. For our experiments, each BB-cache line stores eight sequential BBDs. This provides a balance between spatial locality and the tag overhead. With the BTB, on the other hand, each cache line describes a single target address (one tag per one BTB entry) for greater flexibility with replacement. The increased overhead for tag storage in BTB balances out the fact that the BB-cache entries are larger due to the instruction pointer field. For the same number of entries, the BTB and BB-cache implementations require the same number of SRAM bits. Same number of entries means that the number of branches that BTB can store is equal to the number of basic block descriptors the BB-cache can store.

Figure 4.4 shows the organization and operation of the translation look-aside buffer (**TLB**) used for caching virtual to physical address mappings for BLISS descriptors. We show a fully associative TLB as an example. However, just like any other TLB, it can have other associativities. The TLB operates in the following manner. The BBD virtual page frame address is compared with all of the tags. In case of a match, the BBD page offset is concatenated with the physical page frame address from the TLB to form the BBD physical address. The TLB also includes the higher bits [31:15] of the instruction pointer. Those bits are concatenated with the lower bits [14:2] available in the BBD to from the full



BB-cache Entry Format

type (4b)	target (30b)	length (4b)	instr. pointer (13b)	hints (2b)	bimod (2b)

Figure 4.3: The basic block descriptors cache (BB-cache) and the cache line format.

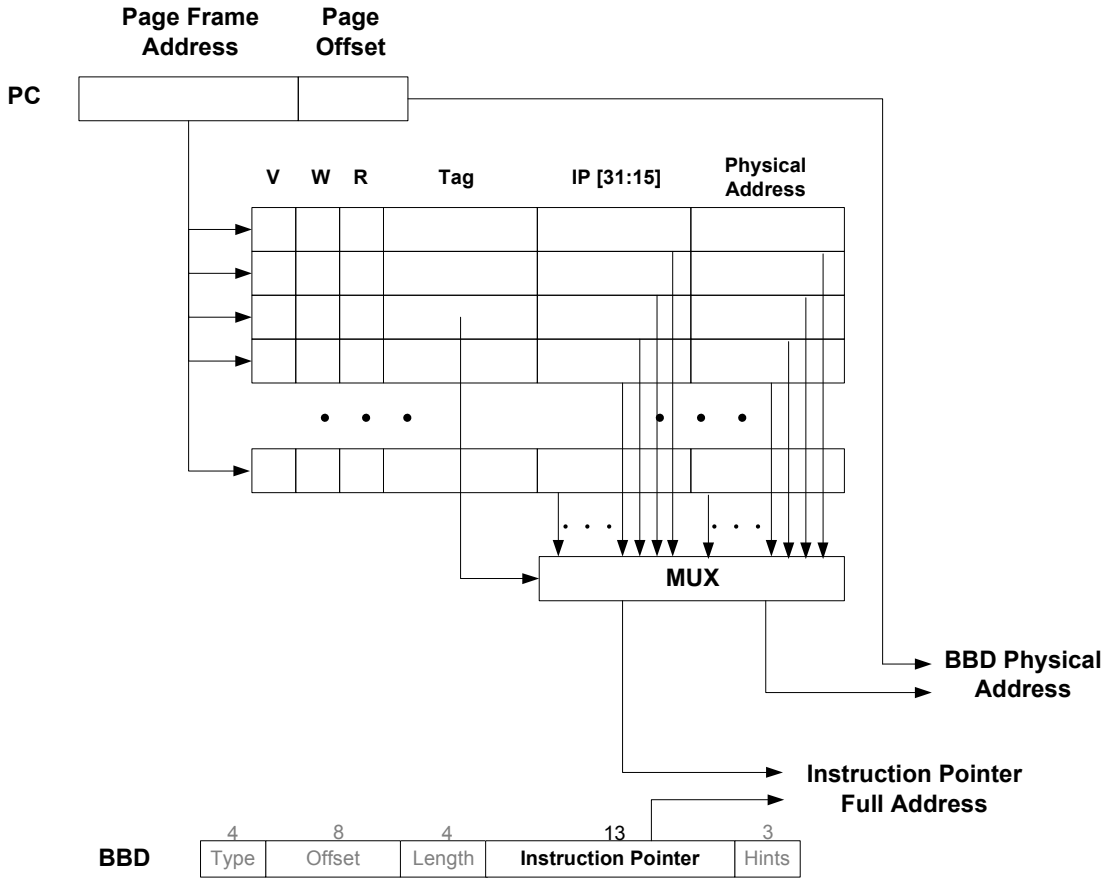


Figure 4.4: The block diagram and operation of the TLB for descriptor accesses.

instruction pointer address. Figure 4.4 only shows a single IP[31:15] field for each TLB entry which implies that instructions for all BBDs within a page must be within the same 32-KByte instruction frame. Alternatively, we can have multiple IP[31:15] fields within a TLB entry to support more flexibility.

4.1.2 Benefits of ISA Support for Basic Blocks

The BLISS front-end alleviates all of the shortcomings of a conventional front-end. The BLISS front-end improves the accuracy of branch direction and target prediction, removes the instruction cache access from the critical path of prediction, and minimizes the number

of instruction cache misses. Moreover, it allows energy-efficient accesses to the instruction cache and predictors.

Branch Prediction

The BLISS front-end minimizes the number of branch direction mispredictions. The PC in the BLISS ISA always points to basic block descriptors (i.e. control-flow instructions); the hybrid predictor is only used and trained for PCs that correspond to branches. With a conventional front-end, on the other hand, the PC may often point to non control-flow instructions which causes additional interference and slower training for the hybrid predictor. In addition, the exact address of the descriptor is used to access the predictor. This is not the case with a conventional front-end where the address of the first instruction in the fetch block is used to access the predictor instead.

The type of the BBD also allows us to selectively use the predictor in an accurate manner. For BBDs that correspond to blocks terminating with an unconditional control-flow instruction (J, JAL, etc), the hybrid predictor is not used nor trained. This reduces interference and saves energy in the predictor. Finally, the static prediction hints in basic block descriptors allow for judicious use of the hybrid predictor. Strongly biased branches do not use the predictor and branches that exhibit strong local or global correlation patterns use only one of its components. This further improves the prediction accuracy and leads to additional energy savings.

The BLISS front-end also minimizes the number of branch target mispredictions. The BB-cache stores the full address for PC-relative branches and jumps. The expansion takes place on BB-cache refills from lower levels of the memory hierarchy, which eliminates target mispredictions even for the first time the branch is executed. Unlike the BTB, when a BB-cache entry is evicted, the data is still available in lower memory hierarchy.

The BBQ decouples control-flow prediction from instruction fetching. Multi-cycle latency for large instruction cache no longer affects prediction accuracy, as the vital information for speculation is included in basic block descriptors available through the BB-cache (block type, target offset). Hence, the predictor can run ahead, even when the instruction cache experiences temporary stalls. Compared to the pipeline for a conventional ISA, the BLISS-based microarchitecture adds one pipeline stage for fetching basic block descriptors. The additional stage increases the misprediction penalty. This disadvantage of BLISS is more than compensated for by improvements in prediction accuracy due to reduced interference at the predictor.

Instruction Cache

The contents of the BLISS BBQ provide an early, yet accurate view into the instruction address stream and can be used for instruction prefetching to hide instruction cache misses. Figure 4.5 presents the prefetching scheme used with BLISS. The instruction cache is a non-blocking cache that can handle up to four misses concurrently. The instruction cache has a single read port that can be used by either the instruction fetch unit or the prefetcher. When the instruction cache is serving a miss or when the instruction queue is full, the port can be used by the prefetcher to lookup further instructions in the cache based on the content of the BBQ. If the instructions are not available in the instruction cache, the instructions are prefetched from lower memory hierarchy. The prefetched cache lines are brought into a separate buffer to prevent instruction cache pollution. Once the prefetched cache line is accessed by the instruction fetch unit, it is moved into the instruction cache.

Many prefetching techniques have been widely used with conventional instruction sets to hide the latency of cache misses. The simplest technique is the sequential prefetching [103, 77]. In this scheme, one or more sequential cache lines that follow the current fetched line are prefetched. Stream prefetching only helps with misses on sequential instructions.

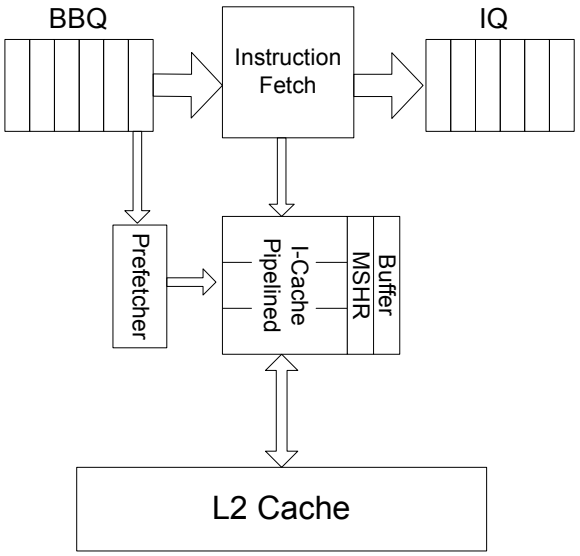


Figure 4.5: The prefetcher in the BLISS-based front-end.

With BLISS, the prefetcher initiates prefetches for cache lines on the predicted path of execution using the BBQ content. The advantage of such a scheme is that it can prefetch potentially useful instructions even for non-sequential access patterns as long as branch prediction is sufficiently accurate and requires no training. BLISS prefetching is similar to history-based schemes [104, 39, 48] that use the patterns of previous accesses to initiate the new prefetches. The difference is that we require no additional history tables as the BB-cache and the predictors act as an accurate history mechanism.

Energy Efficiency

BLISS allows us to achieve high energy efficiency in two ways: it reduces the wasted energy in the overall design and reduces the front-end energy. BLISS reduces the energy wasted on fetching and executing mispredicted instructions as a result of the improved prediction accuracy. By reducing execution time, the BLISS-based design also saves on the energy consumed by the clock tree and the processor resources even when they are stalling or idling.

The availability of basic block descriptors allows for energy optimizations that reduce the front-end energy consumption. Each basic block defines exactly the number of instructions needed from the instruction cache. Using segmented word lines [30] for the data portion of the instruction cache, we can fetch the necessary words while activating only the necessary sense-amplifiers in each case. As front-end decoupling tolerates higher instruction cache latency without loss in speculation accuracy, we can access first the tags for a set associative instruction cache, and in subsequent cycles, access the data only in the way that hits [91]. Furthermore, we can save decoding and tag access energy in the instruction cache by merging instruction cache accesses for sequential blocks in the BBQ that hit in the same instruction cache line. Finally, the front-end can avoid the access to some or all prediction tables for descriptors that are not conditional branches or for descriptors identified as statically predictable by branch hints. Chapter 6 discusses these optimization techniques in more details and evaluates them.

4.2 Hardware-only Decoupled Front-End

A decoupled front-end similar to the one in Figure 4.2 can be implemented without the ISA support provided by BLISS. The FTB design proposed by Reinman et. al. [89, 90] describes the latest of such design. In this section, we briefly describe their design and compare it to the BLISS-based front-end design.

4.2.1 The FTB Front-End Design

Figure 4.6 depicts the FTB front-end by Reinman et. al. [89, 90], which represents a comprehensive, high-performance architecture for a decoupled, block-based front-end. The design uses a *fetch target buffer (FTB)* as an enhanced basic block BTB [111]. Each FTB entry describes a *fetch block*, a set of sequential instructions starting at a branch target and ending with a strongly biased, taken branch or an unbiased branch [88]. A fetch block may

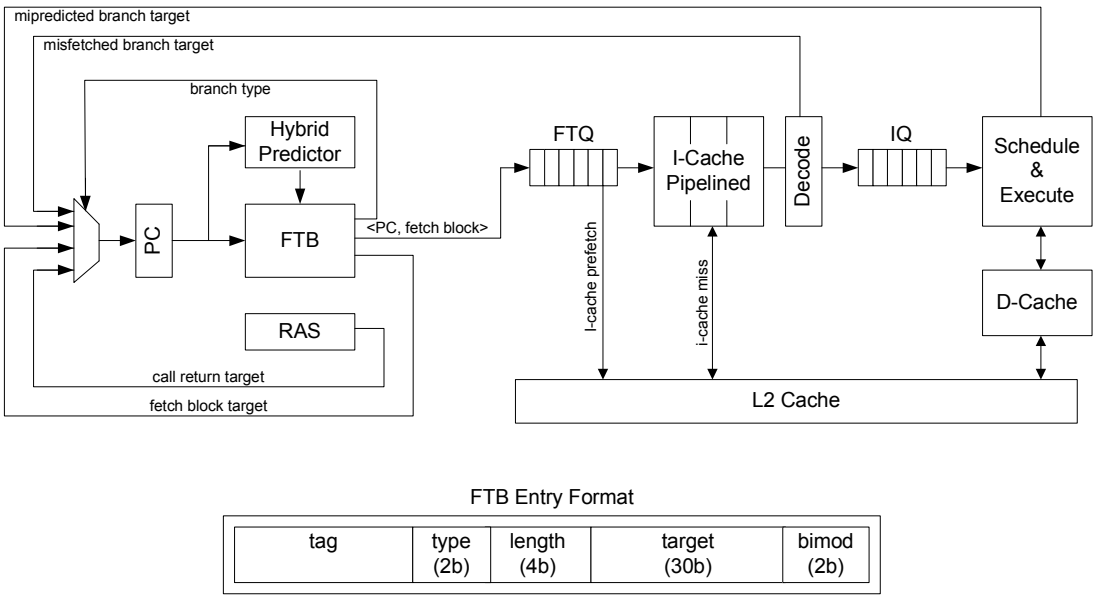


Figure 4.6: The FTB architecture for a decoupled, block-based front-end.

include several strongly biased, not-taken branches. Therefore, a fetch block may include several basic blocks. Apart from a tag that identifies the address of the first instruction in the fetch block, the FTB entry contains the length of the block, the type of the terminating branch or jump, its predicted target, and its predicted direction. Fetch blocks are created in hardware by dynamically parsing the stream of executed instructions and observing the behavior of control-flow instructions.

Each cycle, the FTB is accessed using the program counter. On an FTB hit, the starting address of the block (PC), its length, and the predicted direction and target are pushed in the *fetch target queue (FTQ)*. Similar to the BBQ, the FTQ decouples control-flow prediction from instruction cache accesses. On an FTB miss, the front-end injects into the FTQ maximum length (15 instructions), fall-through fetch blocks starting at the miss address, until an FTB hit occurs or the back-end of the processor signals a *misfetch* or a *misprediction*. A misfetch occurs when the decoding logic detects a jump in the middle of a fetch block. In this case, the pipeline stages behind decoding are flushed including the FTQ and the IQ,

a new FTB entry is allocated for the fetch block terminating at the jump, and execution restarts at the jump target. A misprediction occurs when the execution core retires a taken branch in the middle of a fetch block or when the control-flow prediction for the terminating branch (target or direction) proves to be incorrect. In either case, the whole pipeline is flushed and restarted at the branch target. If the fetch block was read from the FTB, the FTB entry is updated to indicate the shorter fetch block or the change in target/direction prediction. Otherwise, a new FTB entry is allocated for the block terminating at the mispredicted branch. Even though both misfetches and mispredictions lead to creation of new fetch blocks, no FTB entries are allocated for fall-through fetch blocks because sequential blocks are automatically fetched in the case of a FTB miss. Next section will review qualitatively the advantages and the disadvantages of the FTB design compared to the BLISS design.

4.2.2 Hardware vs. Software Basic Blocks

The FTB design encapsulates all the advantages of a decoupled, block-based front-end. Nevertheless, the performance of the FTB-based design is limited by inaccuracies introduced during fetch block creation and by the finite capacity of the FTB. When a jump instruction is first encountered, a misfetch event will flush the pipeline front-end in order to create the proper fetch block. When a taken branch is first encountered, a full pipeline flush is necessary to generate the proper FTB entry. This is also the case when a branch switches behavior from biased not-taken to unbiased or taken, in which case we need to shorten the existing fetch block. In addition, we lose accuracy at the predictor tables as the entry that was used for the branch at the end of the old block, will now be used for the branch that switched behavior. A fetch block is identified by the address of its first instruction and not by the address of the terminating branch or jump. Hence, as the length of a fetch block changes, the branch identified by its address also changes. The branch terminating the old block will need to train new predictor entries.

Moreover, the frequency of such problematic events can be significant due to the finite capacity of the FTB. As new FTB entries are created, older yet useful entries may be evicted due to capacity or conflict misses. When an evicted block is needed again, the FTB entry must be recreated from scratch leading to the misfetches, mispredictions, and slow predictor training highlighted above. In other words, an FTB miss can cost tens of cycles, the time necessary to refill the pipeline of a wide processor after one or more mispredictions. Finally, any erroneous instructions executed for the large, fall-through fetch blocks injected in the pipeline on FTB misses lead to wasted energy consumption. Using a smaller fetch block length may reduce the number of erroneous instructions executed; however, this will lead to the creation of smaller fetch-blocks that will negatively affect performance. The FTB design is incapable of merging smaller blocks into a single larger block.

The BLISS front-end alleviates the basic problems of the FTB-based design. First, the BLISS basic blocks are software defined. They are never split or recreated by hardware as jumps are decoded or branches change their behavior. In other words, the BLISS front-end does not suffer from misfetches or mispredictions due to block creation. Of course, BLISS still has mispredictions due to incorrect prediction of the direction or target of the branch terminating a basic block, but there are no mispredictions due to discovering or splitting fetch blocks. In addition, the PC used to index prediction tables for a branch is always the address of the corresponding BBD. This address never changes regardless of the behavior of other branches in the program, which leads to fast predictor training. Second, when new descriptors are allocated in the BB-cache, the old descriptors are not destroyed. As part of the program code, they exist in main memory and in other levels of the memory hierarchy (e.g. L2-cache). On a BB-cache miss, the BLISS front-end retrieves missing descriptors from the L2-cache in order of ten cycles in most cases. Given a reasonable occupancy in the BBQ, the latency of the L2-cache access does not drain the pipeline from instructions. Hence, the BLISS front-end can avoid the mispredictions and the energy penalty associated with recreating fetch blocks on an FTB miss.

The potential drawbacks of the BLISS front-end are the length of the basic blocks and the utilization of the BB-cache capacity. FTB fetch blocks can be longer than BLISS basic blocks as they can include one or more biased not-taken branches. Longer blocks allow the FTB front-end to fetch more instructions per control-flow prediction. However, the BLISS front-end actually fetches more *useful* instructions per control-flow prediction. Although the BLISS blocks are smaller, they are accurately defined by software. For FTB, the long, fall-through fetch blocks introduced on FTB misses contain large numbers of erroneous instructions that lead to misfetches, mispredictions, and slow predictor training. In addition, the IPC is typically constrained by the commit width of the back-end. The IPC for imbalanced back-ends are further constrained by instruction dependencies. Therefore, having larger fetch blocks may not improve performance. The BLISS front-end may also underutilize the capacity of the BB-cache by storing descriptors for fall-through blocks or blocks terminating with biased not-taken branches. This can lead to higher miss rates for the BB-cache compared to the FTB. In Chapter 5, we will show that the BB-cache achieves good hit rates for a variety of sizes and consistently outperforms an equally sized FTB.

4.3 Related Work

Block-based front-end engines were introduced by Yeh and Patt to improve prediction accuracy [111], with basic block descriptors formed by hardware without any additional ISA support. Decoupled front-end techniques have been explored by [17] and [105]. Reinman et al. combined the two techniques in the FTB decoupled front-end design [88, 89, 90]. Ramirez et al. applied an FTB-like approach to long sequential instruction streams created with code layout optimizations and achieved 4% performance improvement [86].

Significant amount of front-end research has also focused on trace caches [94, 29], trace predictors [45], and trace construction [78]. Trace caches have been shown to work

well with basic blocks defined by hardware [14, 49]. One can form traces on top of the basic blocks in the BLISS ISA. BLISS provides two degrees of freedom for code layout optimizations (blocks and instructions), which could be useful for trace formation and compaction.

Other research in front-end architectures has focused on multi-block prediction [100, 99], control-flow prediction [83, 26], and parallel fetching of no contiguous instruction streams [74, 95]. Such techniques are rather orthogonal to the block-aware ISA and can be used with a BLISS-based front-end engine.

4.4 Summary

This chapter presented the BLISS-based front-end that fetches the basic block descriptors and the associated instructions in a decoupled manner. The BLISS-based front-end replaces the branch target buffer with a BB-cache that caches the block descriptors in programs. The descriptors provide the front-end with the architectural information necessary for control-flow prediction in a compressed and accurate manner. This allows the decoupled front-end to improve prediction accuracy by judicious use and training of branch predictors, remove the instruction cache latency from the prediction critical path, and accurately prefetch instructions to hide instruction cache misses.

A decoupled front-end with similar advantages can be implemented without the ISA support provided by BLISS [88, 89]. However, its performance is limited by inaccuracies introduced during fetch block creation and by the finite capacity of the fetch-blocks storage. With BLISS, on the other hand, the software defined basic blocks allow the decoupled front-end to avoid the inaccuracies of hardware creation of fetch blocks and to store fetch block information at any level of the memory hierarchy.

Chapter 5

Evaluation for High-End Superscalar Processors

The BLISS ISA provides architectural support for control-flow prediction and instruction delivery. The basic block descriptors convey the software information necessary for accurate branch prediction and guided instruction prefetching. In the previous chapter, we introduced a processor front-end that decouples control-flow speculation from instruction cache accesses. While instruction fetching requires the information in the BBDs, instruction decoding is no longer in the critical path for accurate prediction.

This chapter presents a quantitative evaluation of BLISS for high-end superscalar processors. Section 5.1 presents the methodology and tools. In Section 5.2, we discuss in details the evaluation results for the BLISS-based front-end and compare it to both a conventional front-end design and a hardware-only, decoupled design (FTB). Section 5.3 provides further insights by comparing the BLISS front-end to specific variants of the FTB design. In Section 5.4, we explore the design space for the BLISS front-end by varying key architectural parameters. While this chapter primarily focuses on the performance evaluation for high-end processor cores, Chapter 6 presents the energy evaluation and Chapter 7 evaluates BLISS for embedded processor designs.

5.1 Methodology

5.1.1 Processor Configurations

Our evaluation covers three processor models. The ”Base” model reflects a superscalar processor that uses a RISC ISA. Its front-end relies on a conventional BTB and does not decouple control-flow predictions from instruction fetching. The FTB model uses a decoupled front-end with hardware-only techniques to form extended fetch blocks. It follows the organization discussed in Section 4.2.1. The BLISS model uses a decoupled front-end that utilizes the basic block descriptors in the BLISS ISA. It follows the organization described in Section 4.1.1. We assume that the three models operate at the same clock frequency. For performance, we report instructions committed per cycle (**IPC**) to focus on architectural differences rather than circuit differences.

Table 5.1 presents the microarchitecture parameters used in the evaluation of the three processor models. We simulate both 8-way and 4-way execution cores with all three models. The 8-way execution core is generously configured to reduce back-end stalls so that any front-end performance differences are obvious. The 4-way execution core is loosely modeled after the Alpha 21264 processor [52] and represents a more practical implementation. The three models differ only in the front-end. All of the other parameters are identical. The pipeline of the base model consists of six stages: fetch, decode, issue, execute, writeback, and commit stage. Both of the BLISS and FTB designs have an additional pipe stage. The extra stage in the BLISS design is for fetching BBDs and in the FTB design is for accessing the FTB and pushing fetch-blocks into the FTQ.

In all comparisons, the number of blocks (entries) stored in BTB, FTB, and BB-cache is the same so no architecture has an unfair advantage. Actually, all three structures take approximately the same number of SRAM bits to implement for the same number of entries. The BTB/FTB/BB-cache is always accessed in one cycle. The latency of the other caches in clock cycles is set properly based on its relative size compared to BTB/FTB/BB-cache.

Front-End Parameters			
	Base	**FTB**	**BLISS**
Fetch Width	8 instructions/cycle (4)	1 fetch block/cycle	1 basic block/cycle
Target Predictor	BTB: 2K entries 4-way, 1-cycle access	FTB: 2K entries 4-way, 1-cycle access	BB-cache: 2K entries 4-way, 1-cycle access 8 entries per cache line
Decoupling Queue	–	FTQ: 4 entries	BBQ: 4 entries
Prefetching	–	Based on FTQ	Based on BBQ

Common Processor Parameters	
Prediction	1 prediction per cycle
Hybrid Predictor	gshare: 4K counters PAg L1: 1K entries, PAg L2: 1K counters selector: 4K counters
RAS	32 entries with shadow copy
Instruction cache	32 KBytes, 4-way, 64B blocks, 1 port, 2-cycle access pipelined
Issue/Commit Width	8 instructions/cycle (4)
IQ/RUU/LSQ Size	64/128/128 entries (32/64/64)
FUs	12 INT & 6 FP (6, 3)
Data cache	64 KBytes, 4-way, 64B blocks, 2 ports, 2-cycle access pipelined
L2 cache	1 MByte, 8-way, 128B blocks, 1 port, 12-cycle access, 4-cycle repeat rate
Main memory	100-cycle access

Table 5.1: The microarchitecture parameters used for performance evaluation of high-end superscalar processors. The common parameters apply to all three models (Base, FTB, BLISS). Certain parameters vary between 8-way and 4-way processor configurations. The table shows the values for the 8-way core with the values for the 4-way core in parenthesis.

The same instruction cache and predictors are used in all three models. Both of the FTB and BLISS include an extra queue for decoupling. The size of the BTB/FTB/BB-cache should be approximately 1/4 to 1/8 the size of the instruction cache as each basic block corresponds to 4 to 8 instructions on average (see Section 3.4.2).

For the FTB and BLISS front-ends, we implement instruction prefetching based on the contents of the FTQ and BBQ buffers described in Section 4.1.2. When the instruction cache is stalled due to a miss or because the IQ is full, the contents of FTQ/BBQ entries are used to look up further instructions in the instruction cache. Prefetches are initiated when a

potential miss is identified. The prefetched data goes to a separate prefetch buffer to avoid instruction cache pollution. The simulation results account for contention for the L2-cache bandwidth between prefetches and regular cache misses.

For the case of BLISS, we present results with (*BLISS-Hints*) and without (*BLISS*) the static prediction hints in each basic block descriptor. When available, static hints allow for judicious use of the hybrid predictor. Strongly biased branches do not use the predictor and branches that exhibit strong local or global correlation patterns use only one of its components.

5.1.2 Tools and Benchmarks

Our simulation framework is based on the Simplescalar/PISA 3.0 toolset [16], which we modified to add the FTB and BLISS front-end models. For energy measurements, we use the Wattch framework with the cc3 power model [15]. In this non-ideal, aggressive conditional clocking model, power is scaled linearly with port or unit usage, except that unused units dissipate 10% of their maximum power, rather than drawing zero power. Energy consumption was calculated for a 0.10μm process with a 1.1V power supply. The reported *Total Energy* includes all the processor components (front-end, execution core, and all caches). Access times for cache structures were calculated using Cacti v3.2 [102].

For our experimental evaluation, we study 12 benchmarks from the SPEC CPU2000 suite using their reference datasets [38]. The selected benchmarks have varying requirements on the front-end. The rest of the benchmarks in the SPEC CPU2000 suite perform similar and their results are not shown for brevity. For benchmarks with multiple datasets, we run all of them and calculate the average. The benchmarks are compiled with gcc at the -O3 optimization level. With all benchmarks and all front-ends, we skip the first billion instructions in each dataset and simulate another billion instructions for detailed analysis. The BLISS code is generated using the process explained in Section 3.3.

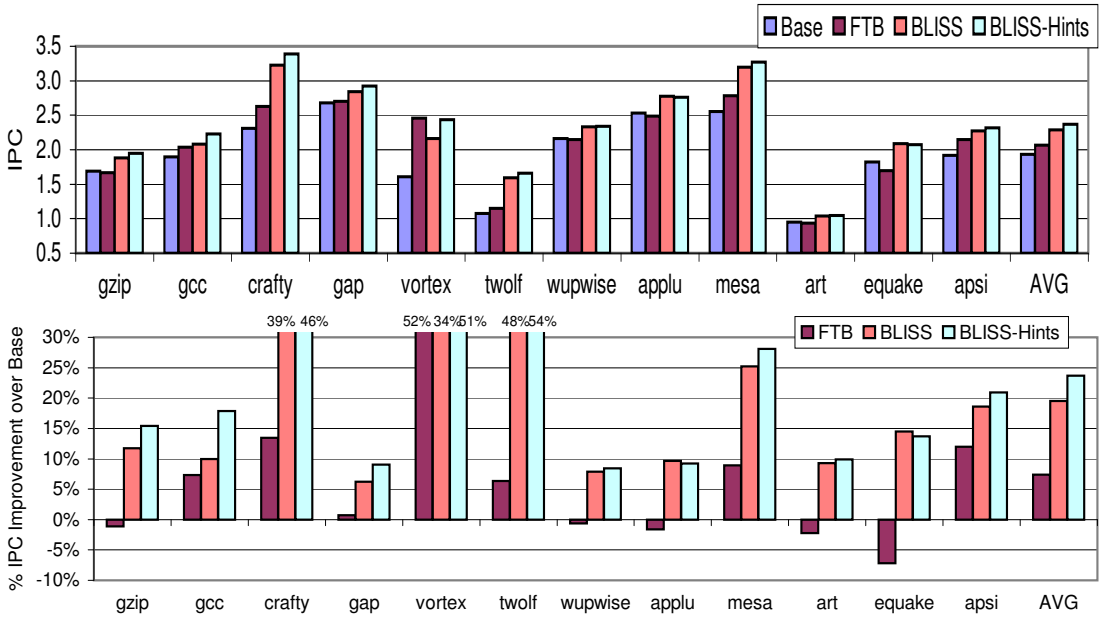


Figure 5.1: Performance comparison for the 8-way processor configuration with the Base, FTB, and BLISS front-ends. The top graph presents raw IPC and the bottom one shows the percentage of IPC improvement over the Base for FTB and BLISS.

5.2 Evaluation

This section presents the performance comparison between the base, FTB, and BLISS models. To illustrate the source of the performance differences, we also study in details prediction accuracy issues and the behavior of target prediction tables, instruction cache, L2-cache, and instruction prefetcher for each model.

5.2.1 Performance Evaluation

Figure 5.1 compares the IPC achieved for the 8-way superscalar processor configuration with the three front-ends. The graphs present both raw IPC and percentage of IPC improvement over the base front-end. The FTB design provides a 7% average IPC improvement over the base, while the BLISS front-end allows for 20% and 24% improvement over the

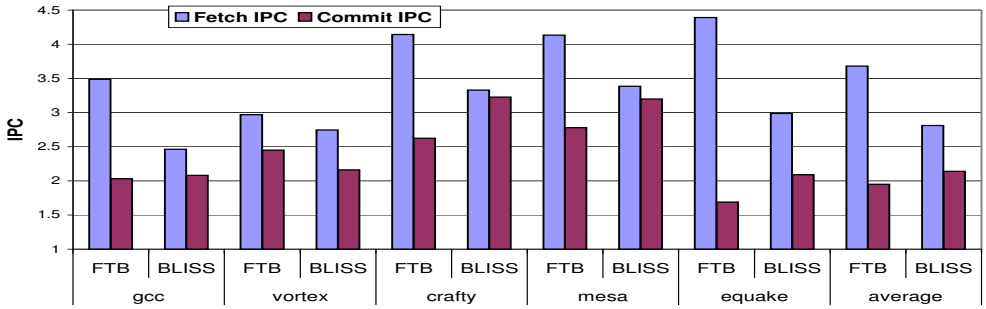


Figure 5.2: Fetch and commit IPC for the 8-way processor configuration with the FTB and BLISS front-ends. We present data for a representative subset of benchmarks, but the average refers to all benchmarks in this study. For BLISS, we present the data for the case without static branch hints.

base without and with branch hints respectively. The FTB front-end provides IPC improvements over the base for 7 out of 12 benchmarks, while for the remaining benchmarks there is either no benefit or a small slowdown. On the other hand, the BLISS front-end improvement over the base is consistent across all benchmarks. Even without static hints, BLISS outperforms FTB for all benchmarks except `vortex`. For `vortex`, the FTB front-end is capable of forming long fetch blocks which helps in achieving a higher FTB hit rate (see Figure 5.4). With hints, BLISS achieves even higher IPC improvements. This is mainly due to the improved prediction accuracy, as we will see in Section 5.2.2.

Both FTB and BLISS have fundamental advantages over the base due to their decoupled front-end (see Section 4.1.1 and 4.2.1). The FTB design is more aggressive than BLISS in terms of instruction fetching. Nevertheless, overly aggressive instruction fetch may hurt overall performance due to the cost of misfetches and mispredictions. To illustrate this issue, Figure 5.2 compares the fetch and commit IPC for the FTB and BLISS front-ends. The fetch IPC is defined as the average number of instructions described by the blocks inserted in the FTQ/BBQ in each cycle. Looking at fetch IPC, the FTB design fetches more instructions per cycle than BLISS (3.6 versus 2.8 on the average). The FTB advantage is due to the larger blocks and because the front-end generates fall-through blocks on FTB misses, while the BLISS front-end stalls on BB-cache misses and retrieves the descriptors

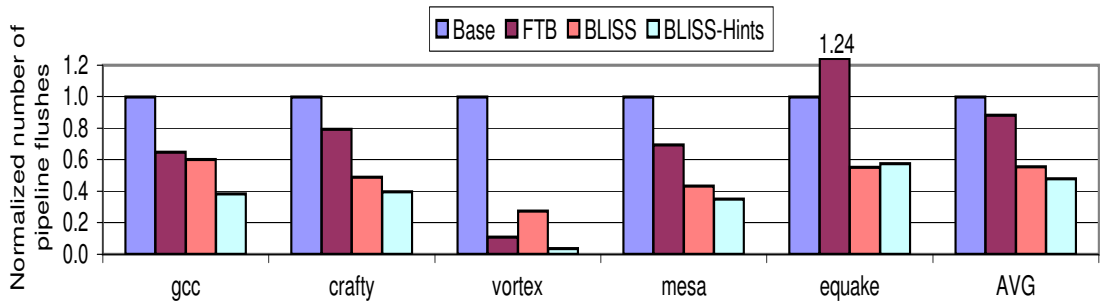


Figure 5.3: Normalized number of pipeline flushes due to direction and target mispredictions for the 8-way processor configuration with the Base, FTB, BLISS, and BLISS-HINTS front-ends. We present data for a representative subset of benchmarks, but the average refers to all benchmarks in this study.

from the L2-cache. Nevertheless, in terms of commit IPC (instructions retired per cycle), the BLISS front-end has an advantage (1.9 versus 2.1). In other words, a higher ratio of instructions predicted by the BLISS front-end turn out to be useful. The long, fall-through fetch blocks introduced on FTB misses contain large numbers of erroneous instructions that lead to misfetches, mispredictions, and slow predictor training. On the other hand, the BB-cache in BLISS always retrieves an accurate descriptor from the L2-cache. In other words, BLISS uses the information available through the instruction set to strike a balance between over and under-speculation.

5.2.2 Prediction Accuracy Analysis

To understand the performance difference between the FTB and BLISS designs, we are going to look first at the prediction accuracy and in the following sections at the instruction cache, L2-cache, and the FTB/BB-cache behaviors.

Figure 5.3 quantifies the differences in prediction accuracy for the three designs by comparing the number of full pipeline flushes. A full pipeline flush occurs when a branch is executed and either its target or direction is mispredicted. In such a case, all of the instructions following the branch are removed from the pipeline and fetching starts at the

correct address. These flushes have a severe performance impact as they empty the full processor pipeline. Compared to the base, BLISS reduces by 41% the number of pipeline flushes due to target and direction mispredictions. Flushes in BLISS are slightly more expensive than in the base design due to the longer pipeline, but they are less frequent. The BLISS advantage is due to the availability of control-flow information from the BB-cache regardless of instruction cache latency and the accurate indexing and judicious use of the hybrid predictor. When static branch hints are in use (*BLISS-Hints*), the branch prediction accuracy is improved by an average of 1.2% from 93.4% without hints to 94.6% with hints. The improved prediction accuracy results in an additional 10% reduction in the number of pipeline flushes.

The FTB front-end also reduces by 17% the number of pipeline flushes due to target and direction mispredictions. However, it has significantly higher number of pipeline flushes compared to the BLISS front-end as dynamic block recreation affects the prediction accuracy of the hybrid predictor due to longer training and increased interference. The FTB design also suffers from partial (front-end) pipeline flushes due to misfetches when the decoding logic detects a jump in the middle of a fetch block. The number of misfetches can be used to quantify the effectiveness of FTB in delivering fetch blocks. In the next section, we will discuss this in more details.

5.2.3 FTB and BB-Cache Analysis

Figure 5.4 evaluates the effectiveness of the BB-cache in delivering BBDs and the FTB in forming fetch blocks by comparing their hit rates. Since the FTB returns a fall-through block address even when it misses to avoid storing the fall-through blocks, we define the FTB miss rate as the number of misfetches divided over the number of FTB accesses. A misfetch occurs when the decoding logic detects a jump in the middle of a fetch block. At the same storage capacity, the BLISS BB-cache achieves a 2% to 3% higher hit rate than the

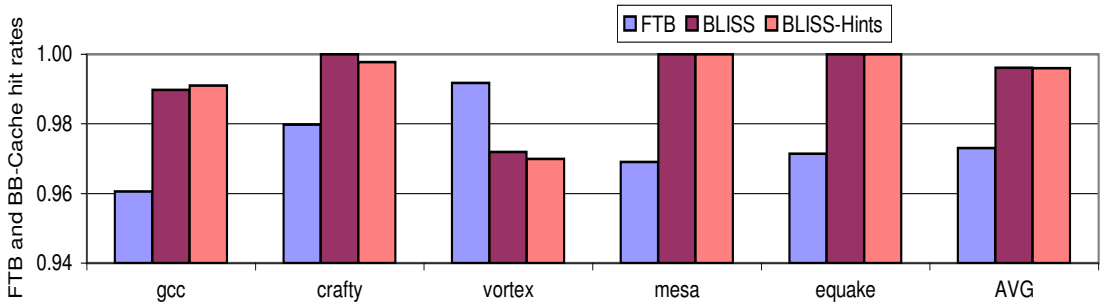


Figure 5.4: FTB and BB-cache hit rates for the 8-way processor configuration. We present data for a representative subset of benchmarks, but the average refers to all benchmarks in this study.

FTB as the BB-cache avoids block splitting and recreation that occur when branches change behavior or when the cache capacity cannot capture the working set of the benchmark.

The FTB has an advantage for programs like `vortex` that stress the capacity of the target cache and include large fetch blocks. For `vortex`, the FTB packs 9.5 instructions per entry (multiple basic blocks), while the BB-cache packs 5.5 instructions per entry (single basic block). Even though BLISS may not be able to merge basic blocks on biased branches, its fetch rate is higher than the consumption rate of the back-end. As long as this is the case, being more aggressive in creating large fetch blocks does not improve performance. It may even hurt performance due to frequent misspeculations (see Figure 5.2). It is also interesting to note that the BB-cache miss rates for BLISS with and without hints are almost identical. A BB-cache hit or miss is independent from whether the prediction provided is accurate. The small difference between the two is due to the slightly different control-flow paths followed by the two due to differences in prediction.

5.2.4 Instruction Cache Analysis

Figure 5.5 compares the normalized number of instruction cache accesses and misses for the FTB and BLISS front-ends over the base design. Although both of the FTB and BLISS

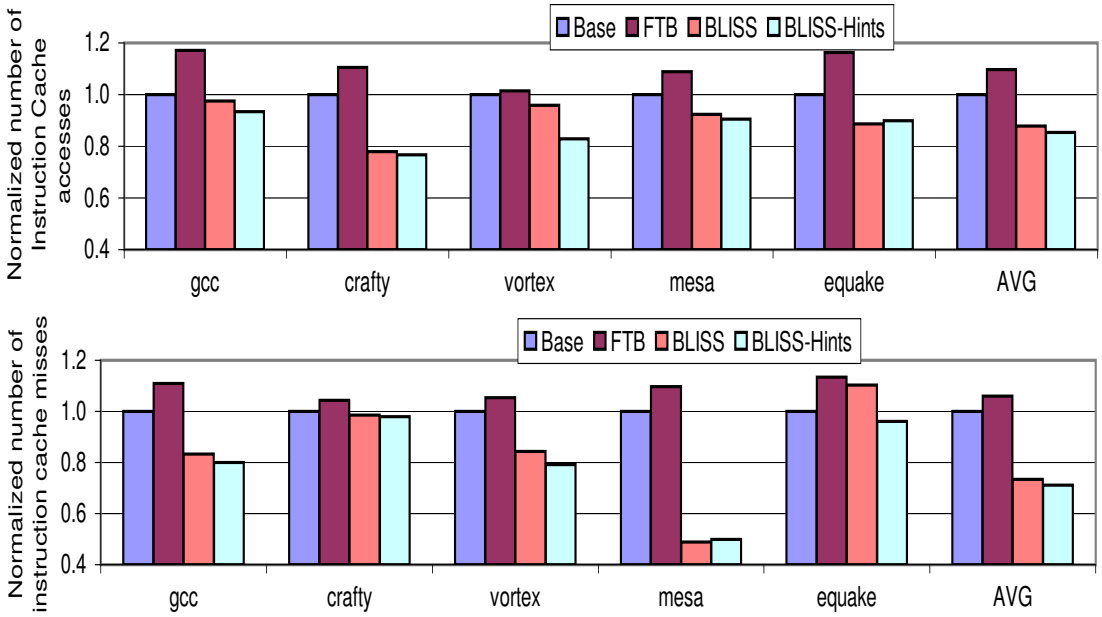


Figure 5.5: Instruction cache comparison for the 8-way processor configuration with the Base, FTB, BLISS, and BLISS-HINTS front-ends. The top graph compares the normalized number of instruction cache accesses and the bottom one shows the normalized number of instruction cache misses. We present data for a representative subset of benchmarks, but the average refers to all benchmarks in this study.

designs enable prefetching based on the contents of the decoupling queue, the BLISS design has fewer instruction cache misses and accesses. The BLISS advantage is due to the more accurate prediction as shown in Figure 5.3 and the reduced number of instructions by the basic optimizations. The BLISS front-end has 12% fewer instruction cache accesses and 27% fewer misses compared to the base design. Even with prefetching and accurate prediction, the FTB front-end has 10% higher number of instruction cache accesses and 6% higher number of misses compared to the base design. The increase is a result of the maximum length fetch blocks that are inserted in the FTQ after an FTB miss. This difference would even be higher for a front-end with smaller instruction cache. Note that a higher number of instruction cache accesses or misses also has direct impact on the energy consumption of the front-end.

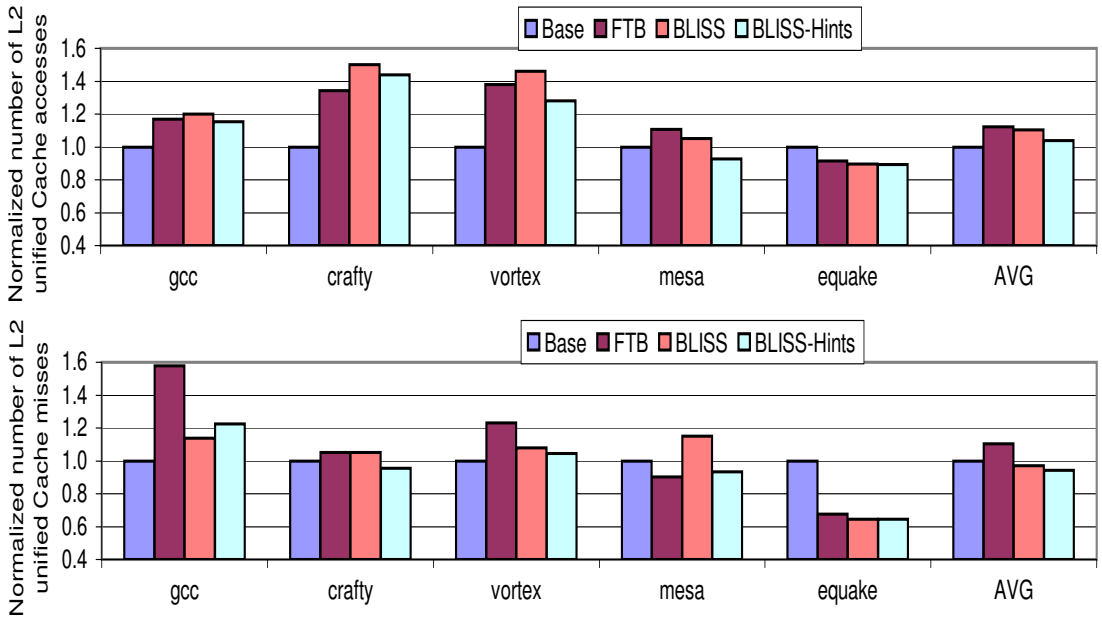


Figure 5.6: L2-cache comparison for the 8-way processor configuration with the Base, FTB, BLISS, and BLISS-HINTS front-ends. The top graph compares the normalized number of L2-cache accesses and the bottom one shows the normalized number of L2-cache misses. We present data for a representative subset of benchmarks, but the average refers to all benchmarks in this study.

5.2.5 L2-Cache Analysis

Figure 5.6 compares the normalized number of the L2-cache accesses and misses for the FTB and BLISS front-ends. The total number of L2-cache accesses includes accesses from the instruction cache and data cache. For BLISS, it also includes accesses from the BB-cache and the prefetcher. As expected, both of the FTB and BLISS front-ends have a higher number of L2-cache accesses due to prefetching. Although the BLISS L2-cache serves the BB-cache misses in addition to the instruction cache and data cache misses, the number of L2-cache accesses and misses are slightly better than the numbers for the FTB design. The BLISS design has a 10% higher average number of L2-cache accesses than the base design, while the FTB design has a 12% higher average. FTB also exhibits a 10% higher L2-cache misses; this is due to the large number of erroneous instructions that are fetched

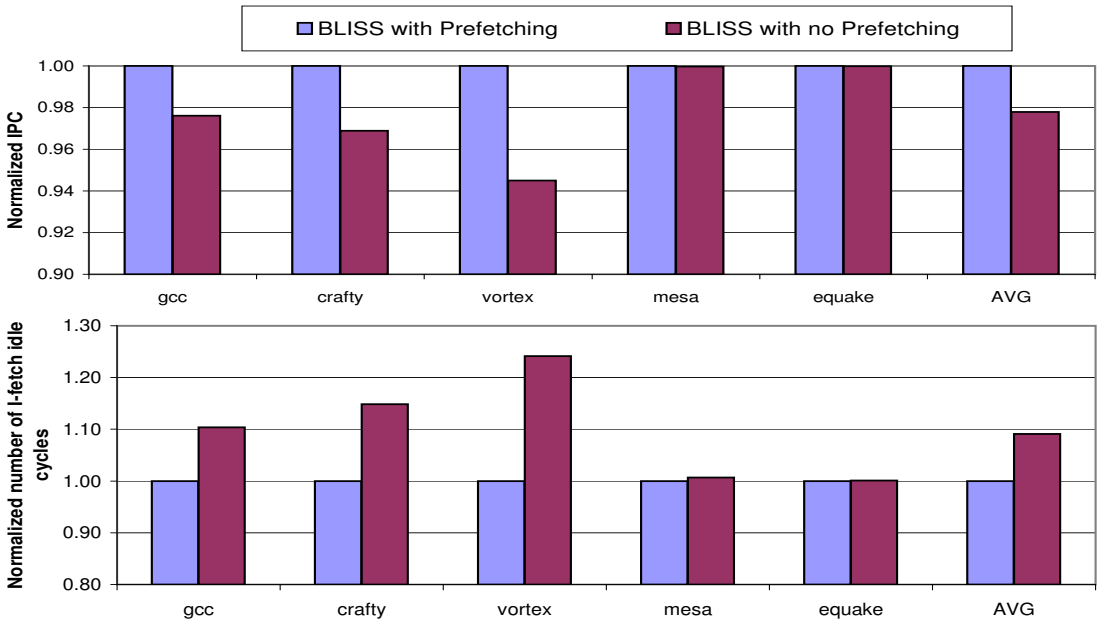


Figure 5.7: Impact of instruction prefetching for the 8-way processor configuration with the BLISS front-end. The top graph presents the normalized IPC with no prefetching. The bottom graph presents the normalized number of cycles the instruction fetch unit is idle due to instruction cache misses with no prefetching.

by the FTB front-end while it is forming the fetch blocks dynamically. BLISS is slightly better than the base design with a 3% fewer L2-cache misses. The BLISS advantage is mainly due to the better and accurate prediction as shown in Figure 5.3.

5.2.6 Instruction Prefetching Analysis

Figure 5.7 quantifies the performance impact when prefetching is not enabled for the BLISS front-end. The top graph in Figure 5.7 presents the normalized IPC for BLISS with no prefetching. On average, performance degrades by 2.5% when prefetching is not enabled. Most of the loss is from INT benchmarks as they tend to have large instruction footprints and somewhat irregular code access patterns. For FP benchmarks, the IPC degradation is negligible as those applications are typically dominated by tight code loops with high temporal locality. The bottom graph in Figure 5.7 compares the normalized number of cycles

the instruction fetch unit is idle due to instruction cache misses for the BLISS front-end when prefetching is not enabled. On average, prefetching reduces by 10% the number of cycles the instruction fetch unit is stalling due to instruction cache misses while instructions are being retrieved from lower memory hierarchy. This number would even be higher for a front-end with smaller instruction cache.

5.3 Detailed Comparison to FTB Variants

In Section 5.2, we demonstrated that a BLISS-based front-end outperforms the hardware-based FTB design described in Section 4.2.1. To further understand the differences between hardware-only and software-assisted basic block formation, this section compares BLISS to two FTB variants that attempt to reduce its sensitivity to over-speculation in block formation.

In the first design, biased not-taken branches are not embedded in fetch blocks. Therefore, any branch in the middle of a fetch block terminates the block and leads to a misfetch when first decoded. This design is essentially BLISS implemented fully in hardware with no software support and no block coalescing. We refer to this less aggressive FTB design as FTB-simple. This design fixes couple of the problems with the original FTB design. First, the branch predictor is accurately trained as fetch blocks are consistent over time and are not shortened when branches change behavior. Second, all branches are predicted by the hybrid predictor, eliminating the implicit not-taken prediction for the branches embedded in the middle of the fetch blocks. This eliminates mispredictions and pipeline flushes associated with those embedded branches when there is a conflict with the branch predictor itself.

Nevertheless, the advantages of FTB-simple come at an additional cost. First, the increased number of misfetches caused by detecting biased not-taken branches in the middle of fetch blocks may have a negative impact on performance. In addition, the formed fetch blocks are smaller than the blocks created in the original FTB design as biased not-taken

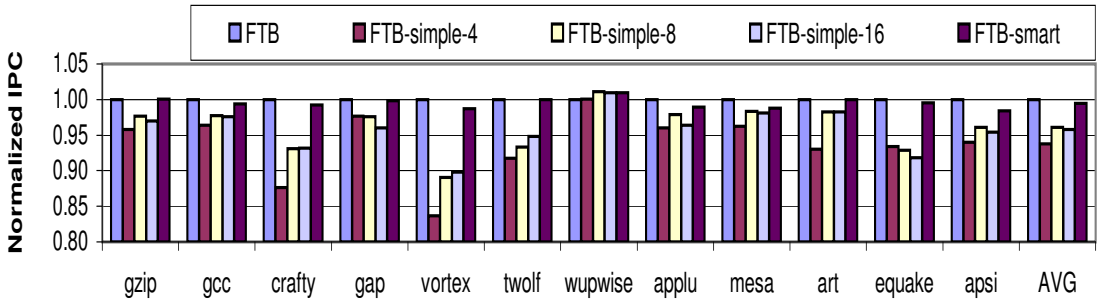


Figure 5.8: Normalized IPC for the FTB-simple and FTB-smart designs over the original FTB design for the 8-way processor configuration.

branches are no longer embedded in the fetch blocks. This increases the contention for the finite capacity of the FTB and reduces the fetch bandwidth. Finally, the hybrid predictor is now also used for the biased not-taken branches which may lead to different interference patterns with other branches.

The second FTB design allows for embedded branches in the middle of fetch blocks only if their prediction is not-taken. If a branch in the middle of the block is predicted taken, the decode stage will issue a misfetch. In this case, the pipeline stages behind decoding are flushed and fetching restarts at the branch target. We refer to this design as FTB-smart. The advantage of this design over the original FTB design is that some possible mispredictions caused by the default not-taken policy on an FTB miss are converted to misfetches which are resolved in the decode stage. Compared to BLISS, the FTB-smart design allows for block coalescing on biased not-taken branches. The disadvantage of this design is that it relies on the predictor to form the fetch blocks. If the prediction is not accurate, then additional misfetches will occur and extra fetch blocks may increase the contention for the finite capacity of the FTB.

Figure 5.8 compares the normalized IPC for the FTB-simple and FTB-smart over the original FTB design. For FTB-simple, we present data with different fetch-block lengths: 4, 8, and 16 instructions. The FTB-smart design uses a fetch-block length of 16 instructions similar to the original FTB design. Figure 5.9 compares the normalized number of

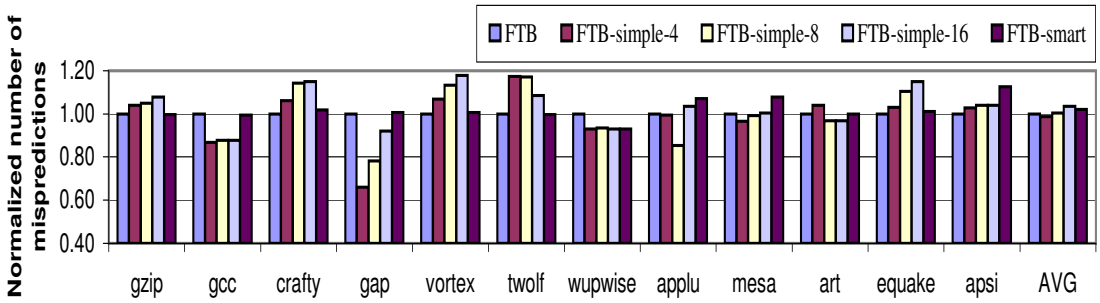


Figure 5.9: Normalized number of mispredictions for the FTB-simple and FTB-smart designs over the original FTB design for the 8-way processor configuration.

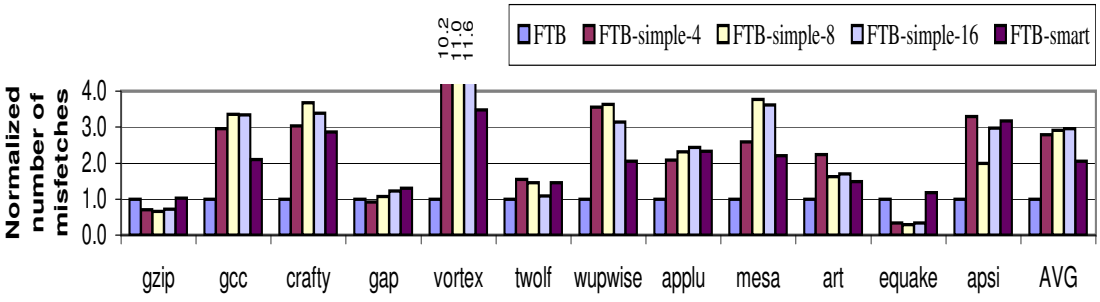


Figure 5.10: Normalized number of misfetches for the FTB-simple and FTB-smart designs over the original FTB design for the 8-way processor configuration.

mispredictions and Figure 5.10 compares the normalized number of misfetches for the FTB-simple and FTB-smart designs over the original FTB design.

For all of the FTB-simple configurations, we see consistent performance degradation. This is mainly due to the significant increase in the number of misfetches for FTB-simple compared to the original design. The small improvement in the prediction accuracy is not enough to compensate the significant increase in the number of misfetches. Even though the number of misfetches slightly decreases with smaller fetch-blocks, the performance degrades as the average length of fetch-blocks committed decreases from 8.3 instructions in the original FTB design to 7.5, 5.4, and 3.4 for FTB-simple-16, FTB-simple-8, and FTB-simple-4 respectively. Note that programs like `vortex` which benefits the most from block coalescing are the worst performer with the FTB-simple design.

For FTB-smart, we get also a consistent increase in the number of misfetches. However, the increase is less dramatic compared to the FTB-simple design. In terms of IPC and number of mispredictions, we see a slight change with averages close to the original FTB design. The average fetch-block length for FTB-smart also slightly decreases from 8.3 in the original FTB design to 8.2 instructions per fetched blocks.

Overall, the original FTB design outperforms all of the other FTB configurations. The BLISS-based design outperforms the original FTB design as it balances over- and under-speculation with better utilization of the BB-cache capacity. BLISS attempts to strike a balance between hardware and software features that optimize the critical metric for front-end engines: useful instructions predicted per cycle.

5.4 Sensitivity Analysis

Our analysis thus far has used a single configuration for both the front-end and the back-end of the processor. In this section, we validate our conclusions by examining the sensitivity of the BLISS and FTB models to key architectural parameters such as the target prediction tables, instruction cache, and issue width. In all configurations, the FTB and the BB-cache are always accessed in one cycle. The latency of the instruction cache in clock cycles is set properly based on its relative size compared to the FTB or BB-cache.

5.4.1 Sensitivity to BB-Cache and FTB Parameters

The performance with both decoupled front-ends depends heavily on the miss rate of the FTB and BB-cache respectively. As we showed in Figure 5.4, the high BB-cache miss rate for `vortex` leads to a performance advantage for the FTB design which is able to pack more instructions per FTB entry for this benchmark. Figure 5.11 presents the average IPC across all benchmarks for the 8-way processor configuration with the FTB and BLISS

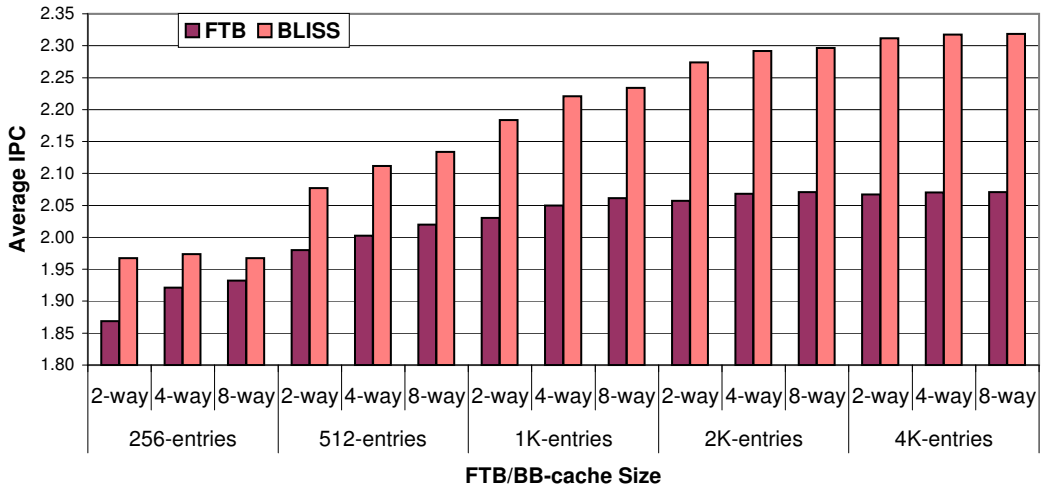


Figure 5.11: Average IPC for the 8-way processor configuration with the FTB and BLISS front-ends as we scale the size and associativity of the FTB and BB-cache structures. For the BLISS front-end, we assume that static prediction hints are not available in this case.

front-ends as we scale the size (number of entries) and associativity of the FTB and BB-cache structures. The BB-cache is organized with 8 entries per cache lines in all cases.

Figure 5.11 shows that for all sizes and associativities the BLISS front-end outperforms FTB. The performance for both front-ends improves with larger sizes up until 2K entries which are sufficient to capture the working set of basic blocks or fetch blocks for most programs. The performance difference does not change significantly across different sizes. The increasing number of entries eliminates stalls due to BB-cache misses for BLISS and reduces the inaccuracies introduced by fetch block recreation due to FTB misses in the FTB design. Associativity is less critical for both front-ends. With 512 or 1K entries, 4-way associativity is preferred but with a larger FTB or BB-cache, 2-way is sufficient. Note that with 256 entries, the FTB is more sensitive to the associativity than the BLISS front-end.

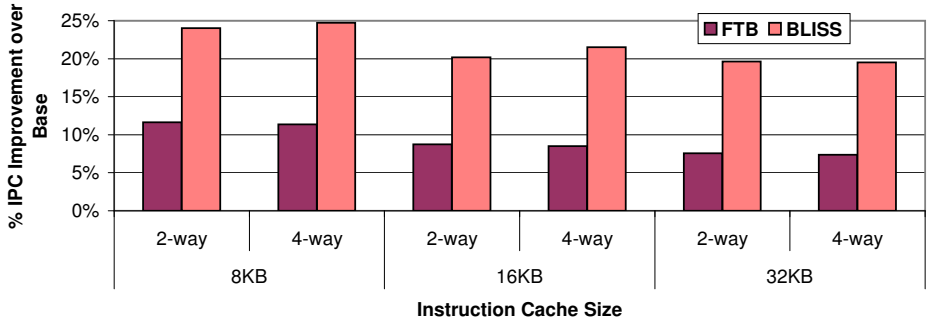


Figure 5.12: Average percentage of IPC improvement with the FTB and BLISS front-ends over the base design as we vary the size and associativity of the instruction cache. We simulate an 8-way execution core, 2-cycle instruction cache latency, and 2K entries in the BTB, FTB, and BB-cache respectively.

5.4.2 Sensitivity to Instruction Cache Size

The use of a small instruction cache was one of the motivations for the FTB and the BLISS front-ends. The reason is that smaller instruction caches have lower access latency and lower power and energy consumption. The instruction prefetching enabled by the FTQ and BBQ can compensate for the increased miss rate of a small instruction cache.

Figure 5.12 shows the IPC improvement with the FTB and BLISS front-ends over the base design as we vary the size and associativity of the instruction cache in the 8-way processor configuration. Note that the instruction cache size of the base design is varied along with the size used for the FTB and BLISS designs. Both decoupled front-ends provide IPC advantages over the baseline for all instruction cache sizes. However, the IPC improvement drops as the instruction cache size grows to 32 KBytes (from 12% to 7% for FTB, from 24% to 20% for BLISS). With a larger cache size, the instruction cache incurs less number of misses and thus the benefit of prefetching is less significant. The BLISS front-end maintains a 13% IPC lead over the FTB design for all instruction cache sizes and associativities.

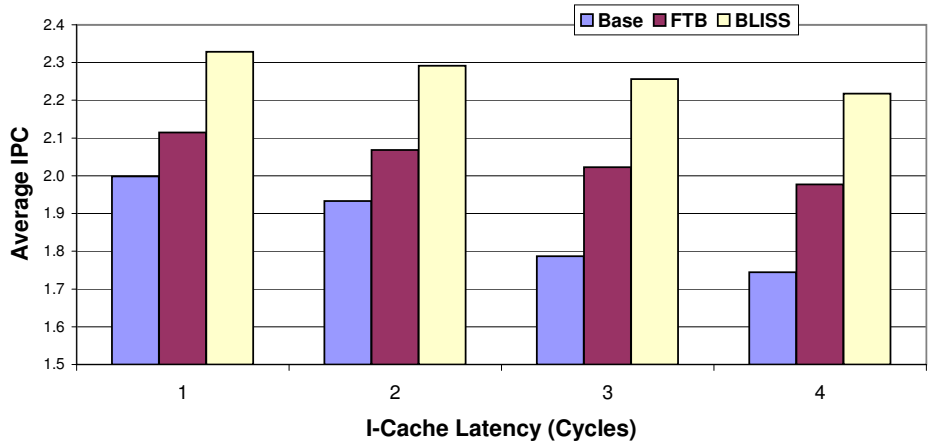


Figure 5.13: Average IPC with the Base, FTB, and BLISS front-ends as we vary the latency of the instruction cache from 1 to 4 cycles. We simulate an 8-way execution core, 32 KByte pipelined instruction cache, and 2K entries in the BTB, FTB, and BB-cache respectively.

5.4.3 Sensitivity to Instruction Cache Latency

Another advantage of a decoupled front-end is the ability to tolerate higher instruction cache latencies. The information in the FTB and the BB-cache allow for one control-flow prediction per cycle even if it takes several cycles to fetch and decode the corresponding instructions in order to locate the actual control-flow instructions in the stream. Tolerance to high instruction cache latencies can be useful with decreasing the instruction cache area and power for a fixed capacity or with allowing for a larger instruction cache within a certain area, power budget, and clock frequency. Larger instruction caches are desirable for enterprise applications that tend to have larger instruction footprints [9].

Figure 5.13 presents the average IPC for the 8-way processor configuration with the base, FTB, and BLISS front-ends as we scale the instruction cache latency from 1 to 4 cycles. For the base design, IPC decreases 13% as we scale the instruction cache latency from 1 to 4 cycles. With both the FTB and BLISS front-ends, IPC decreases approximately 5% between the two end points, which shows good tolerance to instruction cache latency. The performance loss is mainly due to the higher cost of recovery from mispredictions and

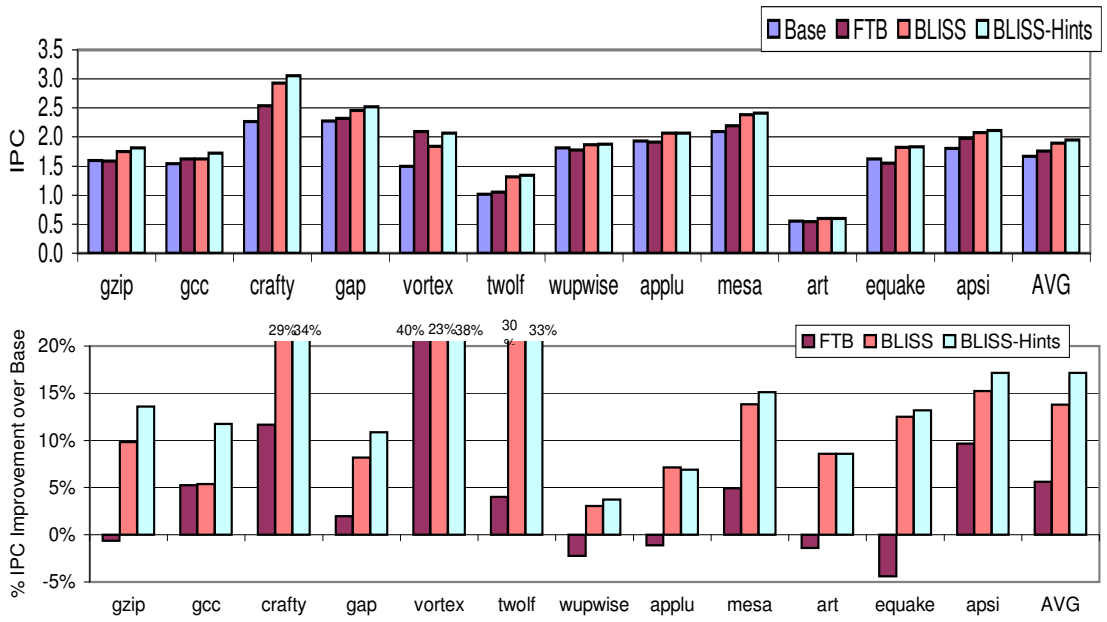


Figure 5.14: Performance comparison for the 4-way processor configuration with the Base, FTB, and BLISS front-ends. The top graph presents raw IPC and the bottom one shows the percentage of IPC improvement over the Base for FTB and BLISS.

misfetches. The actual number of mispredictions and misfetches does not scale up significantly, which validates the decoupled front-end approach. BLISS outperforms FTB across all latency values, which validates that architectural support for basic block descriptors is superior to hardware-based block creation on top of a conventional instruction set.

5.4.4 4-way Processor Analysis

The 8-way processor configuration analyzed in Section 5.2 represents an aggressive design point, where the execution core is designed for minimum number of back-end stalls. Figure 5.14 shows the impact of the front-end selection on the 4-way execution core configuration described in Table 5.1, which represents a practical commercial implementation.

Figure 5.14 shows that the performance comparison with the 4-way execution core is nearly identical to that with the 8-way core. FTB provides a 6% performance advantage

over the base design, while BLISS allows for 14% or 17% IPC improvements without and with the static hints respectively. The absolute values for the improvements are lower than with the 8-way core due to the additional stalls in the execution core that mask performance challenges in the front-end.

In Chapter 7, we are going also to provide additional evidence that demonstrates the advantage of the BLISS design by evaluating it for embedded processor designs.

5.5 Summary

In this chapter, we performed a detailed evaluation of BLISS for high-end superscalar processors. Compared to conventional superscalar processors, BLISS allows for highly accurate control-flow speculation and instruction delivery which leads to 20% IPC advantage. We illustrated the usefulness of the hints mechanism in BLISS by using it to implement branch prediction hints and showed that it further improves the performance by 4%. We also compared BLISS to a comprehensive decoupled front-end that dynamically builds fetch blocks in hardware and demonstrated that the BLISS-based front-end achieves 13% performance improvement over it. Unlike techniques that rely solely on larger and more complex hardware structures, our proposal attempts to strike a balance between hardware and software features that optimize the critical metric for front-end engines: useful instructions predicted per cycle. Finally, we showed that the BLISS benefits are robust across a wide range of architectural parameters for superscalar processors.

While this chapter focused primarily on performance, next chapter evaluates BLISS from the energy perspective and demonstrates that the BLISS based front-end improves energy on top of the performance benefits.

Chapter 6

Energy Optimizations

Modern high-end processors must provide high application performance in an energy effective manner. Energy efficiency is essential for dense server systems (e.g. blades), where thousands of processors may be packed in a single colocation site. High energy consumption can severely limit the server scalability, its operational cost, and its reliability [28]. Furthermore, an energy-efficient high performance design allows semiconductor vendors to use the same processor core in chips for both server and notebook applications. For notebooks, energy consumption is directly related to battery life.

The instruction fetch mechanism largely influences the energy behavior for superscalar processors [93]. The front-end determines how often the processor is executing useful instructions, mispredicted instructions, or no instructions at all. After an instruction cache miss, the front-end stalls and wastes leakage energy until the required instructions are retrieved from lower memory hierarchy. Similarly, when the front-end misspeculates, energy is wasted by fetching and executing erroneous instructions from the wrong execution path. The front-end itself also consumes a significant percentage of the processor total energy as it contains large memory arrays (instruction cache, predictor, BTB) that are accessed nearly every cycle. On average, 13% of the total energy is consumed in the front-end itself alone for a 4-way superscalar processor [115].

This chapter presents and evaluates the energy optimizations with the BLISS-based front-end. Section 6.1 explains how the BLISS front-end reduces the wasted energy in the overall design (front-end and back-end). In Section 6.2, we discuss the optimizations that reduce the energy consumption in the front-end. Section 6.3 explains the methodology used for evaluation. In Section 6.4, we present and analyze the evaluation results. Section 6.5 highlights related work.

6.1 Reducing Wasted Energy

In Section 2.3, we reviewed how the front-end detractors affect energy consumption. We showed that 14% and 7% of the processor total energy are wasted by branch direction and target mispredictions for INT and FP applications respectively. We also showed that an additional 12% and 6% of the total energy are wasted on instruction cache detractors for INT and FP applications respectively.

The performance enhancements by BLISS address the sources of wasted energy as well. We demonstrated in Figure 5.2 that BLISS better utilizes the processor resources by sustaining high instruction throughput as a result of guided prefetching and accurate branch prediction. This eliminates cycles where the back-end is idle wasting leakage energy. We also demonstrated in Figure 5.3 that BLISS improves prediction accuracy and significantly reduces the number of expensive pipeline flushes. This allows BLISS to reduce the energy wasted on fetching and executing mispredicted instructions. It also eliminates the overhead of recovering the pipeline from branch mispredictions when the erroneous instructions are removed from the pipeline. By reducing execution time, the BLISS-based design also saves on the energy consumed by the clock tree and the processor resources even when they are stalling or idling. Overall, the performance benefits of BLISS translates directly to energy benefits as well as it avoids paying leakage energy wasted on idle resources and reduces dynamic energy wasted on fetching and executing mispredicted instructions.

The FTB design also reduces overall wasted energy as it improves instruction bandwidth and reduces the number of pipeline flushes. However, as we saw in Figure 5.2 and Figure 5.3, BLISS achieves higher instruction throughput and has fewer pipeline flushes compared to the FTB design. This allows BLISS to compare favorably to the FTB design in terms of energy efficiency in addition to performance.

6.2 Energy Efficient Front-End

In this section, we show how the BLISS-based front-end facilitates several energy optimizations in the predictor and instruction cache. The optimizations target both the frequency of accesses to each front-end resource and the energy consumed per access for each resource.

6.2.1 Predictor Optimizations

BLISS reduces significantly the number of accesses to the branch predictors. Compared to the conventional design, BLISS reduces by 61% (54% for INT and 71% for FP applications) the number of accesses to the predictor tables. This is mainly due to two reasons. First, the basic block decoupling queue (BBQ) allows for control-flow predictions at the rate of one block per cycle. As we demonstrated in Section 3.4, the SPEC CPU benchmarks have an average of 7.7 instructions per executed basic block (5.8 for integer, 9.7 for floating-point). With more than 50% of the executed basic blocks have more than 4 instructions, making one prediction for every 4 instructions fetched from the instruction cache is wasteful. With BLISS, only one prediction is required for each basic block. This saves energy in accessing and training the predictors. Note that the average reduction in the number of accesses to the prediction tables for the FP applications is higher than the average for the INT applications as FP applications have longer basic block sizes. Second, the type of the BBD allows us to selectively access and train the predictor in an accurate manner. For BBDs that correspond to fall-through blocks or blocks terminating

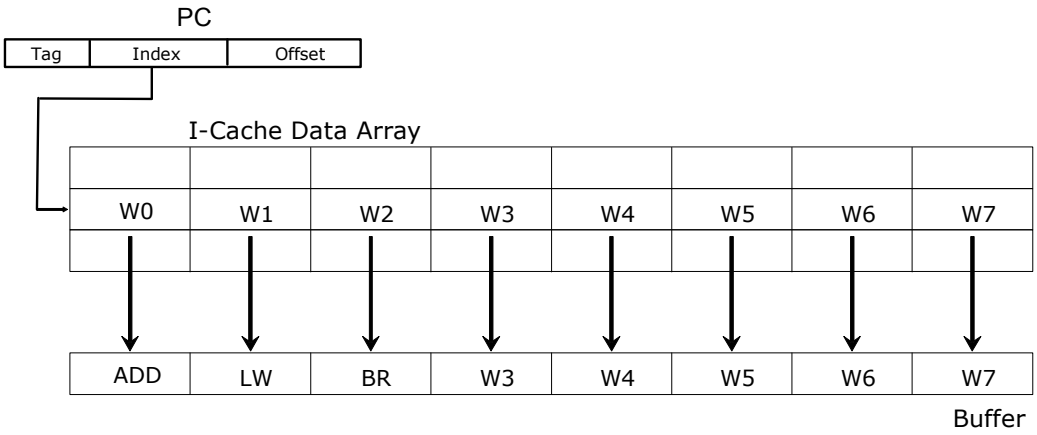
with an unconditional control-flow instruction (J, JAL, etc), the predictor is not used nor trained.

BLISS also reduces the average energy consumed per predictor access. Decoupling allows for the direction prediction to be verified with a large, tagless, hybrid predictor, while the fetch block is waiting in the BBQ [105]. Relaxing the predictor access time allows the design of the predictor to be optimized for energy efficiency. Moreover, the static hints when available, allow for judicious use of the hybrid predictor. Once the hints are available for a BBD, the front-end can determine which is the best predictor to consult and update for the associated control-flow instruction without accessing or training the selector. Strongly biased branches do not use the predictor table at all and branches that exhibit strong local or global correlation patterns use only one of its components. This saves 48% of the energy consumed in the hybrid predictor.

6.2.2 Instruction Cache Optimizations

BLISS reduces the number of accesses to the instruction cache. We can merge the instruction accesses for sequential blocks in the BBQ that hit in the same cache line, in order to save decoding and tag access energy. Moreover, BLISS reduces the energy consumed per instruction cache access as it facilitates energy optimizations in the instruction cache through selective way/word access and serial access to data and tags without sacrificing performance.

Reading the complete cache line on instruction fetch is not energy efficient as only a subset of the instructions in the line are typically used (e.g. 4 out of 16). A jump or a taken branch in the middle of the cache line would possibly force the front-end to start fetching from a different cache line. Instructions in the early part of a cache line, right before the target of a taken branch, are also less likely to be required. With BLISS, each basic block descriptor defines exactly the number of instructions needed from the instruction cache and



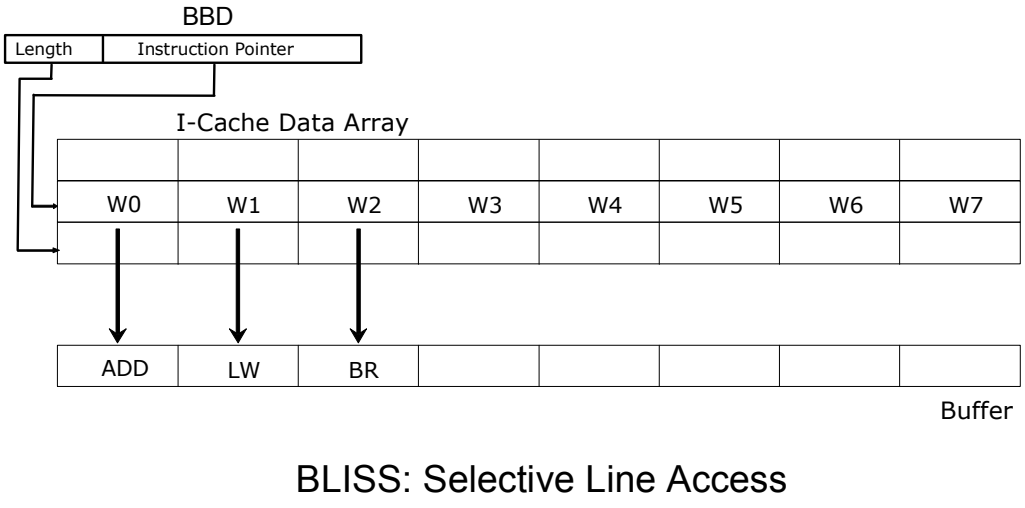
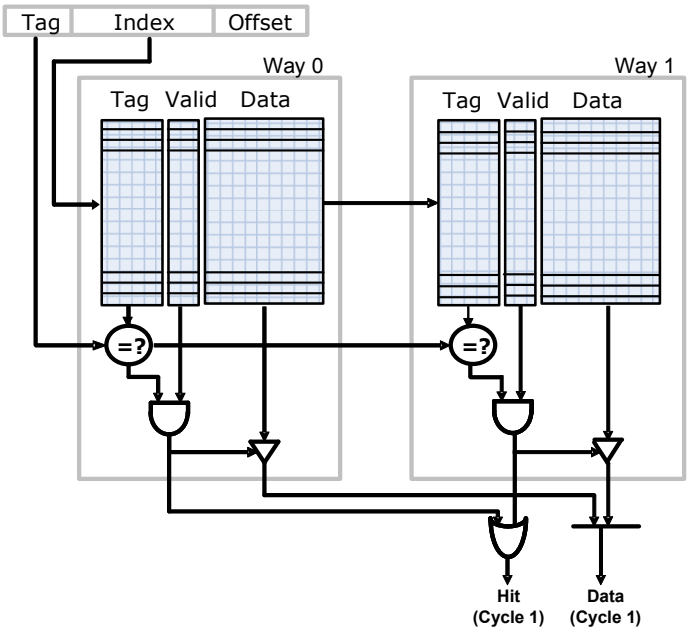


Figure 6.1: An example to illustrate selective word access with BLISS.

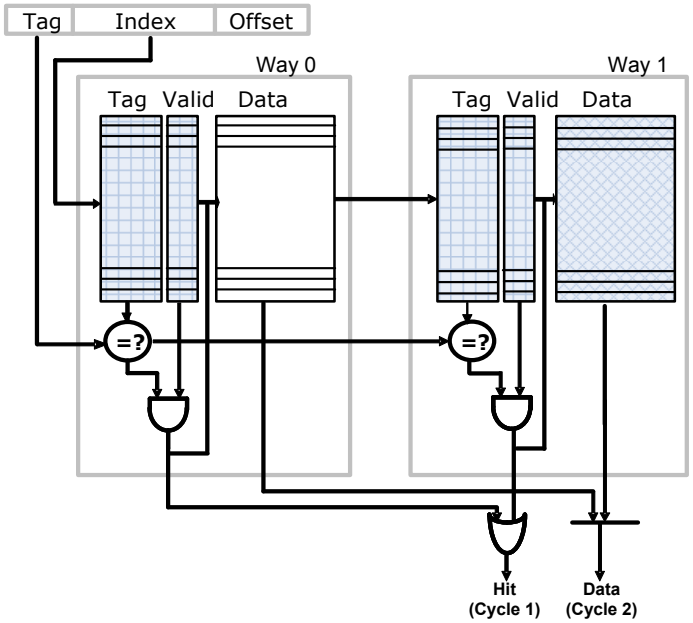
their position within the cache line. Using segmented word lines for the data portion of the instruction cache, we can fetch the necessary instruction words while activating only the necessary sense-amplifiers in each case.

Similar to the bit line segmentation approach proposed in [30], the internal organization of each raw in the data array is modified into segments. Each word line is split into independent segments. An additional common word line runs across the segments. The word line within each segment can be connected or isolated from the common line. The length of the basic block and the instruction pointer available in the BBD can be used to isolate all but the targeted segments from the common word line. The prechargers and sense-amplifiers of the bit lines for the isolated segments are disabled saving their energy. The effective capacitive loading on the common line also decreases. The reduction is somewhat offset by the common line that spans a single segment and the diffusion capacitances of the isolating switches. Figure 6.1 illustrates this optimization by comparing an instruction cache access for a conventional cache to the BLISS selective word approach. The selective word access of the instruction cache with a configuration similar to the instruction cache in Table 6.1 saves 20% of the energy consumed per instruction cache access.

Instruction access latency is often on the critical path for the overall processor clock cycle. In a conventional set-associative cache design, the tag and data arrays are accessed in parallel to reduce latency. This approach wastes energy in the bit lines and sense-amplifiers of the cache as it must drive all associative ways of the data component. Front-end decoupling can tolerate higher instruction cache latency with minimal impact to the performance. Hence, we can access first the tags for a set-associative instruction cache, and in subsequent cycles, access the data only in the way that hits [91]. A serial cache design breaks up the instruction cache lookup into two components: the tag comparison and the data lookup. The tag array is accessed first. In the next cycle, we access the data array for only the way that hits avoiding unnecessarily driving the bit lines of other ways of the cache and decreasing the number of necessary sense-amplifiers. In Figure 6.2, we illustrate this optimization by



Conventional: Parallel Access



BLISS: Serial Access

Figure 6.2: An illustration of the serial access to the tag and data arrays for the instruction cache.

Front-End Parameters			
	Base	**FTB**	**BLISS**
Fetch Width	4 Instructions/cycle	1 Fetch Block/cycle	1 Basic Block/cycle
Target Predictor	BTB: 1K entries, 4-way 1-cycle access	FTB: 1K entries, 4-way 1-cycle access	BB-cache: 1K entries, 4-way 1-cycle access 8 entries/line
Decoupling Queue	–	FTQ: 8 entries	BBQ: 8 entries
I-cache Latency	2-cycle pipelined	3-cycle pipelined	

Common Processor Parameters	
Hybrid Predictor	gshare: 4K counters PAg L1: 1K entries, PAg L2: 1K counters selector: 4K counters
RAS	32 entries with shadow copy
I-cache	16 KBytes, 4-way, 64B blocks, 1 port
Issue/Commit	4 instructions/cycle
IQ/RUU/LSQ	32/64/64 entries
FUs	6 INT & 3 FP
Data cache	32 KBytes, 4-way, 64B blocks, 2 ports, 2-cycle access pipelined
L2 cache	1 MByte, 8-way, 128B blocks, 1 port, 12-cycle access, 4-cycle repeat rate
Main memory	100-cycle access

Table 6.1: The microarchitecture parameters used for the energy optimization experiments. The common parameters apply to all three models (base, FTB, BLISS).

comparing the conventional parallel access approach of the instruction cache to the BLISS serial access. The BLISS serial access of the 4-way set-associative cache configuration listed in Table 6.1 saves on average 58% of the energy consumed per instruction cache access.

6.3 Methodology

For energy evaluation, we use a similar methodology and tools to what we used for performance evaluation discussed in Section 5.1. We simulate a 4-way superscalar processor to evaluate energy of the BLISS-based front-end and compare it to the conventional (base)

and the FTB-based front-ends. The 4-way configuration presents a practical commercial implementation for high-end processors in 2007. Table 6.1 summarizes the key architectural parameters, which are very similar to the 4-way parameters listed in Table 5.1 except for the modestly sized instruction cache and data cache. The instruction cache latency for both the BLISS-based and FTB-based designs is set to 3 cycles to support the serial access.

We also study the same 12 SPEC CPU2000 benchmarks using their reference datasets compiled at the -O3 optimization level. For energy measurements, we use the Wattch framework with the cc3 power model [15]. In this non-ideal, aggressive conditional clocking model, power is scaled linearly with port or unit usage, except that unused units dissipate 10% of their maximum power, rather than drawing zero power. Energy consumption was calculated for a 0.10μm process with a 1.1V power supply. In the following results, the reported *Front-end Energy* includes instruction cache, predictors, and BTB, FTB-cache, or BB-cache. *Total Energy* includes all the processor components (front-end, execution core, and all caches).

Table 6.2 presents the normalized energy consumed per access and the normalized average energy consumption for the various components of the 4-way processor configuration in Table 6.1. The branch predictor alone consumes 5.6% of the total energy per access. The instruction cache alone consumes 6.9% of the total energy per access. The average energy consumptions for both the branch predictor and the instruction cache are higher (9.9% and 13.6% respectively) because they are accessed nearly every cycle. This is not the case for other structures (L2-cache, FP-ALU, etc) as they are idle for several cycles wasting only leakage energy (10% for the Wattch cc3 model). FP-ALU, for example, is only accessed for programs that have FP instructions.

Component	Energy Per Access	Average Energy Consumption
Branch Predictor	5.6%	9.9%
Instruction Cache	6.9%	13.6%
Rename Logic	0.9%	1.4%
Instruction Window	9.3%	16.2%
Load/Store Queue	6.4%	4.5%
Arch. Register File	5.9%	3.8%
Result Bus	6.0%	8.8%
INT ALU	9.1%	7.3%
FP ALU	14.0%	11.2%
Data Cache	16.7%	16.3%
Level 2 Cache	19.1%	7.0%

Table 6.2: Energy per access and average energy consumption for the various components of the 4-way processor configuration.

6.4 Evaluation

In this section, we present the energy evaluation for BLISS. First, we evaluate the energy consumed by the front-end alone and then we consider the overall energy consumption for the processor. Finally, we consider energy-delay-squared product (ED^2P) as a metric that combines performance and energy for high-end processors.

6.4.1 Front-End Energy

Figure 6.3 compares the front-end energy saving achieved for the 4-way processor configuration with the three front-ends. On average, 13% of the total processor energy is consumed in the front-end engine itself as it contains a number of large SRAM structures (cache, BTB, predictors). The FTB design saves 52% of the front-end energy consumed by a conventional front-end design. The BLISS design achieves even higher improvement and saves 65% of the front-end energy consumed by a conventional front-end.

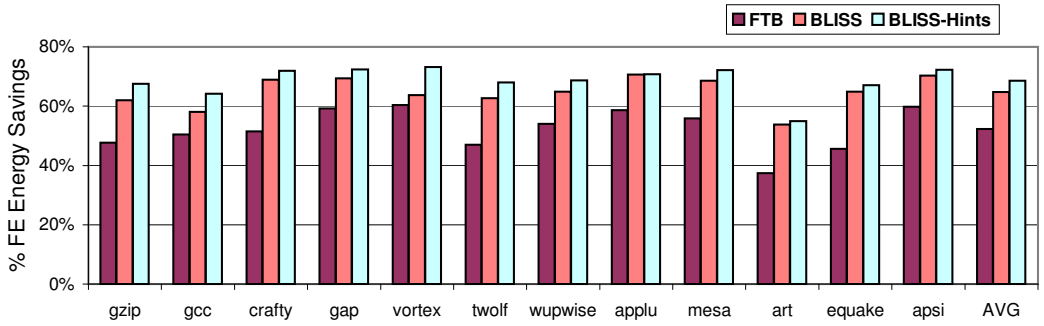


Figure 6.3: Front-end energy saving comparison for the 4-way processor configuration for the BLISS and FTB front-ends.

To understand the sources of energy savings, we look at the energy consumption for prediction and instruction cache accesses. Figure 6.4 compares the normalized energy consumed on prediction for the FTB and BLISS designs over the base design. The prediction energy for the base design includes energy consumed in accessing and training the BTB and the predictor tables. For the FTB design, prediction energy includes the energy consumed in accessing the FTB and the predictors. For BLISS, it includes the energy consumption for the BB-cache and the predictors. Both of the FTB and the BLISS designs significantly reduce the prediction energy consumption by only accessing and training the predictor once per block. The BLISS design reduces by 63% the energy consumed for prediction. The FTB design only reduces by 42% the energy consumption for prediction, as dynamic block recreation affects the prediction accuracy of the hybrid predictor and results in longer training and increased interference. When static branch hints are in use (BLISS-Hints), an additional 8% saving is achieved as hints allow the front-end to determine which component of the hybrid predictor is the best to consult and update for the associated control-flow instruction without accessing or training the selector.

Figure 6.5 presents the normalized energy consumption for accessing the instruction cache for the FTB and BLISS front-ends over the base design. Both of the FTB and BLISS front-ends reduce significantly the energy consumption in the instruction cache through selective word accesses and serial tag/data accesses in the instruction cache. The BLISS

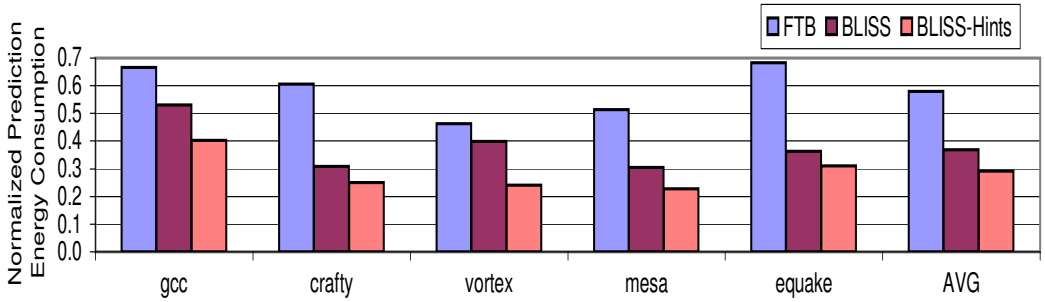


Figure 6.4: Normalized prediction energy consumption for the 4-way processor configuration for the FTB and BLISS front-ends over the base design. We present data for a representative subset of benchmarks, but the average refers to all benchmarks in this study.

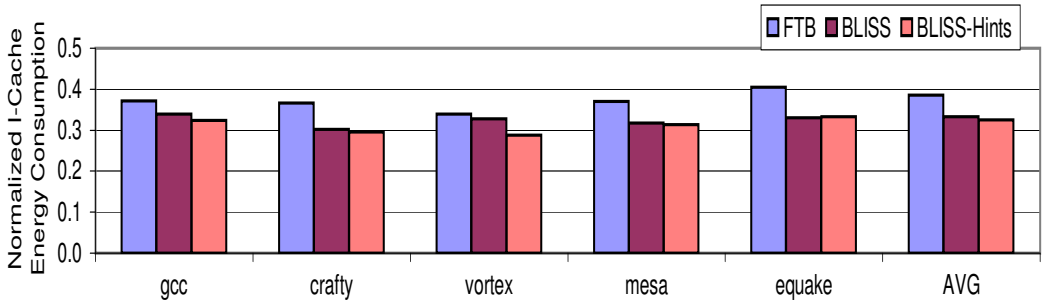


Figure 6.5: Normalized instruction cache energy consumption for the 4-way processor configuration for the FTB and BLISS front-ends over the base design. We present data for a representative subset of benchmarks, but the average refers to all benchmarks in this study.

energy saving in accessing the instruction cache is 7% better than the FTB design. This is mainly because BLISS reduces by 12% the number of accesses to the instruction cache compared to the base, while for the FTB design, the number of instruction cache accesses is 12% higher than the base design due to the maximum length fetch blocks that are inserted in the FTQ after an FTB miss. Note that BLISS achieves similar savings with and without the static prediction hints. The small difference between the two is due to the slightly different control-flow paths followed by the two due to differences in prediction.

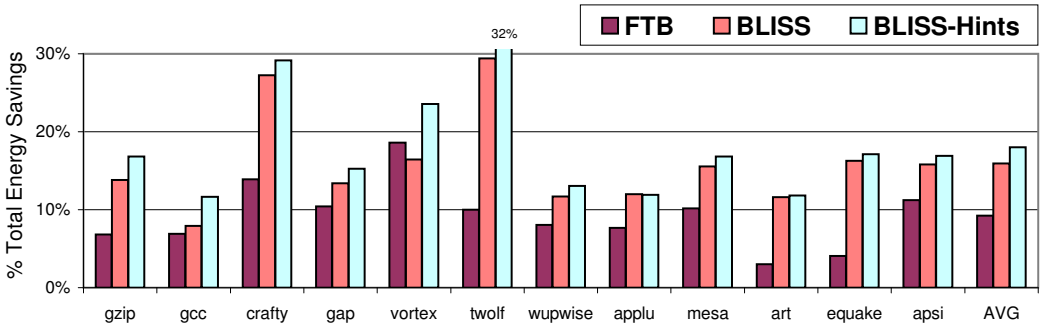


Figure 6.6: Total energy saving comparison for the 4-way processor configuration for the BLISS and FTB front-ends.

6.4.2 Total Energy

Figure 6.6 compares the saving in total energy consumption achieved for the 4-way processor configuration with the three front-ends. The reported total energy includes all the processor components (front-end, execution core, and all caches). BLISS offers 16% and 18% total energy advantages over the base design with and without the static prediction hints respectively. BLISS-based design actually achieves 75% of the total energy improvement suggested in Figure 2.8. BLISS also provides 7% total energy advantage over FTB as dynamic fetch block creation in the FTB front-end leads to execution of misspeculated instructions that waste energy.

Figure 6.7 presents the normalized energy consumption for the various components of the 4-way processor for the BLISS design without static branch hints over the base design. The significant reduction in energy consumption in the BLISS front-end accounts for almost half of the total energy savings (7.9%). We also see a consistent 7% to 8% reduction in the energy consumption for all of the processor components as BLISS limits the leakage energy wasted by idle resources and reduces the dynamic energy wasted on fetching and executing erroneous misspeculated instructions in the whole pipeline.

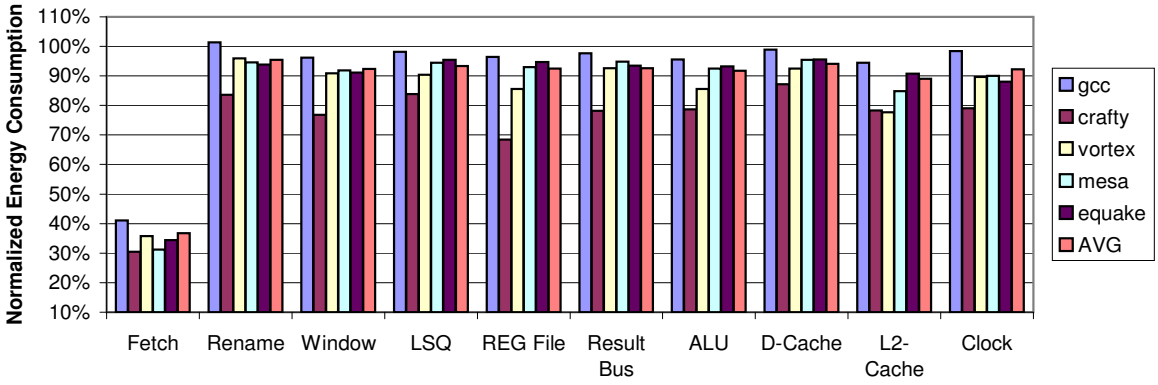


Figure 6.7: Normalized energy consumption for the various components of the 4-way processor for the BLISS design without static branch hints over the base design. We present data for a representative subset of benchmarks, but the average refers to all benchmarks in this study.

6.4.3 Energy-Delay Product Comparison

Energy-delay products are important metrics in microarchitecture design as they indicate how efficient the processor is at converting energy into speed of operation. They capture the energy usage per operation. A lower value indicates that energy is more efficiently translated into the speed of operation. The energy-delay product (**EDP**) represents an equal tradeoff between energy and delay [31]. The energy-delay-squared product (**ED^{2}P**) places more emphasis on deliverable performance over energy consumption [66], which is more appropriate for high-performance, energy-efficient processors. The ED^{2}P implies that a 1% reduction in circuit delay is worth paying a 2% increase in energy usage.

Figure 6.8 compares the Energy-delay-squared product (ED^{2}P) improvement achieved for the 4-way processor configuration with the three front-ends. The BLISS design achieves an 83% improvement in ED^{2}P, while the FTB design has only 35% overall ED^{2}P improvement over the base.

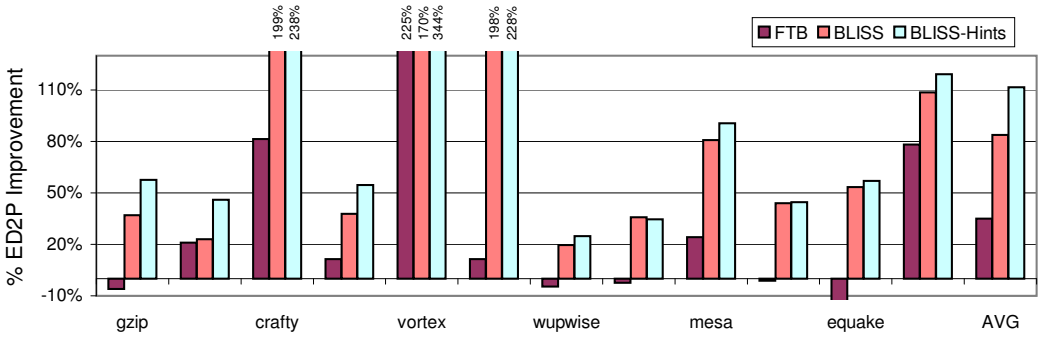


Figure 6.8: Energy-delay-squared product (ED^2P) improvements comparison for the 4-way processor configuration for the BLISS and FTB front-ends.

6.5 Related Work

Significant research has focused on reducing power and energy consumption in or through the front-end. Most techniques trade off a small performance degradation for significant energy savings. Some techniques target improving the instruction cache energy consumption by way prediction [84], selective cache way access [2], sub-banking [30], tag comparison elimination [75, 114], and reconfigurable caches [87, 113]. Other techniques target improving the energy efficiency of the predictors using sub-banking [76], front-end gating [64], selective prediction [7], eliminating predictor and BTB accesses for non-branch instructions [76], using profile data to eliminate meta predictor [27] or to switch off part of the predictor [20], and selective predictor accesses to avoid using predictor for well-behaved branches [8].

Other techniques focused on controlling over-speculation in the pipeline for branches with low prediction confidence to limit energy wasted on misspeculated instructions. A confidence estimator is used to assess the quality of branch predictions [34, 44]. Pipeline gating [64] uses confidence information to stop wrong-path instructions from entering the pipeline. Selective throttling [4] applies different gating techniques depending on the confidence estimation, with the goal of obtaining an optimal tradeoff between power and performance. There are also techniques that focus on tuning the resources of the processor to the

needs of the program by monitoring its performance to reduce energy [5, 43]. Such techniques are rather orthogonal to the block-aware ISA and can be used with a BLISS-based front-end engine.

6.6 Summary

In this chapter, we demonstrated that BLISS reduces the processor overall energy consumption as it minimizes leakage energy wasted by idle resources and reduces dynamic energy wasted on fetching and executing erroneous misspeculated instructions. We also showed that BLISS significantly reduces the front-end energy consumption by reducing the number of accesses to the front-end structures and reducing the average energy consumed per access for each structure. Through detailed simulation, we have shown that the BLISS-based design allows for 63% front-end energy, 16% total energy, and 83% ED^2P improvements over a conventional superscalar design. The ISA-supported front-end also outperforms (21% front-end energy, 7% total energy, and 48% ED^2P) advanced decoupled front-ends that dynamically build fetch blocks in hardware. This significant energy improvement is achieved in addition to the performance advantages demonstrated in Chapter 5. Overall, this work establishes the potential of using expressive ISAs to address difficult hardware problems in modern processors in ways that benefit **both** performance and energy consumption.

Chapter 7

BLISS for Embedded Processors

Our analysis so far primarily focused on the performance and energy evaluation of BLISS for high-end processors. This chapter shifts the focus to embedded processors, which are a key component in most consumer, communications, industrial, and office automation products. Similarly to high-end, performance and energy efficiency are critical design metrics for embedded processors. Good performance is important for embedded processors in order to meet the increasing requirements of demanding applications such as image, voice, and video processing. Energy consumption dictates if the processor can be used in portable or deeply embedded systems for which battery size and lifetime are vital parameters. In previous chapters, we showed that BLISS improves both metrics for high-end processors, but we must also validate that this result holds for lower-end, embedded designs.

In addition to performance and energy efficiency, code size is a critical design metric for embedded processors. Code size determines the amount and cost of on-chip or off-chip memory necessary for program storage. Instruction memory is often as expensive as the processor itself. Even though it seems counter-intuitive, the use of additional block descriptors in BLISS leads to significant reductions in the code size. The descriptors enable code size optimizations by removing redundant sequences of instructions across basic blocks

and by allowing a fine-grain interleaving of 16-bit and 32-bit instructions without overhead instructions.

In this chapter, we examine the use of BLISS to *improve all three efficiency metrics* for embedded processors at the same time: smaller code size **and** better performance **and** lower energy consumption. Improving the three metrics simultaneously is traditionally difficult to achieve as they introduce conflicting tradeoffs. Most known techniques improve one or two metrics, but not all the three at the same time. In Section 7.1, we present the code size optimizations enabled with BLISS. Section 7.2 explains the tools and methodology. In Section 7.3, we present and analyze the evaluation results. Section 7.4 highlights related research.

7.1 Code Size Optimizations

Naïve translation of a RISC binary such as MIPS-32 to the corresponding BLISS executable leads to larger code size due to the addition of block descriptors. With five instructions per block on the average, the code size increase is 20%. Nevertheless, BLISS allows for three types of code size optimizations that eliminate this handicap and lead to significant code size savings over the original.

7.1.1 Basic Optimizations

Basic code size optimizations target redundant jump and branch instructions. These optimizations are unique to BLISS. All jump instructions can be removed as they are redundant; the BBD defines both the control-flow type and the offset. Moreover, certain conditional branch instructions can be eliminated if they perform a simple test (equal/not equal to zero) on a register value produced within the same basic block. We encode the simple condition test in the opcode of the producing instruction which is typically a simple integer arithmetic operation (add or sub). Note that the branch target is provided by the BBD and does not

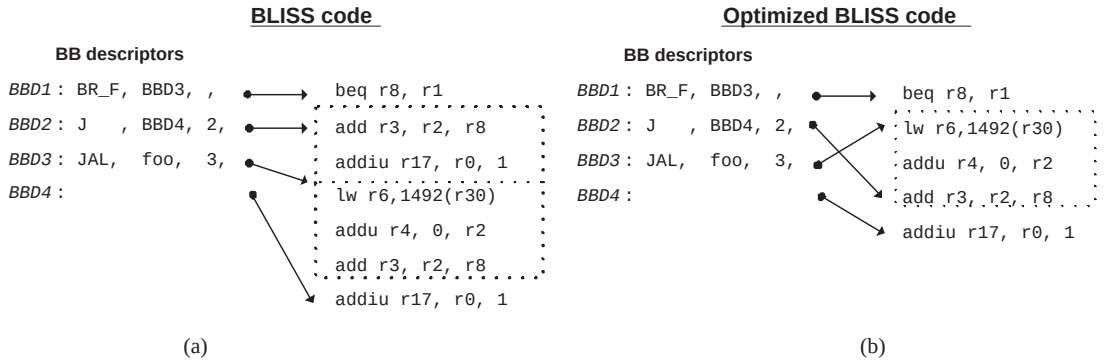


Figure 7.1: Example to illustrate the block-subsetting code optimization. (a) Original BLISS code. (b) BLISS code with the block-subsetting optimization. For illustration purposes, the instruction pointers in basic block descriptors are represented with arrows.

need to be provided by any regular instruction, which frees a large number of instruction bits.

7.1.2 Block Subsetting

BLISS facilitates the removal of repeated sequences of instructions [22]. All instructions in a basic block can be eliminated, if the exact sequence of the instructions can be found elsewhere in the binary. The matching sequence does not have to be an identical basic block, just an identical sequence of instructions. We maintain the separate descriptor for the block but change its instruction pointer to point to the unique location in the binary for that instruction sequence. We refer to this optimization as *Block Subsetting*. Figure 7.1 presents an example to illustrate this optimization. The two instructions in the second basic block in the original code appear in the exact order towards the end of the instruction section. Therefore, they can be removed as long as the instruction pointer for BBD2 is updated.

Block subsetting leads to significant code size improvements because programs frequently include repeated code patterns. Moreover, the compiler generates repeated patterns for tasks like function setup, stack handling, and loop setup. By removing jump and branch

instructions, the basic code size optimizations expose more repeated instruction sequences that block subsetting can eliminate. In Figure 7.1, the removal of the JAL instruction, which would otherwise terminate the third basic block, enables the elimination of the instructions in the second basic block. Instruction similarity is also improved because BLISS stores branch offsets in the BBDs and not in regular instructions.

In theory, we can improve similarity further if we look at data-flow graphs abstracting out the specific registers used instead of looking at specific code sequences. However, encoding such similarity can be expensive. Alternatively, more repeated instruction sequences can be exposed if we consider instruction reordering. In this study, we do not evaluate such techniques.

Block subsetting can affect performance both ways by interfering with the instruction cache hit rate. It can reduce the hit rate as it decreases spatial locality in instruction references. Two sequential basic blocks may now point to instruction sequences in non-sequential locations. However, the BLISS front-end can tolerate higher instruction cache miss rates as it allows for effective prefetching using information in the basic block descriptors. Block subsetting can also improve cache performance as it reduces the cache capacity wasted on repeated sequences. A comprehensive evaluation of this optimization is presented in Section 7.3.

7.1.3 Block-Level Interleaving of 16/32-bit Code

An effective technique for code size reduction is to extend the instruction set to support two instruction lengths, with the processor capable of executing both of them. The long instructions are the instructions of the original ISA. The short instructions are a subset of the long instructions and encode the most commonly used instructions using a short encoding format. MIPS-16 and Thumb-2 [54, 108] are examples of such instruction sets. They provide 16-bit extensions to the 32-bit ISAs. A section of code that completely uses

Figure 7.2: Code size, execution time, and total energy consumption for 32-bit, 16-bit, and selective 16-bit executables for a processor similar to Intel's XScale PXA270 processor running the MediaBench benchmarks. Lower bars present better results.

the 16-bit instructions can potentially save 50% of the size of a similar code that uses the 32-bit instructions. However, the short instruction format implies access to a limited set of registers, limited number of opcodes, and a very short immediate and offset field. These challenges limit the potential saving and lead to an increased number of dynamic instructions that result in significant performance losses.

The granularity of interleaving 16-bit and 32-bit code can have a significant impact on the performance overhead of 16-bit instructions. Interleaving can be done at different levels. MIPS-16 [54] allows mixing of 16-bit and 32-bit instructions at the function-level granularity. A special `JALX` instruction is used to switch between functions with 16-bit and 32-bit instructions. However, function-level granularity is restrictive as many functions contain both performance critical and non-critical code. Alternatively, one can interleave 16-bit and 32-bit code at instruction granularity [36, 55, 81]. Special instructions are still necessary to switch between the 16 and 32-bit sections, hence there is an overhead for each switch.

Figure 7.2 shows the impact of using a 16-bit ISA on the average code size, execution time, and energy consumption for a simple embedded processor like the XScale PXA270

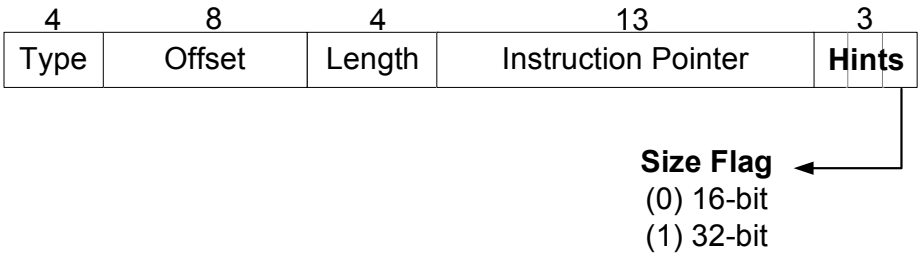


Figure 7.3: The basic block descriptor format with the size flag that indicates if the actual instructions in the block use 16-bit or 32-bit encoding.

running the MediaBench applications. The 16-bit instructions lead to 41% code size savings at the cost of 11% and 13% higher execution time and energy consumption. It is possible to recover the performance and energy overhead by selectively using 16-bit instructions only for non-critical sections of the code using a few overhead instructions to specify switches between the two formats [36, 55]. As shown in Figure 7.2, selective use of short instructions maintains the code size savings and restores the performance and energy consumption of the original, 32-bit code.

BLISS provides a flexible mechanism for interleaving 16-bit and 32-bit code at the granularity of basic blocks. This is significantly better than the function-level granularity in MIPS-16. It is also as flexible as the instruction-level granularity because either all instructions in a basic block are frequently executed (performance critical) or none of them is. A single bit in the hints field of the basic block descriptor shown in Figure 7.3 provides a flag to specify if the block contains 16-bit or 32-bit instructions. No new instructions are required to specify the switch between the 16-bit and 32-bit modes. Hence, frequent switches between the two modes incur no additional runtime penalty. We can aggressively interleave 16-bit and 32-bit instructions in the code. The only restriction is that a set of 16-bit instructions must start at a 32-bit alignment point. 16-bit NOP instructions are used for alignment when required.

All descriptors are 32-bit even for blocks that contain 16-bit instructions. This is a small overhead as instructions on average have five to eight times the size of descriptors.

In addition, as we discussed in the previous section, the availability of descriptors enables removing redundant sequences of instructions across basic blocks. The resulting reduction in code size significantly exceeds the small overhead of having 32-bit descriptors for blocks with 16-bit instructions. As descriptors are fetched early in the processor pipeline, we know the mode for each set of instructions before they are even fetched. This allows us to dynamically expand the 16-bit instructions into their corresponding 32-bit instructions at fetch stage with no performance overhead. With conventional processors, the translation usually occurs during the decode stage.

7.2 Methodology

7.2.1 Processor Configurations

To demonstrate the wide applicability of the BLISS ISA across the spectrum of embedded computing, we simulate two processor configurations. The first one is modeled after the Intel XScale PXA270 processor [42] as an example of a low-power embedded CPU for hand-held and portable applications. The second configuration is comparable to the IBM PowerPC 750GX processor [40] as a high-end embedded core for networking systems. Table 7.1 summarizes the key architectural parameters used for the two configurations. For BLISS, we split the baseline instruction cache resources between regular instructions (3/4 for BLISS instruction cache) and block descriptors (1/4 for BB-cache) as each basic block corresponds to 4 to 8 instructions on average. The smaller BLISS instruction cache does not incur more misses as 17% of the original MIPS instructions are eliminated from the BLISS code by the simple code size optimizations. The BLISS design no longer requires the BTB, therefore that area can be used for the BBQ. We fully model all contention for the L2-cache bandwidth between BB-cache misses and instruction cache or D-cache misses.

Front-End Parameters				
	XScale PXA270		PowerPC 750GX	
	Base	**BLISS**	**Base**	**BLISS**
Fetch Width	1 inst/cycle	1 BB/cycle	4 inst/cycle	1 BB/cycle
BTB	32-entry 4-way	–	64-entry 4-way	–
BB-cache	–	8 KBytes 4-way 32B Blocks 1-cycle access	–	8 KBytes 4-way 32B Blocks 1-cycle access
I-cache	32 KBytes 4-way 32B Blocks 2-cycle access	24 KBytes 3-way 32B Blocks 2-cycle access	32 KBytes 8-way 32B Blocks 1-cycle access	24 KBytes 6-way 32B Blocks 1-cycle access
Decoupling Queue	–	4 entries	–	4 entries

Common Processor Parameters		
	XScale PXA270	**PowerPC 750GX**
Execution	in-order	out-of-order
Predictor	Bimod 256-entries	Bimod 512-entries
RAS	8 entries	16 entries
Issue/Commit Width	1 instructions/cycle	4 instructions/cycle
IQ/RUU/LSQ Size	16/32/32 entries	32/64/64 entries
FUs	1 INT & 1 FP	2 INT & 1 FP
D-cache	32 KBytes, 4-way, 32B blocks 1 port, 2-cycle access	32 KBytes, 8-way, 32B blocks 1 port, 1-cycle access
L2 cache	256 KBytes, 4-way, 64B blocks 1 port, 5-cycle access	1 MByte, 8-way, 128B blocks 1 port, 5-cycle access
Main memory	30-cycle access	45-cycle access

Table 7.1: The microarchitecture parameters used for embedded processors evaluation and code size optimization experiments.

The operation of the front-end with BLISS for embedded processors is similar to the operation for the high-end processors. The front-end resources for the two evaluated embedded processors in Table 7.1 are sized differently from the high-end processor in Table 5.1. The back-ends for the two embedded designs also have fewer resources with the XScale design consuming instructions in an in-order manner at lower throughput. Nevertheless, the front-end behavior for the two embedded designs is the same as the high-end.

7.2.2 Tools and Benchmarks

Similarly to the case of high-end processors, our simulation framework for embedded systems is based on the Simplescalar tool set. We also used the same methodology in Section 6.3 for energy evaluation. We assume a 0.10μm process with a 1.1V power supply.

Instead of the SPEC benchmarks, we study 10 applications from the MediaBench suite [57]. The MediaBench programs are more appropriate for the evaluation of embedded processors as they include common embedded applications such as communication and multimedia processing. The benchmarks are compiled at the -O2 optimization level using gcc and simulated to completion. The -O2 level includes optimizations that improve performance significantly but do not increase the code size. The compiler does not perform loop unrolling or function inlining when this option is specified. Hence, the density of the initial code is quite good.

Block subsetting is performed in the code optimization step during the BLISS code generation process (see Figure 3.3). If all of the instructions of a basic block appear elsewhere in the code stream, the instructions are eliminated and the descriptor pointer is updated. Although instruction rescheduling and register re-allocation might help in identifying additional repetitions [22], they are not considered in this study. We allow splitting a large basic block into two sequential basic blocks to further reduce the static code size. Basic block splitting requires adding extra BBDs, however, a net reduction in the code size can be still achieved if the instructions from one or both of the new blocks can be eliminated. Block splitting is only performed when the net results is a reduction in the static code size and only for large blocks (more than 6) to avoid negatively affecting the BB-cache performance.

To determine which basic blocks will use 16-bit encoding for their instructions, we employ the static profitability-based heuristic proposed in [36]. Converting a block to use the short instruction format impacts both the code size and the performance of the program. Instructions in 16-bit format can only access 8 registers and may lead to performance loss

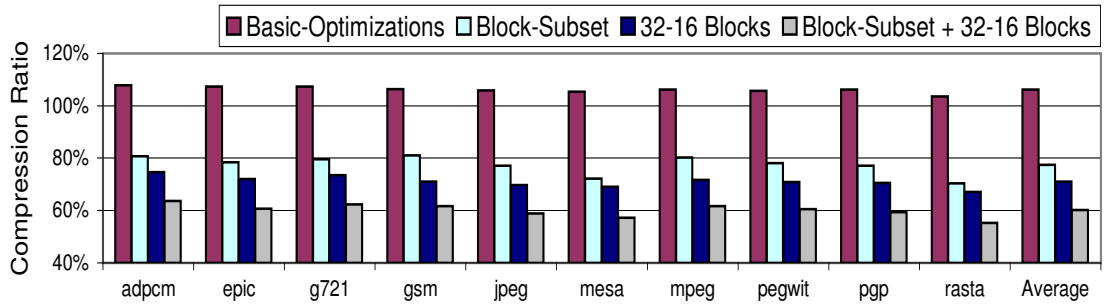


Figure 7.4: Compression ratio achieved for the different BLISS executables over the baseline 32-bit MIPS code.

due to register spilling. In their technique, the impact is estimated using a profitability analysis (PA) function. The PA function estimates the difference in code size and performance if the block were to be implemented in the short format compared to normal encoding. These estimates can be used to tradeoff between performance and code size benefits for the program. The heuristic tries to achieve similar code size reduction to what is possible with exclusive use of 16-bit instructions without impacting performance.

7.3 Evaluation for Embedded Processors

This section presents the code size, performance, and energy evaluation of BLISS for the two embedded processors.

7.3.1 Code Size

Figure 7.4 presents the compression ratio achieved for the different BLISS executables compared to the MIPS-32 code size. Compression ratio is defined as the percentage of the compressed code size over the original code size. This means that lower compression ratios are better.

Direct translation with basic-optimizations (*Basic-Optimizations* bar) of MIPS-32 code leads to an increase in code size with a 106% average compression ratio. This is 6% worse

than the original MIPS-32 code. Block subsetting (*Block-Subset* bar) yields an average compression ratio of 77%. Mixing 16- and 32-bit instruction sets makes the BLISS executables 29% less than the MIPS-32 code (71% compression ratio). Combining the two optimizations leads to 61% compression ratio. Note that when the two optimizations are enabled, the individual reductions in code size do not add up. This is due to two reasons. First, block subsetting is only performed within blocks of the same instruction size: 16-bit instruction blocks can only be considered for block subsetting with other 16-bit blocks and the same applies to 32-bit blocks. Hence, the opportunity for removing repeated sequences is less. Second, the saving from eliminating 16-bit instructions is half the saving from eliminating 32-bit instructions.

Table 7.2 presents additional detailed statistics on the code size optimizations studied. Extra instructions are required when interleaving 16 and 32-bit blocks due to register spilling as instructions in 16-bit format can only access 8 registers. It is interesting to note that 95% of the code is none performance critical and can be converted to 16-bit encoding.

7.3.2 Performance Analysis

Figure 7.5 compares the percentage of IPC improvement achieved for the different BLISS executables for the XScale and the PowerPC processor configurations over the base design. The original BLISS with basic optimizations provides an 11% average IPC improvement for the XScale configuration and a 9% average improvement for the PowerPC configuration over the base design. The BLISS advantage is mostly due to the elimination of a significant number of pipeline flushes as a result of more accurate prediction (see Section 5.2.2). Note that the performance advantage for the PowerPC configuration is slightly lower than the advantage for the XScale. As we explained in Section 4.1.2, BLISS tolerates higher instruction cache access time. The instruction cache access time for the XScale is two cycles, while the access time for the PowerPC is only one cycle. This implies that the performance

Benchmark	MIPS-32	BLISS Basic Optimization		Block-Subset without BB-splitting	Block-Subset with BB-splitting		Interleaving 16/32 Blocks	
	Code Size (KByte)	J/B Inst. Removed	No. of BBs	No. of Inst. Eliminated	Extra BBDs Added	No. of Inst. Eliminated	% of Inst. Using 16-bit	Extra Inst. Added
adpcm	37	1626	2593	2014	534	3054	94%	730
epic	69	2841	4561	3777	1058	6036	96%	816
g721	43	1884	2961	2391	637	3659	93%	664
gsm	70	2805	4397	3405	1178	5649	94%	948
jpeg	112	4391	6567	5744	2115	10109	96%	1286
mesa	453	16811	24673	25183	10529	47825	95%	6224
mpeg2.dec	79	3413	5124	3887	1224	6391	96%	1242
mpeg2.enc	106	4241	6490	4980	1758	8637	95%	1820
pegwit	82	2959	4476	3775	1436	7369	95%	1746
pgp	206	10353	14296	11425	3319	18231	96%	1242
rasta	232	10559	13776	11647	5426	24861	96%	980

Table 7.2: Statistics for the BLISS code size. The extra instructions for the 16-bit format are due to register spilling and the short offsets for data references.

benefit of BLISS for the XScale is going to be higher as BLISS will tolerate its instruction cache latency.

BLISS provides very similar IPC improvements even with block subsetting. The additional instruction cache misses due to reduced instruction locality are well tolerated through prefetching using the contents of the BBQ. The elimination of repeated instruction sequences allows for more unique instructions to fit in the instruction cache at any point in time. Hence, certain applications observe an overall higher instruction cache hit rate.

With interleaved 16-bit and 32-bit code, BLISS achieves a 10% average IPC improvement over the base for XScale and an 8% average improvement for PowerPC. Two factors contribute to the change in performance gains. With 16-bit encoding, twice as many instructions can fit in the instruction cache, which leads to lower miss rate. However, 16-bit encoding introduces additional dynamic instructions to handle register spilling and long

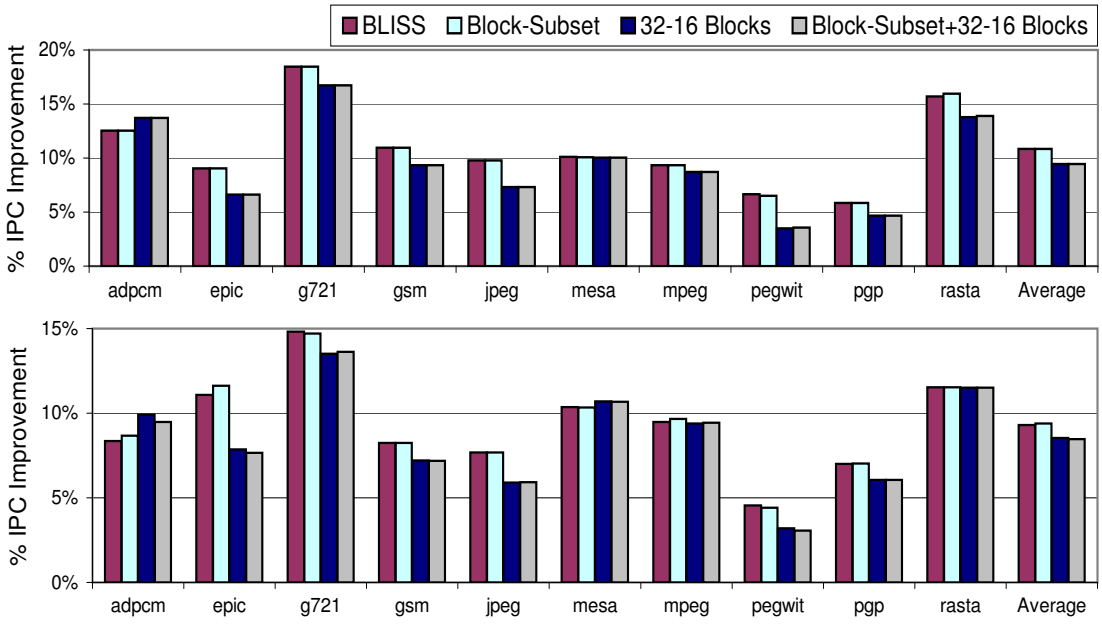


Figure 7.5: Percentage of IPC improvement for the different BLISS binaries over the base design. The top graph is for the XScale processor configuration. The bottom one is for the PowerPC configuration.

offsets for load and store instructions. On average, 1% more instructions are executed compared to the original code. The net effect is a small degradation in average performance compared to the original BLISS. Nevertheless, for benchmarks like `mesa`, which stresses the instruction cache capacity, the net effect is a small performance improvement. Using block subsetting in addition to interleaved 16-bit/32-bit instructions results in a similar performance as the one observed when the optimization is enabled on the original BLISS code.

7.3.3 Energy Analysis

Figure 7.6 compares the percentage of total energy improvement achieved using the different BLISS executables for the XScale and PowerPC processor configurations over the base design. BLISS offers a 23% total energy advantage for the XScale configuration and a

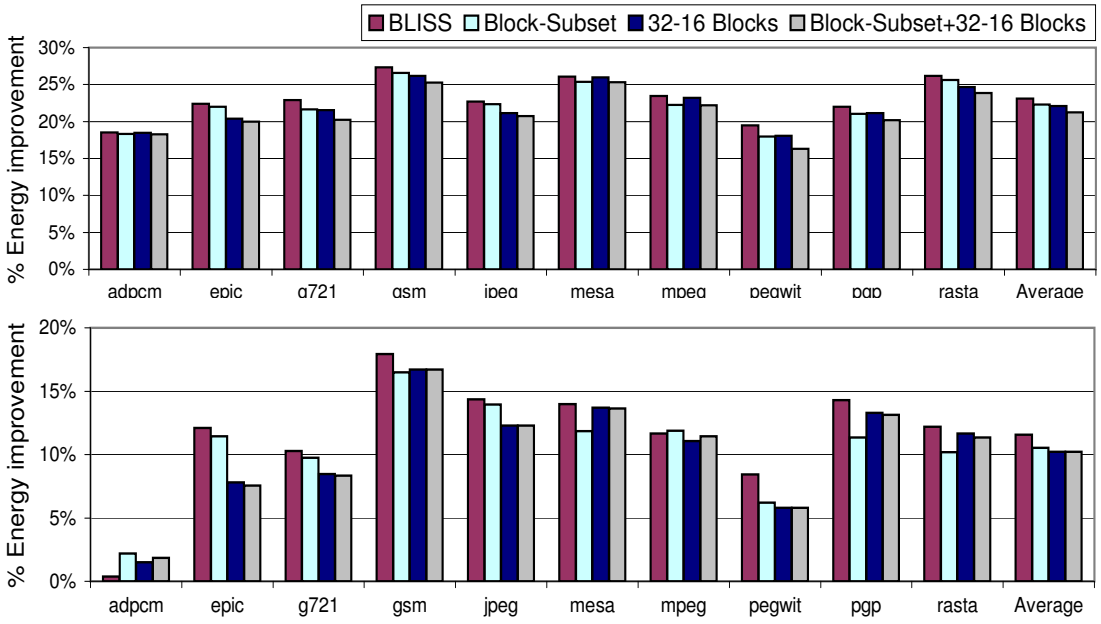


Figure 7.6: Percentage of total energy savings for the different BLISS binaries over the base design. The top graph is for the XScale processor configuration. The bottom one is for the PowerPC configuration.

12% advantage for the PowerPC configuration over the base design. The BLISS advantage is due to a number of factors: reduced energy spent on mispredicted instructions, selective word access in the instruction cache, merging of instruction cache accesses for sequential blocks, and judicious access to the branch predictor. `Adpcm` is the benchmark for which BLISS achieves the lowest energy-savings. `Adpcm` frequently executes short basic blocks (2.5 instructions per block) and requires frequent accesses to the BB-cache. The other benchmarks include 5 instructions or more per basic block.

The energy savings for XScale are higher than the savings for PowerPC. The XScale instruction cache access time is two cycles which places additional pressure on accurate control-flow prediction. Moreover, since XScale uses a single-issue pipeline, its BTB and the predictor are accessed for every instruction in the base design. With BLISS, they are accessed once per basic block. For PowerPC, the BTB and predictor are accessed once per 4 instructions, hence the energy saved in the front-end with BLISS is lower.

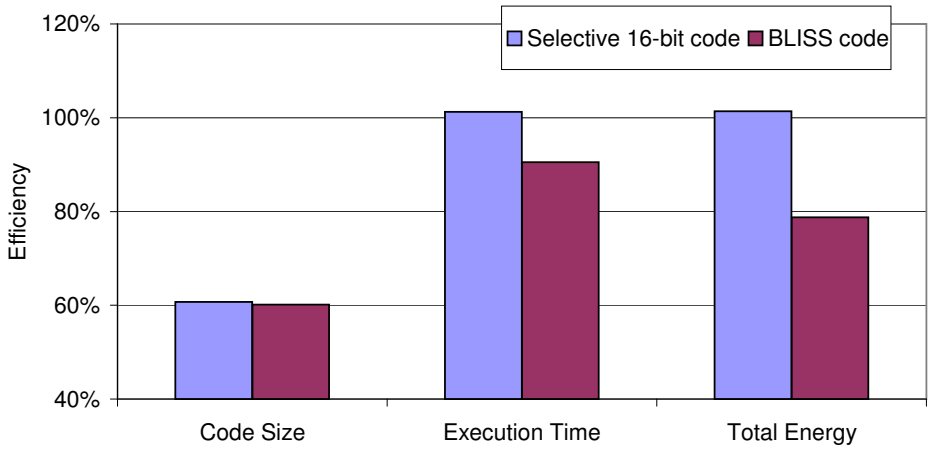


Figure 7.7: Average Code size, execution time, and total energy consumption for selective 16-bit and BLISS (with block-subset and 32/16 blocks) executables for the XScale processor configuration over the base. Lower bars present better results.

Block subsetting leads to slightly lower energy savings. When blocks are split to enhance subsetting, additional accesses to the BB-cache are introduced. Similarly, 16-bit encodings introduce some energy consumption due to the additional instructions. Nevertheless, BLISS with mixed instruction widths and subsetting provides a 21% total energy advantage for XScale and a 10% advantage for PowerPC over the base design.

7.3.4 Comparison to Selective Use of 16-bit Code

So far, we have compared BLISS to the base design running conventional RISC MIPS-32 code. In this section, we compare BLISS with code size optimization to the base running code optimized with selective use of 16 bit-instructions. Figure 7.7 compares BLISS with block subsetting and selective use of 16-bit blocks to selective use of 16-bit instructions with a conventional ISA like Thumb-2 and rISA (*Sel16*) [81, 36, 55]. Note that the same profitability heuristic is used with both ISAs to select which instructions or blocks to encode with 16 bits. The base XScale configuration with the full-sized instruction cache is used for Sel16.

By interleaving 16-bit and 32-bit encodings at instruction granularity, Sel16 achieves a 39% code size reduction. Nevertheless, the extra dynamic instructions for switching lead to a small performance and energy degradation. On the other hand, BLISS provides similar code size reduction and at the same time achieves 10% performance and 21% total energy advantages. BLISS overcomes the code size handicap of the extra block descriptors by allowing an additional code size optimization over Sel16 (block subsetting). Its performance and energy advantages are due to the microarchitecture optimization enabled with the BLISS decoupled front-end and the lack of special instructions for switching between 16-bit and 32-bit code. Overall, BLISS improves upon Sel16 by offering similar code density at superior performance and energy consumption.

7.4 Related Work

Many code size reduction techniques have been proposed and widely used in embedded systems [12]. Most techniques store the compressed program code in memory and decompression happens on instruction cache misses [110, 60, 50] or inside the processor [59, 23]. Compression is typically dictionary based. Such techniques reduce memory footprint and the off-chip bandwidth requirements for instruction accesses. When decompression occurs in the core, additional latency is introduced for instruction execution. When decompression occurs on cache refills, additional pressure is placed on the instruction cache capacity. BLISS reduces code size, places no additional pressure on instruction cache capacity, and improves on execution time. BLISS can be combined with a dictionary compression scheme behind the instruction cache for further code size improvements.

Cooper proposed a compiler framework for discovering and eliminating repeated instruction sequences [22]. The echo instruction has been proposed to facilitate elimination of such redundancies [56]. An echo instruction is used in the place of repeated sequences

and points back to the unique location for that code. Using echo instructions, 84% compression ratio is reported in [56]. BLISS facilitates redundancy elimination with block subsetting, which on its own leads to a 77% compression ratio. Moreover, BLISS allows for significant performance improvements in addition to code compression, which is not the case with previous proposals.

7.5 Summary

This chapter evaluated the use of the block-aware instruction set (BLISS) to achieve code size, performance, and energy improvements for embedded processors. BLISS achieves significant improvements in all three metrics, which is traditionally a challenge to accomplish. The software-defined basic block descriptors in BLISS facilitate code size optimizations by removing redundant sequences of instructions across basic blocks and by allowing a fine-grain interleaving of 16-bit and 32-bit instructions without overhead instructions. We showed that BLISS allows for 40% code size reduction over a conventional RISC ISA and simultaneously achieves 10% performance and 21% total energy improvements. Hence, BLISS improves concurrently the performance and cost of embedded systems.

Overall, the block-aware instruction set compares favorably to previous code density or performance enhancement techniques as it allows concurrent improvements in all three efficiency metrics. Therefore, it can be a significant design option for embedded systems.

Chapter 8

Low Cost Front-End Design for Embedded Processors

Apart from performance, energy, and code size, cost is an important concern for embedded processors. Managing the cost of an embedded system determines its success as profit margin for such systems is typically low. Even a small increase in device cost leads to significant increase in overall production cost for high volume manufacturing. Die area and power consumption are directly related to the cost of embedded processors. Die area determines the cost to manufacture the chip. Power consumption determines the cost to package and cool the chip. The challenge with embedded processors is that area and power efficiency must be achieved without compromising performance and energy efficiency. While energy and power are directly related, optimizing one of them does not necessarily translates to improvement in the other one. Power refers to the activity level at any given point while energy refers to the total amount of activities during program execution. It is possible that two program profiles have same energy usage with different peak power dissipations.

This chapter presents a number of design optimizations that target the power consumption and area of the front-end for embedded processors. Primarily, the goal is to use smaller front-end memory structures that consume less static and dynamic power and take up less

area. The challenge is to avoid performance loss due to the reduced capacity. We explore these optimization techniques using the block-aware instruction set (BLISS) architecture. The software-defined basic block descriptors provide a flexible substrate to implement these optimizations efficiently because the descriptors are directly visible to software, provide accurate information for prefetching, and can carry software hints. These optimizations balance out the performance loss of smaller instruction cache or prediction arrays. Hence, BLISS allows significant reorganization of the front-end without affecting performance, all within the same software model (ISA).

The remainder of this chapter is organized as follows. In Section 8.1, we discuss the background and motivation for this work. Section 8.2 presents the front-end hardware and software optimizations. In Section 8.3, we explain the methodology and tools used for evaluation. In Section 8.4, we present and analyze the evaluation results. Finally, Section 8.5 highlights related work.

8.1 Background and Motivation

Embedded processors consume a large fraction of their power budget in the front-end of their pipeline. The front-end contains several large SRAM structures such as the instruction cache, the branch target buffer (BTB), and the branch predictor, that are accessed on nearly every clock cycle. Such memory arrays are sized to hold a large amount of data in order to obtain good overall performance. For example, the Intel XScale PXA270 processor uses a 32-KByte instruction cache and a 128-entry BTB [42]. Combined, the two structures consume 22% of the processor total power budget.

Nevertheless, different programs exhibit different locality and memory access patterns and even a single program may not need all the available storage at all times. If the processor is executing a tight loop, for example, most of the instruction cache is underutilized

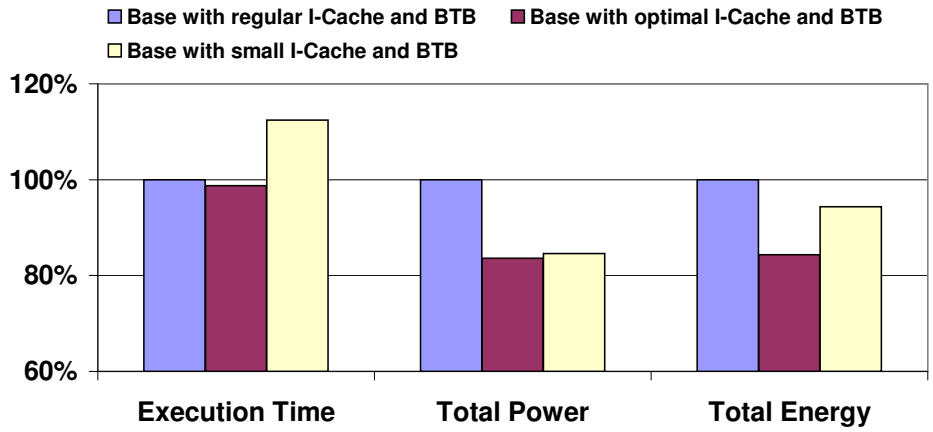


Figure 8.1: Normalized execution time, total power, and total energy consumption for the base design (32-KByte I-cache, 64-entry BTB), the base design with optimal I-cache and BTB, and the base design with small front-end arrays (2-KByte I-cache, 16-entry BTB). The processor core is similar to Intel's XScale PXA270 and is running benchmarks from the MediaBench and SPEC CPU2000 suites. Lower bars present better results.

as smaller cache could provide the same performance but with lower area, power, and energy requirements. Figure 8.1 quantifies the total energy and power wasted in the PXA270 processor due to sub-optimal instruction cache and BTB sizing for MediaBench and SPEC CPU2000 applications. The optimal configuration is found using a method similar to [97] where a continuum of cache sizes and configurations are simulated. During each cycle, the cache with the lowest power from among those that hit is selected. On average, 16% total power and 17% total energy are wasted if the processor uses larger than needed instruction cache and BTB.

Reducing the instruction cache and BTB capacity of embedded processors by a factor of 4 or 8 leads to direct die area and power savings. Table 8.1 presents the normalized power dissipation, area, and access time for different smaller instruction cache configurations over the 32-KByte instruction cache of the PXA270 processor using Cacti [102]. A 2-KByte instruction cache dissipates only 8.4% of the power dissipated by the 32-KByte cache and uses only 4.6% of its area. While the use of smaller arrays reduces die area and power dissipation, several applications will now experience additional instruction cache and BTB

Configuration	Power	Area	Access Time
2 KByte, 2 way associative	8.4%	4.6%	50.7%
4 KByte, 4 way associative	14.6%	9.2%	53.0%
8 KByte, 8 way associative	26.9%	18.0%	58.8%
16 KByte, 16 way associative	51.3%	42.8%	71.5%

Table 8.1: Normalized power dissipation, area, and access time for different instruction cache configurations over the XScale 32-KByte instruction cache configuration.

misses that will degrade performance and increase energy consumption. Figure 8.1 quantifies the performance penalty with the smaller instruction cache and BTB sizes (13% on average). Furthermore, the energy savings from accessing smaller arrays are nearly canceled from the cost of operating the processor longer due to the performance degradation.

8.2 Front-End Optimization

This section presents both hardware and software techniques that can reduce the performance degradation of the small front-end structures. The hardware-based techniques include *instruction prefetching*, *unified instruction cache and BTB structures*, and *tagless instruction caches*. Software-based techniques include *instruction re-ordering* and various forms of *software hints*. Instruction prefetching hides the latency of extra cache misses by fetching instructions ahead of time. Unifying the instruction cache and the BTB allows a program to flexibly use the available storage as needed without the limitations of a fixed partitioning. Alternatively, the BTB and the instruction cache could be organized in such away that the instruction cache tags are no longer required; hence, their area and power overhead can be saved. Instruction re-ordering attempts to densely pack frequently used instruction sequences in order to improve the locality in instruction cache and BTB accesses. Finally, compiler-generated hints can improve the instruction cache performance by guiding the hardware to wisely use the limited resources.

8.2.1 Hardware Prefetching (Hardware)

Instruction cache misses have a severe impact on the processor performance and energy efficiency as they cause the front-end to stall until the missing instructions are available. If an instruction cache is smaller than the working set, misses are inversely proportional to the cache size. Hence, a smaller instruction cache will typically cause additional performance loss. Instruction prefetching can reduce the performance impact of these misses. Instruction prefetching speculatively initiates a memory access for an instruction cache line, bringing the line into the cache (or a prefetching buffer) before the processor requests the instructions. Prefetching from the second level cache or even the main memory can hide the instruction cache miss penalties, but only if initiated sufficiently far ahead in advance of the current program counter.

Most modern processors only support very basic hardware sequential prefetchers. With a sequential or stream-based prefetcher, one or more sequential cache lines after the currently requested one are prefetched [103, 77]. Stream prefetching only helps with misses on sequential instructions. An alternative approach is to initiate prefetches for cache lines on the predicted path of execution [21]. The advantage of such a scheme is that it can prefetch potentially useful instructions even for non-sequential access patterns as long as branch prediction is sufficiently accurate.

As discussed in Section 4.1.2, BLISS supports efficient execution-based prefetching using the contents of the BBQ. The BBQ decouples basic block descriptor accesses from fetching the associated instructions. The predictor typically runs ahead, even when the instruction cache experiences temporary stalls due to a cache miss or when the instruction queue is full. The contents of the BBQ provide an early, yet accurate view into the instruction address stream and are used to lookup further instructions in the instruction cache. Prefetches are initiated when a potential miss is identified. BLISS also improves prediction accuracy (see Section 5.2.2), which makes the execution-based prefetching scheme even

more effective. Prefetching, if not accurate, leads to additional L2-cache accesses that can increase the L2-cache power dissipation.

8.2.2 Unified Instruction Cache and BTB (Hardware)

Programs exhibit different behaviors with respect to the instruction cache and BTB utilization. While some programs stress the instruction cache and are susceptible to its size (e.g., `rasta` from MediaBench), other programs depend more on the BTB capacity (e.g., `adpcm` from MediaBench). Even in a single program, different phases may exhibit different instruction cache and BTB access patterns. Being able to flexibly share the instruction cache and BTB resources could be valuable for those types of programs, especially when the hardware resources are limited.

The BLISS front-end can be configured with a unified instruction cache and BB-cache storage as both instructions and descriptors are part of the architecturally-visible binary code. Each line in the unified cache holds either a few basic block descriptors or a few regular instructions. The unified cache can be accessed by both the descriptor fetch and the instruction fetch units using a single access port. Instruction fetch returns multiple instructions per access (up to a full basic block) to the back-end pipeline and does not need to happen on every cycle. It only needs to occur when the IQ is not full and the BBQ is not empty. On the remaining cycles, we perform descriptor fetches using predicted BBD addresses. For the embedded processors we studied, sharing a single port for instruction and descriptor fetches had a negligible impact on performance.

On a conventional architecture, storing BTB and instruction cache entries in a single structure is more challenging as the same program counter is used to access both structures. This implies that extra information is required to be stored in the unified cache to differentiate between BTB and instruction entries. In addition, the two entries map to the same cache set, causing more conflicts. The BTB and instruction cache are also accessed more

frequently as basic block boundaries are not known until instruction decoding. Hence, sharing a single port is difficult.

8.2.3 Tagless Instruction Cache (Hardware)

In Section 6.2.2, we showed that we could eliminate the data access for all but the way that hits. Now we will focus on eliminating the instruction cache tags altogether (storage and access). BLISS provides an efficient way to build an instruction cache with no tag accesses by exploiting the tags checks performed on descriptor accesses. This improves instruction cache access time, reduces its energy consumption significantly, and eliminates the area overhead of tags. The new tagless instruction cache is organized as a direct mapped cache, with only the data component. Figure 8.2 illustrates the organization of this cache. For each basic block descriptor in the BB-cache, there is only one entry in the tagless instruction cache which can hold a certain number of instructions, 4 in our experiments. A flag bit is used in each descriptor in the BB-cache entry to indicate if the corresponding entry in the tagless instruction cache has valid instructions or not. This flag is initialized during BB-cache refill from L2-cache and is set after the instructions are fetched from the L2-cache and placed in the tagless instruction cache. Moreover, the flag that indicates if the entry in the tagless cache is valid or not can be used by the prefetching logic. This eliminates the need to probe the cache and improves the overall performance of the prefetcher.

The operation of the BLISS front-end with the tagless cache is very similar to what we explained in Section 4.1.1 except the way the instruction cache is accessed. On a BB-cache miss, the missing descriptors are retrieved from the L2-cache. At that stage, the instruction valid bits (**IV**) are initialized for those descriptors indicating that their associated instruction cache entries are invalid. The instruction fetch unit uses the valid bit to determine how to access the instruction cache. If the instruction valid bit is not set, the instructions are retrieved from the L2-cache using the instruction pointer available from the descriptor.

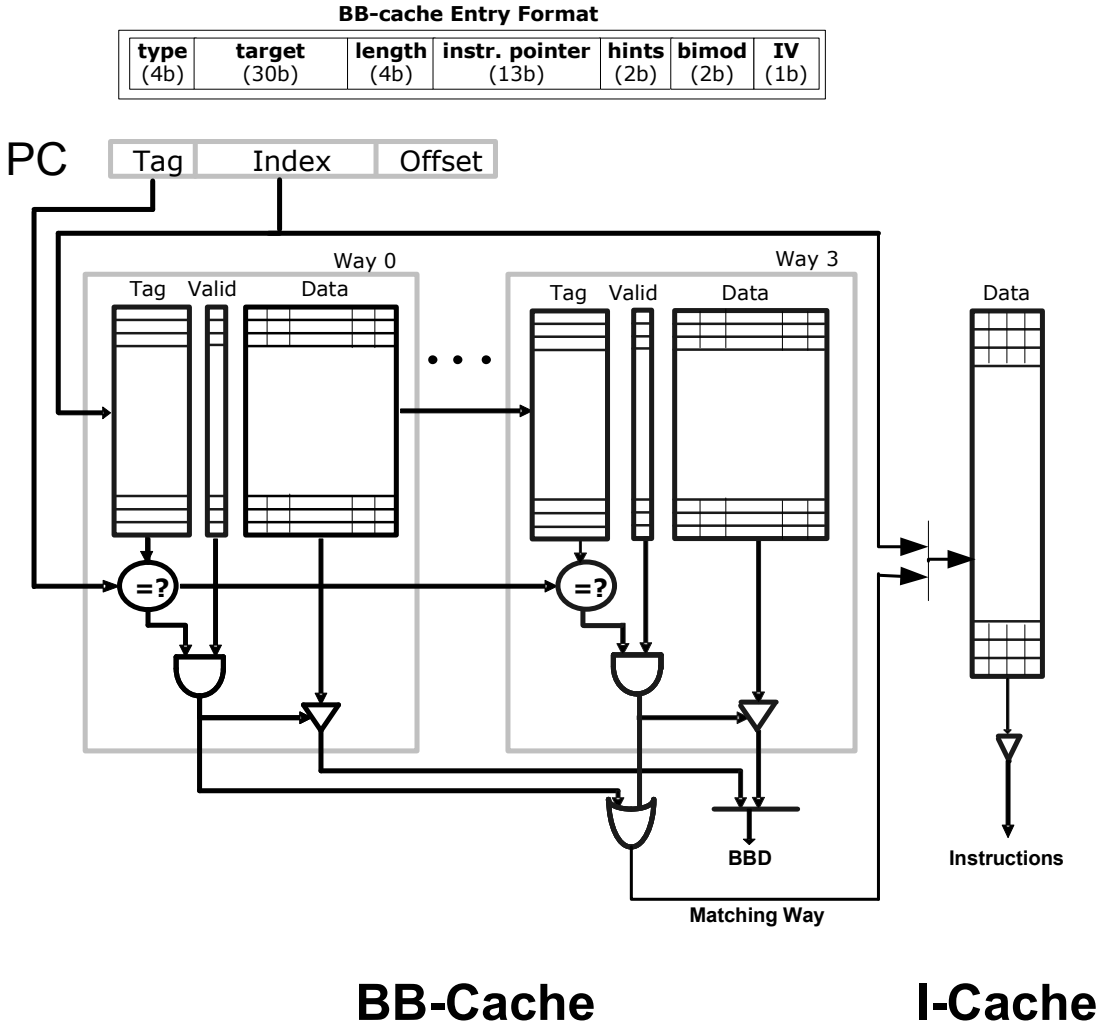


Figure 8.2: The organization of the tagless instruction cache with BLISS.

Once the instructions are retrieved and placed in the instruction cache, the valid bit for the corresponding descriptor is set. If the instruction valid bit is set, the instructions are retrieved from the instruction cache using the index field of the PC and the index of the matching BB-cache way. For basic blocks larger than 4 instructions, only the first four instructions are stored in the instruction cache. In the applications studied, 68% of the executed basic blocks include 4 instructions or less. Similar to the victim cache, we use a 4-entry fully associative cache to store the remaining instructions. This victim cache is accessed in a subsequent cycle and is tagged using the PC. In a case of a miss, the instructions are brought from the L2-cache.

Nevertheless, the tagless instruction cache has two limitations. First, once a BB-cache entry is evicted, the corresponding instruction cache entries become invalid. In addition, the virtual associativity and size of the instruction cache are now linked with that of the BB-cache and cannot be independently set. We can use an alternative approach for indexing the tagless cache to solve this limitation. We can determine the location in the instruction cache independently by an additional pointer field in the BB-cache format. This is similar to having a fully associative instruction cache, but with additional complexity in its management (keep track of LRU, etc).

8.2.4 Instruction Re-ordering (Software)

Code re-ordering at the basic block level is a mature method that tunes a binary to a specific instruction cache organization and improves hit rate and utilization. Re-ordering uses profiling information to guide placement of basic blocks within the code. The goal is to arrange closely executed blocks into chains that are laid out sequentially, hence increasing the number of instructions executed per cache line. The improved spatial locality reduces the miss rate for the instruction cache of a specific size. This implies that we can afford using a smaller cache without negatively impacting the performance.

Pettis and Hansen suggested a bottom-up block-level positioning algorithm [80]. In their approach, they split each procedure into two procedures, one with the commonly used basic blocks and one with the rarely used basic blocks (”fluff”). The infrequently executed code is replaced with a jump to the relocated code. Additionally, a jump is inserted at the end of the relocated code to transfer control back to the commonly executed code. Within each of the two procedures, a control-flow graph is used to from chains of basic blocks based on usage counts. The chains are then placed making fall through the likely case after a branch.

Basic block re-ordering is easily supported by BLISS using the explicit block descriptors. Blocks of instructions can be freely re-ordered in the code segment in any desired way as long as we update the instruction pointers in the corresponding block descriptors. Compared to re-ordering with conventional architectures, this provides two major benefits. First, there is no need to split the procedure or introduce additional jump instructions for control transfers between the commonly and the less commonly used code (fewer static and dynamic instructions). The pointers in the block descriptors handle control transfers automatically. Second, re-ordering basic blocks does not affect branch prediction accuracy for BLISS, as the vital information for speculation is included in the basic block descriptors available through the BB-cache (block type, target offset). On a conventional architecture, re-ordering blocks may change the number of BTB entries needed and the conflicts observed on BTB accesses.

8.2.5 Cache Placement Hints (Software)

Conventional caches are designed to be managed purely by hardware. Hardware must decide where to place the data and which data to evict during cache replacement. A consequence is that the cache resources may not be optimally utilized for a specific benchmark, leading to poor cache hit rate. Compilers and profile-based tools can help the processor

with selecting the optimal policies in order to achieve the highest possible performance using the minimal amount of hardware. Hints can indicate at which cache levels it is profitable to retain data based on their access frequency, excluding infrequent data from the first level cache. Hints can also guide the hardware placing data in the cache to avoid conflicts, or improve the cache replacement decisions by keeping data with higher chance of reuse.

A compiler can attach hints to executable code at various granularities, with every instruction, basic block, loop, function call, etc. BLISS provides a flexible mechanism for passing compiler-generated hints at the granularity of basic blocks. The last field of the basic block descriptor contains optional compiler-generated hints. Specifying hints at the basic block granularity allows for fine-grain information without increasing the length of all instruction encodings or requiring additional, out-of-band, instructions that carry the hints. Hence, hints can be communicated without modifying the conventional instruction stream or affecting static or dynamic instruction counts. Furthermore, since descriptors are fetched early in the pipeline, the hints can be useful with decisions with most pipeline stages, even before instructions are decoded.

We evaluate two types of software hints for the L1 instruction cache management. The first type indicates if a basic block should be excluded from the L1 instruction cache. We rely on prefetching, if enabled, to bring excluded blocks from the L2-cache when needed. Note that the hints are visible to the prefetcher, therefore, cache probing is not required for those blocks. A very simple heuristic based on profiling information is used to select which cache lines are cache-able. We exclude blocks with infrequently executed code and blocks that exhibit high miss rates. The second type of hints redistributes the cache accesses over the cache sets to minimize conflict misses. The hints are used as part of the address that indexes the cache. The 3 hint bits are concatenated with the index field of the address to form the new cache index field.

Front-End Parameters		
	XScale PXA270	
	Base	**BLISS**
Fetch Width	1 inst/cycle	1 BB/cycle
I-cache		
Regular	32 KBytes, 32-way, 32B Blocks, 2-cycle access	
Small	2 KBytes, 2-way, 32B Blocks, 2-cycle access	
BTB/BB-cache		
Regular	64-entry, 4-way	64-set, 4-way
Small	16-entry, 2-way	16-set, 2-way
BBQ	–	4 entries

Common Processor Parameters	
	XScale PXA270
Execution	single-issue, in-order with 1 INT & 1 FP unit
Predictor	256-entry bimod with 8 entry RAS
D-cache	32 KBytes, 4-way, 32B blocks, 1 port, 2-cycle access
L2-cache	128 KBytes, 4-way, 64B blocks, 1 port, 5-cycle access
Main memory	30-cycle access

Table 8.2: The microarchitecture parameters for base and BLISS processor configurations used for power and area optimization experiments.

8.3 Methodology

Table 8.2 summarizes the key architectural parameters for the base and BLISS processor configurations. Both are modeled after the Intel XScale PXA270 [42]. For fair energy comparisons, the base design uses serial instruction tag and data accesses. We have also performed experiments with a high-end embedded core comparable to the IBM PowerPC 750GX [40] and the achieved results are consistent.

For performance evaluation, we used similar tools to what we discussed in Section 5.1.2. We also used the same tools presented in Section 6.3 for energy evaluation. For performance, we report IPC, ignoring the fact that processors with smaller caches may be able to run at higher clock frequencies than processors with larger caches. We study 10 benchmarks form MediaBench suite and SPEC CPU2000 suite compiled at the -O2

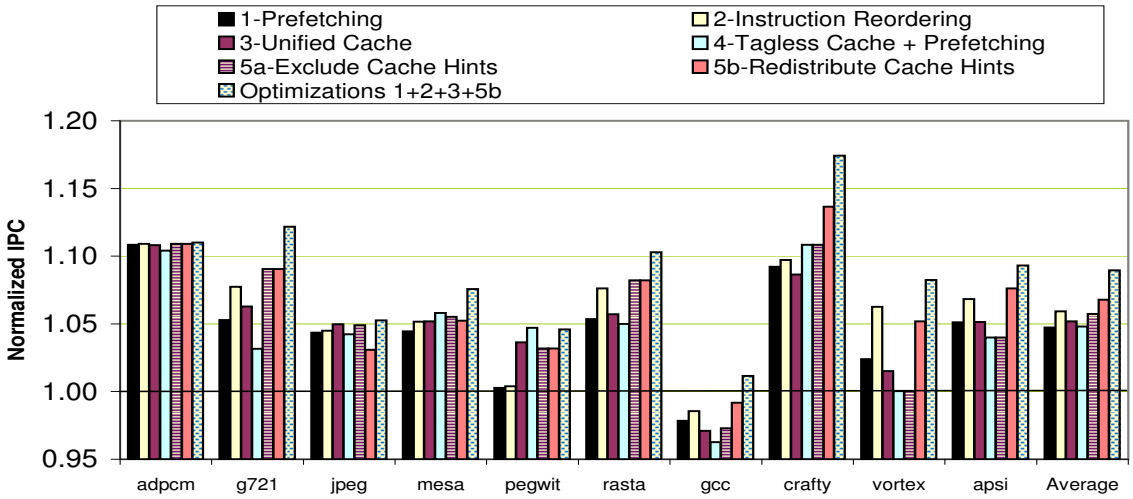


Figure 8.3: Normalized IPC for BLISS with the different front-end optimizations over the base. The BLISS design uses the small I-cache and BB-cache. The base design uses the regular I-cache and BTB. The 1.0 line presents the base design. Higher bars present better performance.

optimization level using gcc. The selected benchmarks have relatively high instruction cache or BTB miss rates.

8.4 Evaluation

This section presents the performance, cost, and energy evaluation analysis of BLISS using the different optimization techniques.

8.4.1 Performance Analysis

Figure 8.3 compares the IPC of BLISS with small caches and the various optimizations to that of the base design with large caches (IPC of 1.0). We only present a single combination of optimizations, the best performing one (prefetching + instruction re-ordering + unified cache + redistribute cache hints). For reference, the average normalized IPC for various other configurations is: 0.87 for the base design with small caches, **0.91** for the base design

with small caches and prefetching, and **0.99** for BLISS with small caches and no prefetching. It is important to notice that for all but one benchmark (`gcc`), all optimizations allow BLISS with small caches to reach the IPC of the base design with large caches. The design with the combined optimizations consistently outperforms the base with an average IPC improvement of 9%.

The analysis for the individual optimizations is the following. The advantages of instruction prefetching and re-ordering are consistent across all benchmarks. When combined, re-ordering reduces significantly the prefetching traffic. The unified cache is most beneficial for benchmarks that put pressure on the BTB (e.g., `jpeg`), but may also lead to additional conflicts (e.g., `crafty`). With the tagless cache, the performance greatly depends on the size of the basic blocks executed. For large basic blocks (`vortex` and `apsi`), performance degrades as the instruction cache cannot fit all the instructions in the block (limit of 4). Similarly, for programs with many small blocks (2 or less instructions as in g721), the instruction cache capacity is underutilized. The tagless cache performs best for programs with basic blocks of size 4 instructions like `pegwit`. It is also best to combine the tagless instruction cache with prefetching to deal with conflict misses. Software hints tend to provide a consistent improvement for all of the benchmarks. The redistribute cache hints achieve slightly better performance than the exclude cache hints.

To understand the effectiveness of each technique in reducing the performance impact of the small instruction cache, we look at the instruction cache miss rates for the different optimizations. Figure 8.4 presents the normalized number of instruction cache misses for BLISS with the different front-end optimizations over the base design with the small instruction cache. The reduction in instruction cache misses with prefetching, instruction re-ordering, and unified cache is consistent across most benchmarks with a 20% average. For the tagless instruction cache + prefetching, the decrease varies and largely depends on the basic block average size. Both of the software cache placement hints with prefetching

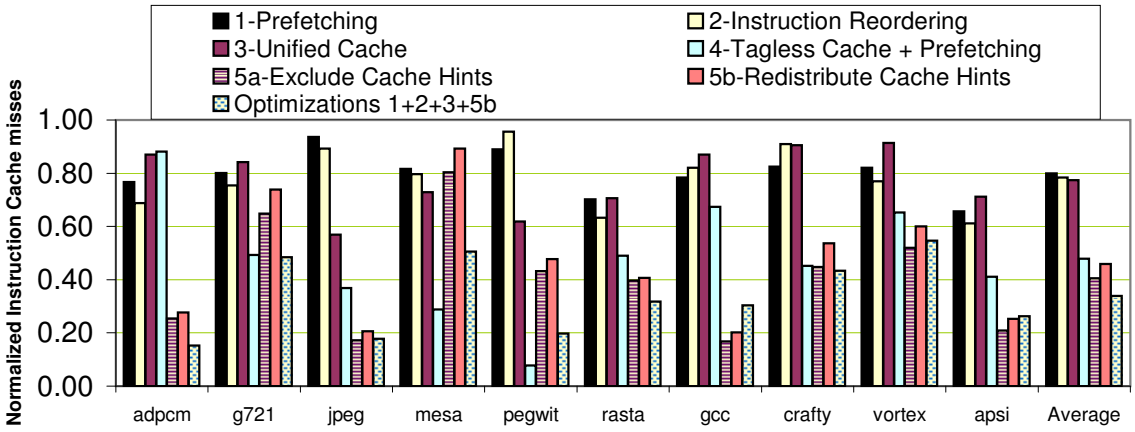


Figure 8.4: Normalized number of instruction cache misses for BLISS with the different front-end optimizations over the base. The BLISS design uses the small I-cache and BB-cache. The base design uses the small I-cache and BTB. Lower bars present better results.

	Power	**Area**	**Access Time**
Instruction Cache	8.4%	4.6%	50.7%
Predictor Tables	75.4%	47.5%	94.7%

Table 8.3: Normalized power dissipation, area, and access time for the small instruction cache and predictor tables over the large structures of the XScale configuration.

significantly reduce the number of cache misses with an average of 58%. Finally, the best combination of the optimizations (prefetching + instruction re-ordering + unified cache + redistribute cache hints) achieves 66% reduction.

8.4.2 Cost Analysis

Power and die area determine the cost to manufacture and package the chip. Table 8.3 summarizes the normalized power and area of the small front-end structures over the large structures for the XScale configuration. We also report the normalized access times for the small front-end structures. However, we ignore the fact that the processor with the small caches can run at higher clock frequency. The small instruction cache only dissipates 8.4% of the power dissipated by the large cache and uses only 4.6% of its area. The small

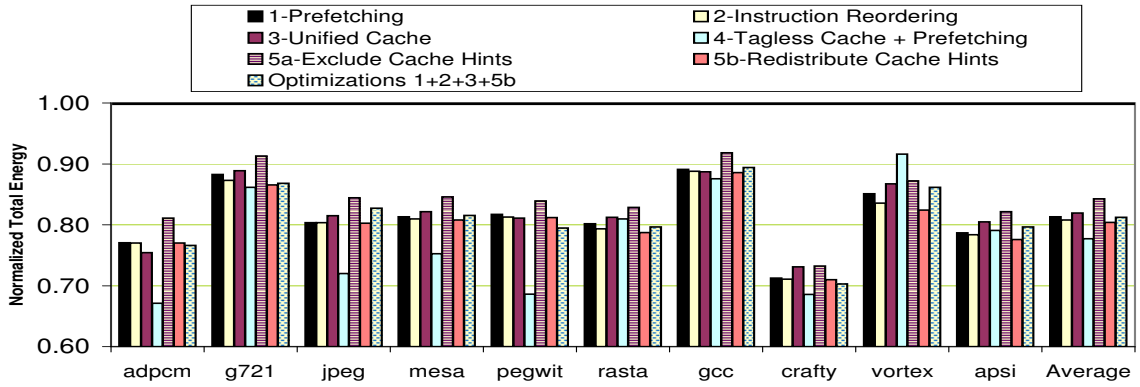


Figure 8.5: Normalized total energy comparison for BLISS with the different front-end optimizations over the base. The BLISS design uses the small I-cache and BB-cache. The base design uses the regular I-cache and BTB. The 1.0 line presents the base design. Lower bars present better results.

predictor tables dissipate 75.4% of the power dissipated by the larger structures and use only 47.5% of the area. The small instruction cache access time is also half of the access time for the large cache.

8.4.3 Total Energy Analysis

Figure 8.5 compares the total energy of BLISS with small caches and the various optimizations to that of the base design with large caches (energy of 1.0). Lower energy is better. For reference, the average total energy for other configurations is: 0.95 for the base design with small caches, **0.93** for the base design with small caches and prefetching, and **0.88** for BLISS with large caches.

With all optimizations, BLISS with small caches consumes less energy than the base with small or large caches. The combined optimizations lead to an energy consumption of 81%. The tagless instruction cache configuration provides significant energy benefits for several benchmarks (`adpcm, jpeg, mesa, pegwit`) as it eliminates redundant tag accesses. However, for `vortex`, the tagless instruction cache has the highest energy consumption. This is due to the fact that `vortex` has large basic blocks that will require to be

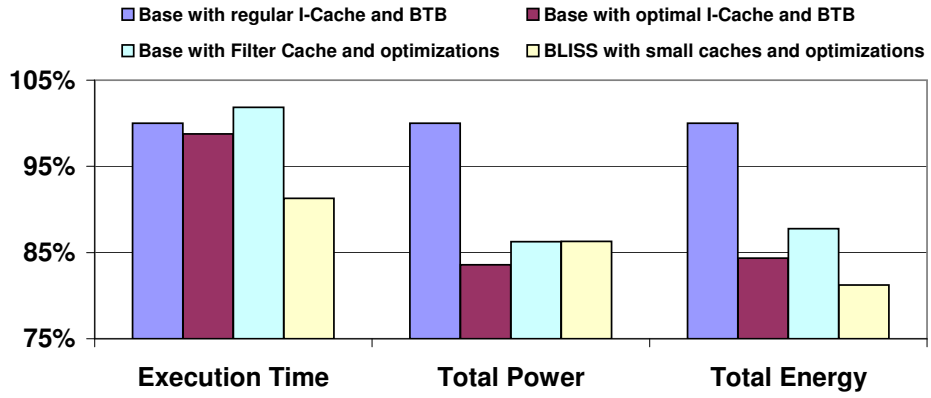


Figure 8.6: Average execution time, total power, and total energy consumption for base design (with large caches), base design (with optimal caches), base design (with Filter cache and a combination of front-end optimizations), and BLISS (with small caches and a combination of front-end optimizations). Lower bars present better results.

prefetched and placed in the small victim cache. For the remaining optimizations, energy consumption tracks the IPC behavior.

8.4.4 Comparison to Hardware-based Techniques

Many techniques have been proposed to save the front-end power without the need for a new ISA. One such example is the Filter cache design proposed by Kin et al. [53]. A Filter cache is a tiny cache introduced as the first level of memory in the instruction memory hierarchy.

Many of the front-end optimizations that are presented in Section 8.2 can also be implemented with a conventional instruction set using the Filter cache. Figure 8.6 summarizes the comparison between BLISS with the combined optimizations (unified cache + prefetching + instruction re-ordering + redistribute hints) to the base design with (Filter cache + prefetching + instruction re-ordering + selective caching hints). Note that similar front-end optimizations and cache sizes are used with both designs. The base XScale configuration with the full-sized instruction cache and BTB is shown as a reference. We also show the results for the base design with optimally-sized caches. We use a method similar to [97]

to quantify the amount of energy wasted due to sub-optimal cache sizes. A continuum of cache sizes and configurations are simulated. During each cycle, the cache with the lowest power from among those that hit is selected.

BLISS with the front-end optimizations provides similar total power reduction to the base design with Filter cache and the optimally-sized design (14% savings). It also provides similar total energy savings to the optimally-sized design (19% reduction). The small advantage is due to the more efficient access of instruction cache in the BLISS base model (see Chapter 6). More important, the power and energy savings do not lead to performance losses as it is the case for the base design with the Filter cache. BLISS provides a 9% performance improvement over the base design with large caches and a 12% performance improvement over the base design with Filter cache and the combined front-end optimizations. The performance advantage is due to two reasons. First, the efficient implementation of front-end optimizations mitigates the negative effects of the small instruction cache and BTB. Second, the block-aware architecture allows for higher prediction accuracy that provides the additional performance gains (see Chapter 5). In addition, BLISS provides 7% energy improvement over the base design with Filter cache and the combined front-end optimizations. Overall, BLISS with small caches and front-end optimizations improves upon the Filter cache with comparable front-end optimizations by offering similar power reduction at superior performance and energy consumption (12% performance and 7% total energy improvements).

We only report IPC for the BLISS and the Filter cache designs, ignoring the opportunity for performance gains if we exploit the faster access time of the small caches. By reducing the clock period, the BLISS and Filter cache designs can run at higher clock frequencies than processors with larger caches which will result in additional performance and energy improvements.

8.5 Related Work

Significant amount of front-end research focused on instruction cache optimizations of microprocessor-based systems because of the cache's high impact on system performance, cost, and power. The use of a tiny (Filter) cache to reduce power dissipation was proposed by Kin et al. [53]. Bellas et al. [10] proposed using a profile-aware compiler to map frequent loops into the Filter cache to reduce the performance overhead. BLISS provides similar power reduction as the Filter cache design and at the same time improves performance and energy consumption. Lee et al. [58] suggested using a tiny tagless loop cache with a controller that dynamically detect loops and fill the cache. The loop cache is an alternative to the first level of memory which is only accessed when a hit is guaranteed. Since the loop cache is not replacing the instruction cache, their approach improves the energy consumption with small performance, area, and total power overhead. Rose et al. [32] evaluated different small cache designs.

Many techniques have been proposed to reduce the instruction cache energy. Some of the techniques include way prediction [84], selective cache way access [2], sub-banking [30], and tag comparison elimination [75, 114], Other research has focused on reconfigurable caches [87, 113] where a subset of the ways in a set-associative cache or a subset of the cache banks are disabled during periods of modest cache activity to reduce power. Using a unified reconfigurable cache has also shown to be effective in providing greater levels of hardware flexibility [63]. Even though reconfigurable caches are effective in reducing energy consumption, they have negligible effect on reducing the peak power or the processor die area.

Many prefetching techniques have been suggested to hide the latency of cache misses. The simplest technique is the sequential prefetching [103, 77]. In this scheme, one or more sequential cache lines that follow the current fetched line are prefetched. History-based schemes [104, 39, 48] use the patterns of previous accesses to initiate the new prefetches.

The execution-based scheme has been proposed as an alternative approach [89, 21]. In this scheme, the prefetcher uses the predicted execution path to initiate accesses. Other types of prefetching schemes include the wrong path prefetching [82] and the software cooperative approach [61]. In the later one, the compiler inserts software prefetches for non-sequential misses. BLISS enables highly accurate execution-based prefetching using the contents of the BBQ.

Much research exists at exploring the benefit of code re-ordering [106]. Most of the techniques use a variation of the code positioning algorithm suggested by Pettis and Hansen [80]. Several researchers have also worked on using software-generated hints to improve the performance and power of caches [46, 68, 13, 41]. BLISS efficiently enables instruction re-ordering with no extra overhead and no impact on speculation accuracy. Moreover, the architecturally visible basic block descriptors allow communicating software hints without modifying the conventional instruction stream or affecting its instruction code footprint.

8.6 Summary

This chapter evaluated several front-end optimizations that improve the performance of embedded processors with small front-end caches. Small caches allow for an area and power efficient design but typically lead to performance challenges. The optimizations included instruction prefetching and re-ordering, selective caching, tagless instruction cache, and unified instruction and branch target caches. We built these techniques on top of the block-aware instruction set (BLISS) architecture that provides a flexible platform for both software and hardware front-end optimizations. The best performing combined optimizations (prefetching + unified cache + redistribute cache hints) allow an embedded processor with small front-end caches to be 9% faster and consume 14% less power and 19% less energy than a similar pipeline with large front-end structures. While some of the optimizations

can also be implemented with a conventional instruction set, they lead to lower performance benefits and are typically more complex. Overall, BLISS allows for low power and low cost embedded designs in addition to performance, energy, and code size advantages. Therefore, it can be a significant design option for embedded systems.

Chapter 9

Conclusions and Future Work

This dissertation examined the use of a block-aware instruction set architecture (BLISS) to address the front-end challenges. The theme of this expressive ISA is to allow software to assist the front-end hardware by providing architecture support for control-flow prediction and instruction delivery. BLISS defines basic block descriptors in addition to and separately from the actual instructions in each program. A descriptor describes the type of the control-flow operation that terminates the block, its potential target, and the number of instructions in the basic block. This information is sufficient to tolerate the latency of instruction accesses, regulate the use of prediction structures, and direct instruction prefetching. The architecture also provides a flexible communication mechanism that software can use to provide hardware with critical information about instruction fetching and control-flow.

We evaluated the BLISS ISA across a wide spectrum of processors and demonstrated that BLISS provides efficient ISA-level support for front-end optimizations that target all of the processor efficiency metrics. We showed that BLISS significantly improves both the performance and the energy consumption of high-end superscalar processors. For embedded processors where code size and cost are of greater concern, BLISS allows significant reduction in the code size, total power, and die area in addition to the performance and

energy gains. BLISS also compares favorably to hardware-only techniques built on top of conventional ISAs with no software-support.

Overall, this work demonstrated the potential of delegating hardware functions in superscalar processors to software using an expressive instruction set. The result is a processor with simpler hardware structures that performs better and consumes less energy than aggressive hardware designs that operate on conventional instruction sets.

The primary contributions of this dissertation are:

- We defined the block-aware ISA that provides basic block descriptors in addition to and separately from the actual instructions in each program. The BLISS provides accurate information for control-flow prediction and instruction prefetching without fetching and parsing the actual instruction stream. BLISS also provides a versatile mechanism for conveying compiler-generated hints at basic block granularity without modifying the conventional instruction stream or affecting its instruction code footprint.

- We proposed a decoupled front-end organization based on the BLISS ISA. The new front-end replaces the BTB with a descriptor's cache. It uses the information available in descriptors to improve control-flow accuracy, implement guided instruction prefetching, and reduce the energy used for control-flow prediction and instruction delivery. We demonstrated that the new architecture improves upon conventional superscalar designs by 20% in performance and 16% in energy. We also showed that it outperforms hardware-only approach for decoupled front-ends by 13% and 7% for performance and energy respectively. These benefits are robust across a wide range of architectural parameters.

- We evaluated the use of BLISS for embedded processor designs. We developed a set of code size optimizations that utilize the ISA mechanism to provide code size reduction of 40%. Unlike alternative proposals that tradeoff performance or energy

consumption for code density, we showed that BLISS-based embedded designs provide 10% performance and 21% energy advantages in addition to the improved code size.

- We developed and evaluated a set of hardware and software techniques for low cost front-ends for embedded systems. The optimization techniques target the size and power consumption of instruction caches and predictor tables. We showed that the decoupling features of BLISS and the ability to provide software hints allow for embedded designs that use minimally sized, power efficient caching and predictor structures, without sacrificing performance.

Future Work

We have made a great progress in evaluating and exploring the benefits of using the BLISS ISA to assist the hardware in dealing with the front-end challenges. However, there are still opportunities and challenges to follow up with. In the future, we intend to explore the potentials of the BLISS ISA in several ways:

- We would like to examine the use of the BLISS ISA with a variety of hardware and software optimization techniques such as trace caches [94], hyperblocks [62], and multiple-branch predictors [100]. Those techniques are rather orthogonal to the BLISS ISA as one can form streams or traces on top of the basic blocks in BLISS. BLISS simplifies the hardware for all of them as it eliminates the need to detect basic blocks on the fly.

- We showed that BLISS provides a very flexible mechanism to communicate software hints. We evaluated two different usages to improve code density and prediction accuracy. In the future, we would like to explore some of the other uses that we presented in Section 3.2. In particular, we are interested in using the hints to scale

the processor resources based on the program needs to save energy and power at the back-end of the processor.

- With the recent shift to multi-threaded and multi-core architectures, we would like to explore the use of BLISS for such designs. Exploiting the BBDs for those designs will facilitate fine-grain parallelism, allow better management for shared caches and resources, and help in tightly coupled multi-core/multi-thread synchronization.

The philosophy of the BLISS ISA is to allow the software to assist the hardware in dealing with difficult challenges. While BLISS mainly focuses on the front-end part of the processor, it would be interesting to try to use the same philosophy to deal with the problems at the back-end of the processor.

Bibliography

[1] Vikas Agarwal, M.S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock Rate vs IPC: The End of Road for Conventional Microarchitectures. In *The Proceedings of Intl. Symposium on Computer Architecture*, pages 248–259, Vancouver, BC, Canada, June 2000.

[2] David H. Albonesi. Selective Cache Ways: On-Demand Cache Resource Allocation. In *The Proceedings of Intl. Symposium on Microarchitecture*, pages 248–259, Haifa, Israel, November 1999.

[3] Anon. Tandem Makes a Good Thing Better. *Electronics*, pages 34–38, April 1986.

[4] Juan L. Aragon, Jose Gonzalez, and Antonio Gonzalez. Power-Aware Control Speculation Through Selective Throttling. In *The Proceedings of Intl. Symposium on High-Performance Computer Architecture*, pages 103–112, Anaheim, CA, February 2003.

[5] R. Iris Bahar and Srilatha Manne. Power and Energy Reduction via Pipeline Balancing. In *The Proceedings of Intl. Symposium Computer Architecture*, pages 218–229, Goteborg, Sweden, July 2001.

[6] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A Transparent Dynamic Optimization System. In *The Proceedings of Conference on Programming*

Language Design and Implementation, pages 1–12, Vancouver, BC, Canada, June 2000.

[7] Amirali Baniasadi and Andreas Moshovos. Branch Predictor Prediction: A Power-Aware Branch Predictor for High-Performance Processors. In *The Proceedings of Intl. Conference on Computer Design*, pages 458–461, Freiburg, Germany, September 2002.

[8] Amirali Baniasadi and Andreas Moshovos. SEPAS: a Highly Accurate Energy-Efficient Branch Predictor. In *The Proceedings of Intl. Symposium on Low Power Electronics and Design*, pages 38–43, Newport Beach, CA, August 2004.

[9] Luiz Andre Barroso, Kourosh Gharachorloo, and Edouard Bugnion. Memory System Characterization of Commercial Workloads. In *The Proceedings of Intl. Symposium on Computer Architecture*, pages 3–14, Barcelona, Spain, October 1998.

[10] Nikolaos Bellas, Ibrahim Hajj, Constantine Polychronopoulos, and George Stamoulis. Energy and Performance Improvements in Microprocessor Design using a Loop Cache. In *The Proceedings of Intl. Conference on Computer Design*, pages 378–383, Washington, DC, October 1999.

[11] Brian N. Bershad, Dennis Lee, Theodore H. Romer, and J. Bradley Chen. Avoiding Conflict Misses Dynamically in Large Direct-Mapped Caches. In *The Proceedings of Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, pages 158–170, San Jose, CA, October 1994.

[12] Arpaad Beszedes, Rudolf Ferenc, and Tibor Gyimothy. Survey of Code-Size Reduction Methods. *ACM Computing Surveys*, 35(3):223–267, September 2003.

[13] Kristof Beyls and Erik H. D'Hollander. Generating Cache Hints for Improved Program Efficiency. *Journal of Systems Architecture*, 51(4):223–250, April 2005.

[14] Bryan Black, Bohuslav Rychlik, and John Paul Shen. The Block-Based Trace Cache. In *The Proceedings of Intl. Symposium on Computer Architecture*, pages 196–207, Atlanta, GA, May 1999.

[15] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *The Proceedings of Intl. Symposium on Computer Architecture*, pages 83–94, Vancouver, BC, Canada, June 2000.

[16] Doug Burger and Todd M. Austin. Simplescalar Tool Set, Version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[17] Brad Calder and Dirk Grunwald. Fast and Accurate Instruction Fetch and Branch Prediction. In *The Proceedings of Intl. Symposium on Computer Architecture*, pages 2–11, Chicago, IL, April 1994.

[18] Brad Calder and Dirk Grunwald. Reducing Branch Costs via Branch Alignment. In *The Proceedings of Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, pages 242–251, San Jose, CA, October 1994.

[19] Po-Yung Chang, Eric Hao, Tse-Yu Yeh, and Yale Patt. Branch Classification: a New Mechanism for Improving Branch Predictor Performance. In *The Proceedings of Intl. Symposium on Microarchitecture*, pages 22–31, San Jose, CA, November 1994.

[20] Daniel Chaver, Luis Piuel, Manuel Prieto, Francisco Tirado, and Michael C. Huang. Branch Prediction on Demand: an Energy-Efficient Solution. In *The Proceedings of Intl. Symposium on Low Power Electronics and Design*, pages 390–395, Seoul, Korea, August 2003.

[21] I-Cheng K. Chen, Chih-Chieh Lee, and Trevor N. Mudge. Instruction Prefetching Using Branch Prediction Information. In *The Proceedings of Intl. Conference on Computer Design*, pages 593–601, San Jose, CA, October 1997.

[22] Keith D. Cooper and Nathaniel McIntosh. Enhanced Code Compression for Embedded RISC Processors. In *The Proceedings of Conference on Programming Language Design and Implementation*, pages 139–149, Atlanta, GA, May 1999.

[23] Saumya K. Debray, William Evans, Robert Muth, and Bjorn De Sutter. Compiler Techniques for Code Compaction. *ACM Transactions on Programming Languages and Systems*, 22(2):378–415, March 2000.

[24] Ashutosh S. Dhodapkar and James E. Smith. Managing Multi-Configuration Hardware via Dynamic Working Set Analysis. In *The Proceedings of Intl. Symposium on Computer Architecture*, pages 233–244, Anchorage, AK, May 2002.

[25] David R. Ditzel and Hubert R. McLellan. Branch Folding in the CRISP Microprocessor: Reducing Branch Delay to Zero. In *The Proceedings of Intl. Symposium on Computer Architecture*, pages 2–8, Pittsburgh, PA, June 1987.

[26] Simonjit Dutta and Manoj Franklin. Control Flow Prediction with Tree-Like Subgraphs for Superscalar Processors. In *The Proceedings of Intl. Symposium on Microarchitecture*, pages 258–263, Ann Arbor, MI, November 1995.

[27] Mongkol Ekpanyapong, Pinar Korkmaz, and Hsien-Hsin S. Lee. Choice Predictor for Free. In *The Proceedings of Asia-Pacific Computer Systems Architecture Conference*, pages 399–413, Beijing, China, September 2004.

[28] Wesley M. Felter, Tom W. Keller, Michael D. Kistler, Charles Lefurgy, Karthick Rajamani, Ram Rajamony, Freeman L. Rawson, Bruce A. Smith, and Eric Van Hensbergen. On The Performance and Use of Dense Servers. *IBM Journal of Research and Development*, 47(5/6):671–688, September 2003.

[29] Daniel Holmes Friendly, Sanjay Jeram Patel, and Yale N. Patt. Alternative Fetch and Issue Techniques from the Trace Cache Mechanism. In *The Proceedings of Intl. Symposium on Microarchitecture*, pages 24–33, Research Triangle Park, NC, December 1997.

[30] Kanad Ghose and Milind B. Kamble. Reducing Power in Superscalar Processor Caches Using Subbanking, Multiple Line Buffers and Bit-Line Segmentation. In *The Proceedings of Intl. Symposium on Low Power Electronics and Design*, pages 70–75, San Diego, CA, August 1999.

[31] Ricardo Gonzalez and Mark Horowitz. Energy Dissipation in General Purpose Microprocessors. *IEEE Journal of Solid-State Circuits*, 31(9):1277–1284, September 1996.

[32] Ann Gordon-Ross, Susan Cotterell, and Frank Vahid. Tiny Instruction Caches for Low Power Embedded Systems. *ACM Transactions on Embedded Computing Systems*, 2(4):449–481, November 2003.

[33] Thomas R. Gross and John L. Hennessy. Optimizing Delayed Branches. In *The Proceedings of Annual Workshop on Microprogramming*, pages 114–120, Palo Alto, CA, October 1982.

[34] Dirk Grunwald, Artur Klauser, Srilatha Manne, and Andrew Pleszkun. Confidence Estimation for Speculation Control. In *The Proceedings of Intl. Symposium on Computer Architecture*, pages 122–131, Barcelona, Spain, July 1998.

[35] Rajiv Gupta and Chi-Hung Chi. Improving Instruction Cache Behavior by Reducing Cache Pollution. In *The Proceedings of Conference on Supercomputing*, pages 82–91, New York, NY, November 1990.

[36] Ashok Halambi, Aviral Shrivastava, Partha Biswas, Nikil Dutt, and Alex Nicolau. An Efficient Compiler Technique for Code Size Reduction using Reduced Bit-Width ISAs. In *The Proceedings of Conference on Design, Automation and Test in Europe*, pages 402–408, Paris, France, March 2002.

[37] Eric Hao, Po-Yung Chang, Marks Evers, and Yale N. Patt. Increasing the Instruction Fetch Rate via Block-Structured Instruction Set Architectures. In *The Proceedings of Intl. Symposium on Microarchitecture*, pages 191–200, Paris, France, December 1996.

[38] John L. Henning. SPEC CPU2000: Measuring Performance in the New Millennium. *IEEE Computer*, 33(7):28–35, July 2000.

[39] Wei-Chung Hsu and James E. Smith. A Performance Study of Instruction Cache Prefetching Methods. *IEEE Transactions on Computers*, 47(5):497–508, May 1998.

[40] IBM Corporation. *IBM PowerPC 750GX RISC Microprocessor User's Manual*, February 2004.

[41] Intel Corporation. *Intel Itanium Architecture Software Developers Manual. Revision 2.0*, December 2001.

[42] Intel Corporation. *Intel PXA27x Processor Family Developer's Manual*, October 2004.

[43] Anoop Iyer and Diana Marculescu. Power Aware Microarchitecture Resource Scaling. In *The Proceedings of Conference on Design, Automation and Test in Europe*, pages 190–196, Munich, Germany, March 2001.

[44] Erik Jacobsen, Eric Rotenberg, and J. E. Smith. Assigning Confidence to Conditional Branch Predictions. In *The Proceedings of Intl. Symposium on Microarchitecture*, pages 142–152, Paris, France, December 1996.

[45] Quinn Jacobson, Eric Rotenberg, and James E. Smith. Path-Based Next Trace Prediction. In *The Proceedings of Intl. Symposium on Microarchitecture*, pages 14–23, Research Triangle Park, NC, December 1997.

[46] Prabhat Jain, Srinivas Devadas, Daniel Engels, and Larry Rudolph. Software-Assisted Cache Replacement Mechanisms for Embedded Systems. In *The Proceedings of Intl. Conference on Computer-Aided Design*, pages 119–126, San Jose, CA, November 2001.

[47] Daniel A. Jimnez and Calvin Lin. Dynamic Branch Prediction with Perceptrons. In *The Proceedings of Intl. Symposium on High-Performance Computer Architecture*, pages 197–206, Nuevo Leone, Mexico, January 2001.

[48] Doug Joseph and Dirk Grunwald. Prefetching using Markov Predictors. In *The Proceedings of Intl. Symposium on Computer Architecture*, pages 252–263, Denver, CO, June 1997.

[49] Stephan Jourdan, Lihu Rappoport, Yoav Almog, Mattan Erez, Adi Yoaz, and Ronny Ronen. Extended Block Cache. In *The Proceedings of Intl. Symposium on High-Performance Computer Architecture*, pages 61–70, Toulouse, France, January 2000.

[50] Mauricio Breternitz Jr. and Roger Smith. Enhanced Compression Techniques to Simplify Program Decompression and Execution. In *The Proceedings of Intl. Conference on Computer Design*, pages 170–176, Austin, TX, October 1997.

[51] Vinod Kathail, Michael S. Schlansker, and B. Ramakrishna Rau. HPL PlayDoh Architecture Specification. Technical Report HPL-93-80, HP Labs, February 1994.

[52] R.E Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2):24–36, March 1999.

[53] Johnson Kin, Munish Gupta, and William H. Mangione-Smith. The Filter Cache: An Energy Efficient Memory Structure. In *The Proceedings of Intl. Symposium on Microarchitecture*, pages 184–193, Research Triangle Park, NC, December 1997.

[54] Kevin D. Kissell. MIPS16: High-Density MIPS for the Embedded Market. Technical report, Silicon Graphics MIPS, 1997.

[55] Arvind Krishnaswamy and Rajiv Gupta. Profile Guided Selection of ARM and Thumb Instructions. In *The Proceedings of Joint Conference on Languages, Compilers and Tools for Embedded Systems*, pages 56–64, Berlin, Germany, June 2002.

[56] Jeremy Lau, Stefan Schoenmackers, Timothy Sherwood, and Brad Calder. Reducing Code Size With Echo Instructions. In *The Proceedings of Intl. Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 84–94, San Jose, CA, October 2003.

[57] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: a Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *The Proceedings of Intl. Symposium on Microarchitecture*, pages 330–335, Research Triangle Park, NC, December 1997.

[58] Lea Hwang Lee, Bill Moyer, and John Arends. Instruction Fetch Energy Reduction Using Loop Caches for Embedded Applications with Small Tight Loops. In *The Proceedings of Intl. Symposium on Low Power Electronics and Design*, pages 267–269, San Diego, CA, August 1999.

[59] Charles Lefurgy, Peter Bird, I-Cheng Chen, and Trevor Mudge. Improving Code Density Using Compression Techniques. In *The Proceedings of Intl. Symposium on Microarchitecture*, pages 194–203, Research Triangle Park, NC, December 1997.

[60] Haris Lekatsas, Jorg Henkal, and Wayne Wolf. Code Compression for Low Power Embedded System Design. In *The Proceedings of Conference on Design Automation*, pages 294–299, Los Angeles, CA, June 2000.

[61] Chi-Keung Luk and Todd C. Mowry. Architectural and Compiler Support for Effective Instruction Prefetching: a Cooperative Approach. *ACM Transactions on Computer Systems*, 19(1):71–109, February 2001.

[62] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. Effective Compiler Support for Predicated Execution Using the Hyperblock. In *The Proceedings of Intl. Symposium on Microarchitecture*, Portland, OR, December 1992.

[63] Afzal Malik, Bill Moyer, and Dan Cermak. A Low Power Unified Cache Architecture Providing Power and Performance Flexibility. In *The Proceedings of Intl. Symposium on Low Power Electronics and Design*, pages 241–243, Rapallo, Italy, July 2000.

[64] Srilatha Manne, Artur Klauser, and Dirk Grunwald. Pipeline Gating: Speculation Control for Energy Reduction. In *The Proceedings of Intl. Symposium on Computer Architecture*, pages 132–141, Barcelona, Spain, June 1998.

[65] Rajit Manohar and Mark Heinrich. The Branch Processor Architecture. Technical Report CSL-TR-1999-1000, Cornell Computer Systems Laboratory, November 1999.

[66] Alain J. Martin. Towards an Energy Complexity of Computation. *Information Processing Letters*, 77(2-4):181–187, February 2001.

[67] Cathy May, Ed Silha, Rick Simpson, and Hank Warren. *The PowerPC Architecture*. Morgan Kaufman, San Mateo, CA, 1994.

[68] Scott McFarling. Program Optimization for Instruction Caches. In *The Proceedings of Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–191, Boston, MA, April 1989.

[69] Scott McFarling. Combining Branch Predictors. Technical Report TN-36, DEC WRL, June 1993.

[70] Stephen Melvin and Yale Patt. Enhancing Instruction Scheduling with a Block-structured ISA. *Intl. Journal on Parallel Processing*, 23(3):221–243, June 1995.

[71] Pierre Michaud, Andre Seznec, and Stephan Jourdan. Exploring Instruction-Fetch Bandwidth Requirement in Wide-Issue Superscalar Processors. In *The Proceedings of Intl. Conference on Parallel Architectures and Compilation Techniques*, pages 2–10, Newport Beach, CA, October 1999.

[72] Pierre Michaud, Andre Seznec, and Richard Uhlig. Trading Conflict and Capacity Aliasing in Conditional Branch Predictors. In *The Proceedings of Intl. Symposium on Computer Architecture*, pages 292–303, Denver, CO, June 1997.

[73] Andreas Moshovos, Dionisios N. Pnevmatikatos, and Amirali Baniasadi. Slice-Processors: An Implementation of Operation-Based Prediction. In *The Proceedings of Intl. Conference on Supercomputing*, pages 321–334, Sorrento, Italy, June 2001.

[74] Paramjit S. Oberoi and Gurindar S. Sohi. Parallelism in the Front-End. In *The Proceedings of Intl. Conference on Computer Architecture*, pages 230–240, San Diego, CA, June 2003.

[75] Ramesh Panwar and David Rennels. Reducing the Frequency of Tag Compares for Low Power I-Cache Design. In *The Proceedings of Intl. Symposium on Low Power Design*, pages 57–62, Dana Point, CA, April 1995.

[76] Dharmesh Parikh, Kevin Skadron, Yan Zhang, Marco Barcella, and Mircea R. Stan. Power Issues Related to Branch Prediction. In *The Proceedings of Intl. Symposium on High-Performance Computer Architecture*, pages 233–244, Boston, MA, February 2002.

[77] Gi-Ho Park, Oh-Young Kwon, Tack-Don Han, Shin-Dug Kim, and Sung-Bong Yang. An Improved Lookahead Instruction Prefetching. In *The Proceedings of High-Performance Computing on the Information Superhighway*, pages 712–715, Seoul, South Korea, May 1997.

[78] Sanjay Jeram Patel, Marius Evers, and Yale N. Patt. Improving Trace Cache Effectiveness with Branch Promotion and Trace Packing. In *The Proceedings of Intl. Symposium on Computer Architecture*, pages 262–271, Barcelona, Spain, June 1998.

[79] Yale Patt. Requirements, Bottlenecks, and Good Fortune: Agents for Microprocessor Evolution. *The Proceedings of the IEEE*, 89(11):1153–1159, November 2001.

[80] Karl Pettis and Robert C. Hansen. Profile Guided Code Positioning. In *The Proceedings of Conference on Programming Language Design and Implementation*, pages 16–27, White Plains, NY, June 1990.

[81] Richard Phelan. Improving ARM Code Density and Performance. Technical report, Advanced RISC Machines Ltd, June 2003.

[82] Jim Pierce and Trevor Mudge. Wrong-Path Instruction Prefetching. In *The Proceedings of Intl. Symposium on Microarchitecture*, pages 165–175, Paris, France, December 1996.

[83] Dionisios N. Pnevmatikatos, Manoj Franklin, and Gurindar S. Sohi. Control Flow Prediction for Dynamic ILP Processors. In *The Proceedings of Intl. Symposium on Microarchitecture*, pages 153–163, Austin, TX, December 1993.

[84] Michael D. Powell, Amit Agarwal, T. N. Vijaykumar, Babak Falsafi, and Kaushik Roy. Reducing Set-Associative Cache Energy via Way Prediction and Selective Direct-Mapping. In *The Proceedings of Intl. Symposium on Microarchitecture*, pages 54–65, Austin, TX, December 2001.

[85] Alex Ramirez, Josep L. Larriba-Pey, and Mateo Valero. Branch Prediction Using Profile Data. In *The Proceedings of EuroPar Conference*, pages 386–393, Manchester, UK, August 2001.

[86] Alex Ramirez, Oliverio J. Santana, Josep L. Larriba-Pey, and Mateo Valero. Fetching Instruction Streams. In *The Proceedings of Intl. Symposium on Microarchitecture*, pages 371–382, Istanbul, Turkey, November 2002.

[87] Parthasarathy Ranganathan, Sarita Adve, and Norman P. Jouppi. Reconfigurable Caches and their Application to Media Processing. In *The Proceedings of Intl. Symposium on Computer Architecture*, pages 214–224, Vancouver, BC, Canada, June 2000.

[88] Glenn Reinman, Todd Austin, and Brad Calder. A Scalable Front-End Architecture for Fast Instruction Delivery. In *The Proceedings of Intl. Symposium on Computer Architecture*, pages 234–245, Atlanta, GA, May 1999.

[89] Glenn Reinman, Brad Calder, and Todd Austin. Fetch Directed Instruction Prefetching. In *The Proceedings of Intl. Symposium on Microarchitecture*, pages 16–27, Haifa, Israel, November 1999.

[90] Glenn Reinman, Brad Calder, and Todd Austin. Optimizations Enabled by a Decoupled Front-End Architecture. *IEEE Transactions on Computers*, 50(40):338–335, April 2001.

[91] Glenn Reinman, Brad Calder, and Todd M. Austin. High Performance and Energy Efficient Serial Prefetch Architecture. In *The Proceedings of Intl. Symposium on High-Performance Computing*, pages 146–159, Kansai Science City, Japan, May 2002.

[92] Edward M. Riseman and Caxton C. Foster. The Inhibition of Potential Parallelism by Conditional Jumps. *IEEE Transactions on Computer*, pages 1405–1411, December 1972.

[93] Ronny Ronen, Avi Mendelson, Konrad Lai, Shih-Lien Lu, Fred Pollack, and John P. Shen. Coming Challenges in Microarchitecture and Architecture. *The Proceedings of the IEEE*, 89(3):325–340, March 2001.

[94] Eric Rotenberg, Steve Bennet, and James E. Smith. Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching. In *The Proceedings of Intl. Symposium on Microarchitecture*, pages 24–34, Austin, TX, December 1996.

[95] Oliverio J. Santana, Alex Ramirez, Josep L. Larriba-Pey, and Mateo Valero. A Low-Complexity Fetch Architecture for High-Performance Superscalar Processors. *ACM Transactions on Architecture and Code Optimization*, 1(2):220–245, June 2004.

[96] Michael S. Schlansker and B. Ramakrishna Rau. EPIC: An Architecture for Instruction-Level Parallel Processors. Technical Report HPL-99-111, HP Labs, February 2000.

[97] John S. Seng and Dean M. Tullsen. Architecture-Level Power Optimization - What Are the Limits? *Journal of Instruction-Level Parallelism 7*, 7(3):1–20, January 2005.

[98] Andre Seznec, Stephen Felix, Venkata Krishnan, and Yiannakis Sazeides. Design Tradeoffs for the Alpha EV8 Conditional Branch Predictor. In *The Proceedings of Intl. Symposium on Computer Architecture*, pages 295–306, Anchorage, AK, May 2002.

[99] Andre Seznec and Antony Fraboulet. Effective Ahead Pipelining of Instruction Block Address Generation. In *The Proceedings of Intl. Conference on Computer Architecture*, pages 241–252, San Diego, CA, June 2003.

[100] Andre Seznec, Stephan Jourdan, Pascal Sainrat, and Pierre Michaud. Multiple-Block Ahead Branch Predictors. In *The Proceedings of Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, pages 116–127, Cambridge, MA, October 1996.

[101] L. E. Shar and E. S. Davidson. A Multiminiprocessor System Implemented through Pipelining. *IEEE Computer Magazine*, pages 42–51, February 1974.

[102] Premkishore Shivakumar and Norman P. Jouppi. Cacti 3.0: An Integrated Cache Timing, Power, Area Model. Technical Report 2001/02, Compaq Western Research Laboratory, August 2001.

[103] Jim E. Smith and Wei-Chung Hsu. Prefetching in Supercomputer Instruction Caches. In *The Proceedings of Conference on Supercomputing*, pages 588–597, Minneapolis, MN, November 1992.

[104] Viji Srinivasan, Edward S. Davidson, Gary S. Tyson, Mark J. Charney, and Thomas R. Puzak. Branch History Guided Instruction Prefetching. In *The Proceedings of Intl. Symposium on High-Performance Computer Architecture*, pages 291–300, Nuevo Leone, Mexico, January 2001.

[105] Jared Stark, Paul Racunas, and Yale N. Patt. Reducing the Performance Impact of Instruction Cache Misses by Writing Instructions into the Reservation Stations Out-of-Order. In *The Proceedings of Intl. Symposium on Microarchitecture*, pages 34–43, Research Triangle Park, NC, December 1997.

[106] Hiroyuki Tomiyama and Hiroto Yasuura. Code Placement Techniques for Cache Miss Rate Reduction. *ACM Transactions on Design Automation of Electronic Systems*, 2(4):410–429, October 1997.

[107] Nigel Topham and Kenneth McDougall. Performance of the Decoupled ACRI-1 Architecture: the Perfect Club. In *The Proceedings of Intl. Conference on High-Performance Computing and Networking*, pages 472–480, Milan, Italy, May 1995.

[108] James L. Turley. Thumb Squeezes ARM Code Size. Technical Report 4, Microprocessor Report, March 1995.

[109] Robert G. Wedig and Marc A. Rose. The Reduction of Branch Instruction Execution Overhead Using Structured Control Flow. In *The Proceedings of Intl. Symposium on Computer Architecture*, pages 119–125, Ann Arbor, MI, June 1984.

[110] Andrew Wolfe and Alex Chanin. Executing Compressed Programs on An Embedded RISC Architecture. In *The Proceedings of Intl. Symposium on Microarchitecture*, pages 81–91, Portland, OR, December 1992.

[111] Tse-Yu Yeh and Yale N. Patt. A Comprehensive Instruction Fetch Mechanism for a Processor Supporting Speculative Execution. In *The Proceedings of Intl. Symposium on Microarchitecture*, pages 129–139, Portland, OR, December 1992.

[112] Tse-Yu Yeh and Yale N. Patt. A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History. In *The Proceedings of Intl. Symposium on Computer Architecture*, pages 257–266, Philadelphia, PA, December 1993.

[113] Chuanjun Zhang, Frank Vahid, and Walid Najjar. A Highly Configurable Cache for Low Energy Embedded Systems. *ACM Transactions on Embedded Computing Systems*, 4(2):363–387, May 2005.

[114] Youtao Zhang and Jun Yang. Low Cost Instruction Cache Designs for Tag Comparison Elimination. In *The Proceedings of Intl. Symposium on Low Power Electronics and Design*, pages 266–269, Seoul, Korea, August 2003.

[115] Ahmad Zmily, Earl Killian, and Christos Kozyrakis. Improving Instruction Delivery with a Block-Aware ISA. In *The Proceedings of EuroPar Conference*, pages 530–539, Lisbon, Portugal, August 2005.

[116] Ahmad Zmily and Christos Kozyrakis. Energy-Efficient and High-Performance Instruction Fetch using a Block-Aware ISA. In *The Proceedings of Intl. Symposium on Low Power Electronics and Design*, pages 36–41, San Diego, CA, August 2005.

[117] Ahmad Zmily and Christos Kozyrakis. Block-Aware Instruction Set Architecture. *ACM Transactions on Architecture and Code Optimization*, 3(3):327–357, September 2006.

[118] Ahmad Zmily and Christos Kozyrakis. Simultaneously Improving Code Size, Performance, and Energy in Embedded Processors. In *The Proceedings of Conference on Design, Automation and Test in Europe*, pages 224–229, Munich, Germany, March 2006.