Block-Aware Instruction Set Architecture

AHMAD ZMILY and CHRISTOS KOZYRAKIS
Stanford University

Instruction delivery is a critical component for wide-issue, high-frequency processors since its bandwidth and accuracy place an upper limit on performance. The processor front-end accuracy and bandwidth are limited by instruction-cache misses, multicycle instruction-cache accesses, and target or direction mispredictions for control-flow operations. This paper presents a block-aware instruction set (BLISS) that allows software to assist with front-end challenges. BLISS defines basic block descriptors that are stored separately from the actual instructions in a program. We show that BLISS allows for a decoupled front-end that tolerates instruction-cache latency, facilitates instruction prefetching, and leads to higher prediction accuracy.

Categories and Subject Descriptors: C.1.1 [**Processor Architectures**]: Single Data Stream Architectures—*RISC/CISC, VLIW architectures*

General Terms: Design, Performance

Additional Key Words and Phrases: Instruction set architecture, instruction fetch, basic block, software hints, branch prediction, decoupled architecture

1. INTRODUCTION

Effective instruction delivery is vital for superscalar processors operating at high clock frequencies [Patt 2001; Ronen et al. 2001]. The rate and accuracy at which instructions enter the processor pipeline set an upper limit to sustained performance. Consequently, wide-issue designs place increased demands on the processor *front-end*, the engine responsible for control-flow prediction and instruction fetching. The front-end must mitigate three basic performance 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 multicycle instruction cache accesses in high-frequency designs that introduce additional uncertainty about the existence and direction of branches within the instruction stream. The cost of these detractors for a superscalar processor is up to 48% performance loss and 21% increase in total energy consumption [Zmily and Kozyrakis 2005].

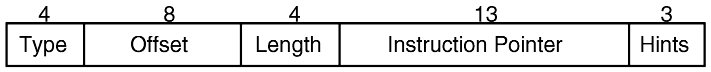
Authors' address: Ahmad Zmily and Christos Kozyrakis, Electrical Engineering Department, Stanford University, Palo Alto, CA 94305; email: zmily@stanford.edu, christos@ee.stanford.edu.

To address these issues in a high performance, yet energy and complexity effective, way, we propose a block-aware instruction set architecture (BLISS) [Zmily et al. 2005]. 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 BLISS instruction set allows the processor front-end to access the software-defined basic block descriptors through a small cache that replaces the branch target buffer (BTB). The descriptors' cache decouples control-flow speculation from instruction-cache accesses. Hence, the instruction-cache latency is no longer in the critical path of accurate prediction. The fetched descriptors can be used to accurately prefetch instructions and 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.

Moreover, the architecturally visible basic block descriptors 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 cache footprint. Compiler hints can be used to improve the program code density, to improve performance by guiding the hardware, or to reduce complexity and power consumption by replacing hardware structures. In this paper, we use the mechanism to implement branch prediction hints that lead to additional prediction accuracy, performance, and energy savings for BLISS-based processors.

The specific contributions of the paper are:

- We describe a block-aware ISA (BLISS) that provides basic block descriptors in addition to and separately from the actual instructions. We describe a set of optimizations during code generation that eliminate the effect of basic block descriptors on code size and allow for up to 18% code density improvements over conventional architectures.
- We propose a decoupled front-end for the block-aware instruction set. The new design replaces the hardware-based branch target buffer with a simple cache for the software-generated basic block descriptors. We show that the BLISS-based design leads to 20% performance improvement and 14% total energy savings over a processor with a conventional front-end.
- We show that the BLISS-based design compares favorably to a processor with an aggressive front-end that forms extended basic blocks and implements similar performance and energy optimizations using hardware-only techniques[Reinman et al. 1999b, 2001]. The static information available in block descriptors allows the BLISS-based design to achieve a better balance between under and overspeculation in the front-end and achieve 13% higher performance and 7% additional total energy savings.
- We demonstrate that the benefits of the block-aware instruction set are robust across a wide range of design parameters for superscalar processors, such



Type : basic block type (type of terminating branch)
- fall-through (FT)
- backward conditional branch (BR_B)
- forward conditional branch (BR_F)
- jump (J)
- jump-and-link (JAL)
- jump register (JR)
- jump-and-link register (JALR)
- call return (RET)
- zero-overhead loop (LOOP)

Offset: displacement for PC-relative branches and jumps

Length: number of instruction in the basic block (0..15)

Instruction pointer :
address of the 1st instruction in the block (bits [14:2])
bits [31:15] are stored in the TLB

Hints: optional compiler-generated hints
used for static branch hints in this study

Fig. 1. The 32-bit basic block descriptor format in BLISS.

as issue width (four-way and eight-way), instruction-cache size and latency, branch target cache size, and associativity.

Overall, this work demonstrates 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.

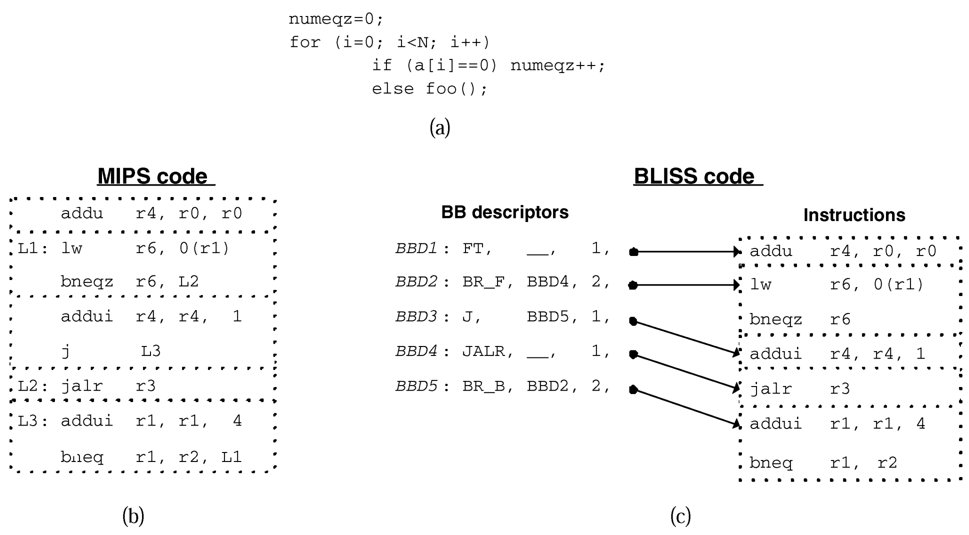
The remainder of this paper is organized as follows. In Section 2, we present the block-aware instruction set. Section 3 describes the modified front-end that exploits the basic block information available in the proposed ISA. Section 4 overviews the experimental methodology used in this paper. In Section 5, we analyze and compare the benefits of our proposal by measuring performance and energy consumption for a wide range of processor configurations. In Section 6 we discuss the related research that this work is based on. Section 7 provides a summary and highlights future work.

2. BLOCK-AWARE INSTRUCTION SET

Our proposal for addressing the front-end performance is based on a *block-aware instruction set (BLISS)* that explicitly describes basic blocks [Zmily et al. 2005]. A basic block (*BB*) 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.

2.1 Instruction Set

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 1 presents the format of a basic block descriptor (*BBD*). Each BBD defines the type of the control-flow operation that terminates the block. The `LOOP` type is a zero-overhead loop construct similar to that in the PowerPC ISA [May et al. 1994]. The BBD also includes an offset field to be used for blocks ending with a branch or a jump with PC-relative addressing. The



```
numeqz=0;
for (i=0; i<N; i++)
        if (a[i]==0) numeqz++;
        else foo();
```

(a)

MIPS code

```
    addu   r4, r0, r0
L1: lw     r6, 0(r1)
    bneqz  r6, L2
    addui  r4, r4,  1
    j       L3
L2: jalr   r3
L3: addui  r1, r1,  4
    bneq   r1, r2, L1
```

(b)

BLISS code

```
BBD1 : FT,    __,    1,      addu    r4, r0, r0
BBD2 : BR_F, BBD4, 2,      lw      r6, 0(r1)
BBD3 : J,     BBD5, 1,      bneqz  r6
BBD4 : JALR,  __,    1,      addui   r4, r4, 1
BBD5 : BR_B, BBD2, 2,      jalr    r3
                              addui   r1, r1, 4
                              bneq    r1, r2
```

(c)

Fig. 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.

actual instructions in the basic block are identified by the pointer to the first instruction and the length field. The BBD only provides the lower bits ([14:2]) of the instruction pointer; bits ([31:15]) are stored in the TLB. The last BBD field contains optional compiler-generated hints, which we discuss in detail in Section 2.3. The overall BBD length is 32 bits.

Figure 2 presents an example program that counts the number of zeros in array a and calls `foo()` for each nonzero element. With a RISC ISA like MIPS, the program requires eight instructions (Figure 2b). The four control-flow operations define five basic blocks. All branch conditions and targets are defined by the branch and jump instructions. With the BLISS equivalent of MIPS (Figure 2c), the program requires five basic block descriptors and seven 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`.

BLISS treats each basic block as an atomic unit of execution, which is similar to the block-structured ISA [Hao et al. 1996; Melvin and Patt 1995]. When a basic block is executed, either every instruction in the block is retired or none of the instructions in the block are 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 Section 5, a limit of eight load/stores per block does not cause a noticeable change in code size or performance. Other precise exceptions are handled as described in Melvin and Patt [1995].

BLISS compresses the control-flow address space for programs as each BBD corresponds to four to eight instructions, on average (see Section 5.2). 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. 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 because of the dense PCs. A thorough evaluation of this issue is left for future work.

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 2 could be the destination register of a previous load instruction.

2.2 Code Size

A naive generation of binary code for BLISS would introduce a basic block descriptor for every four to eight instructions. Hence, BLISS would lead to significant code size increase over a conventional instruction set such as MIPS. However, BLISS allows for aggressive code optimizations that exceed the capacity for the additional descriptors and lead to overall code density advantages. As shown in the example in Figure 2, all jump instructions can be removed as they no longer provide any additional information. Moreover, careful ISA encoding allows us to eliminate a portion of the conditional branch instructions that perform a simple test (equal/not-equal to zero) to a register value produced within the same basic block. Such a simple test can be encoded within the producing instruction. Hence, by introducing condition testing versions for a few

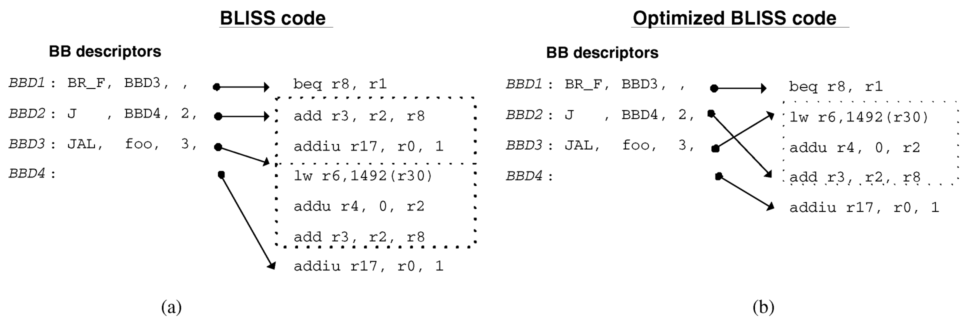


Fig. 3. Example to illustrate the block-subsetting code optimization. (a) Original BLISS code. (b) BLISS code with the block-subsetting optimization. For illustration purposes, the instruction pointers in basic block descriptors are represented with arrows.

common instructions, such as addition, subtraction, and logical operations, we can eliminate a significant portion of branches. Note that the branch target is provided by the basic block descriptor and does not need to be provided by any regular instruction.

BLISS also facilitates the removal of repeated instruction sequences [Cooper and McIntosh 1999]. All instructions in a basic block can be eliminated, if the exact sequence of the instructions can be found elsewhere in the binary. 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 3 presents an example. The two instructions in the second basic block in the original code appear in the exact order toward 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 and loop setup and stack handling. Instruction similarity is also improved because BLISS stores branch offsets in the block descriptors and not in regular instructions. Nevertheless, block-subsetting can affect the locality exhibited in instruction cache accesses. To avoid negative performance impact, block-subsetting can be applied selectively.

2.3 Software Hints

BLISS provides a versatile mechanism for conveying software-generated hints to the processor pipeline. The use of compiler-generated hints is a popular method for overcoming bottlenecks with modern processors [Schlansker and Rau 1999]. 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.

2.3.1 *Potential Uses of Hints Mechanism.* The last field in each basic block descriptor in Figure 1 provides a flexible mechanism for communicating compiler-generated hints at basic block granularity. 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 nonexhaustive list of potential uses of the hints mechanism.

- Code density: The hints field can be used to aggressively interleave 16 and 32-bit instructions at basic block granularity without the overhead of additional instructions for switching between 16 and 32-bit modes [Halambi et al. 2002]. 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 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. Our preliminary experiments indicate that interleaving 16 and 32-bit blocks provides an additional 20% in code size reduction (on top of the 18% reduction presented in Section 5.1), while reducing the performance advantages of BLISS by less than 2%.
- 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 [Melvin and Patt 1995].
- 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 [Moshovos et al. 2001].

- 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 three bits allocated in the format in Figure 1. Hence, there can be a trade-off between the benefits from using the hints and the drawbacks from increasing the BBD length. Exploring this trade-off is an interesting direction for future work, but it is beyond the scope of this paper.

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.

2.3.2 *Case Study: Branch Prediction Hints.* To illustrate the usefulness of the hints mechanism, we use it to implement software hints for branch prediction [Ramirez et al. 2001]. The compiler uses three 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 branches, 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 [Yeh and Patt 1993]).
- 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 [McFarling 1993; Yeh and Patt 1993]).

We use the third bit to provide a default taken or not-taken static prediction outcome. With nonstatically 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 a hybrid predictor and calculate confidence.

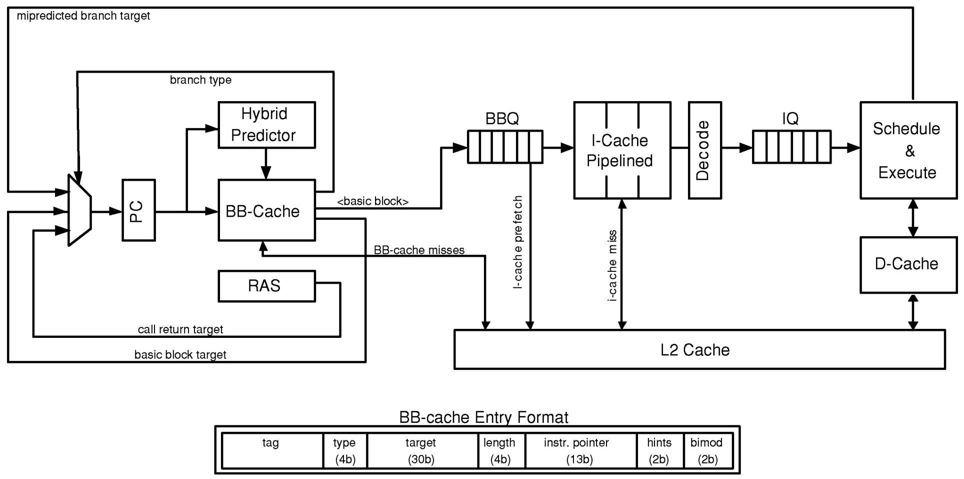


Fig. 4. A decoupled front-end for a superscalar processor based on the BLISS ISA.

3. DECOUPLED FRONT-END FOR THE BLOCK-AWARE ISA

This section presents the superscalar processor front-end for the BLISS ISA and its benefits over a state-of-the-art approach that creates instruction blocks without ISA support.

3.1 Front-End Architecture

The BLISS ISA suggests a superscalar front-end that fetches BBDs and the associated instructions in a decoupled manner. Figure 4 presents a BLISS-based front-end that replaces 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 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 nonsequential 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 nonsequential 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, 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 BB-cache integrates a simple bimodal predictor (see Figure 4). 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 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.

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 the register-based jumps, the offset field is available after the first execution of the basic block. The BB-cache stores eight sequential BBDs per cache line. Long BB-cache lines exploit spatial locality in descriptor accesses and reduce the storage overhead for tags. This is not the case with the BTB that describes a single target address per cache line (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,[1] the BTB and BB-cache implementations require the same number of SRAM bits.

The BLISS front-end alleviates all shortcomings of a conventional front-end. The BBQ decouples control-flow prediction from instruction fetching. Multicycle 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). Since the PC in the BLISS ISA always points to basic block descriptors (i.e., a control-flow instruction), 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. The contents of the BLISS BBQ also provide an early view into the instruction address stream and can be used for instruction prefetching and hide instruction cache misses [Chen and Baer 1994].

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 because of reduced interference at the predictor and the BTB (see Section 5.3).

The availability of basic block descriptors also allows for energy optimizations in the processor front-end. Each basic block exactly defines the number of instructions needed from the instruction cache. Using segmented word lines [Ghose and Kamble 1999] 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 first access the tags for a

[1]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.

set-associative instruction cache, and in subsequent cycles, access the data only in the way that hits [Reinman et al. 2002]. 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.

3.2 Hardware versus Software Basic Blocks

A decoupled front-end similar to the one in Figure 4 can be implemented without the ISA support provided by BLISS. The FTB design [Reinman et al. 1999b, 2001] describes the latest of such design. The design uses a *fetch target buffer (FTB)* as an enhanced basic block BTB [Yeh and Patt 1992]. 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 [Reinman et al. 1999a]. A fetch block may include several strongly biased, not-taken branches. 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.

The FTB is accessed each cycle 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, 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, 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.

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 from biased not-taken to unbiased or taken, when 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. [2] The branch terminating the old block will need to train new predictor entries. Moreover, the frequency of such problematic events can be significant because of the finite capacity of the FTB. As new FTB entries are created, older, yet useful, entries may be evicted because of 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.

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 because of block creation.[3] 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, in Section 5.3, we demonstrate that the BLISS front-end actually fetches more *useful* instructions per control-flow prediction. 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 Section 5.6.1, we show that the BB-cache achieves good

[2]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.
[3]There are still mispredictions because of incorrect prediction of the direction or target of the branch terminating a basic block, but there are no mispredictions because of discovering or splitting fetch blocks.

hit rates for a variety of sizes and consistently outperforms an equally sized FTB.

4. METHODOLOGY

For our experimental evaluation, we study 12 benchmarks from the SpecCPU2000 suite using their reference datasets [Henning 2000]. The rest of the benchmarks in the SpecCPU2000 suite perform similarly 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. We generate BLISS executables using a static binary translator, which can handle arbitrary programs from high-level languages like C or Fortran. The translator consists of three passes. The first pass parses the binary executable, identifies all basic blocks, and creates the basic block descriptors. The second pass implements the code size optimizations discussed in Section 2.2. The final pass outputs the new BLISS executable. The generation of BLISS executable could also be done using a transparent, dynamic compilation framework [Bala et al. 2000].

Our simulation framework is based on the Simplescalar/PISA 3.0 toolset [Burger and Austin 1997], 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 [Brooks et al. 2000] (nonideal, aggressive conditional clocking). Energy consumption was calculated for a 0.10-μm process with a 1.1-V 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 [Shivakumar and Jouppi 2001].

Table I presents the microarchitecture parameters used with simulations. The base model reflects a superscalar processor with a conventional BTB and no decoupling. The pipeline of the base model consists of six pipeline stages: fetch, decode, issue, execute, write-back, and commit stage. The FTB model represents the decoupled front-end that creates fetch blocks in hardware [Reinman et al. 1999b]. The BLISS model represents the modified front-end introduced in Section 3 with software-defined basic blocks and the BB-cache as the BTB replacement. The three models differ only in the front-end. All of the other parameters are identical. 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. We simulate both eight-way and four-way execution cores with all three models. The eight-way execution core is generously configured to reduce back-end stalls so that any front-end performance differences are obvious. The four-way execution core represents a more practical implementation point. 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

Table I. The Microarchitecture Parameters for the Simulations[a]

	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

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
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 KB, 4-way, 64B blocks, 2 ports, 2-cycle access pipelined
L2 cache	1 MB, 8-way, 128B blocks, 1 port, 12-cycle access, 4-cycle repeat rate
Main memory	100-cycle access

[a]The common parameters apply to all three models (base, FTB, BLISS). Certain parameters vary between eight-way and four-way processor configurations. The table shows the values for the eight-way core with the values for the four-way core in parenthesis.

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.

For the FTB and BLISS front-ends, we implement instruction prefetching based on the contents of the FTQ and BBQ buffers [Reinman et al. 1999b]. When the instruction cache is stalled because of 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 prefetch data go 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. EVALUATION

In this section, we present the evaluation results that demonstrate the advantages of the BLISS front-end over the FTB and the base designs.

5.1 Static Code Size

Before we proceed with performance analysis, we first discuss code density. Figure 5 presents the percentage of code size increase for BLISS over the MIPS ISA that it is based on. Direct translation (*Naive* bar) of MIPS code introduces one basic block descriptor every four instructions and leads to an average code

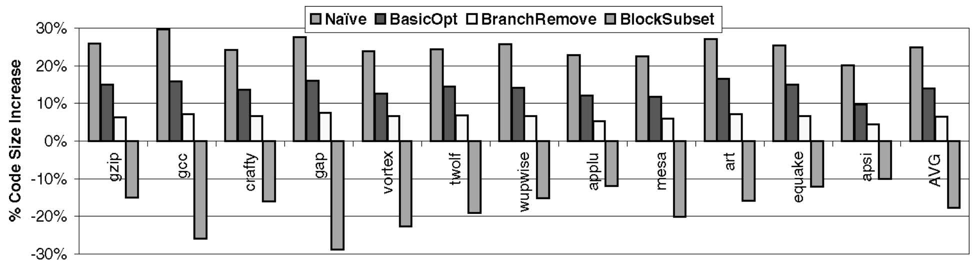


Fig. 5. Static code size increase for BLISS over the MIPS ISA. Positive increase means that the BLISS executables are larger. Negative increase means that the BLISS executables are smaller.

Table II. Dynamic 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%

size increase of 25%. Basic optimization (*BasicOpt* bar) reduces the average code size increase to 14%. The basic optimization includes the removal of redundant jump instructions (see example in Figure 2).

The *BranchRemove* bar in Figure 5 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. Finally, block-subsetting optimization allows the BLISS code size to be *18% smaller* than the original MIPS code. As shown in the example in Figure 3, all instructions for a basic block can be removed if the exact instructions appear in the stream of other instructions.

5.2 Dynamic BLISS Statistics

Table II presents the dynamic distribution of descriptor types for BLISS. Most programs include a significant number of fall-through descriptors. For integer applications, this is mostly because of the large number of labels in the original MIPS code (potential targets of control-flow operations). For floating-point applications, this is mostly because of 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.

Table III. Dynamic Distribution of BBD Lengths 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%
wupwisee	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	50.7%	25.1%	9.5%	14.7%

Table III shows the dynamic distribution of descriptor lengths. It is interesting to notice that, even for integer applications, an average of 40% of the executed basic blocks include more than four instructions. This implies that making one prediction for every four instructions fetched from the instruction cache is often wasteful. 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).

5.3 Performance Comparison

Figure 6 compares the IPC achieved for the eight-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 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 9). The remaining of this section provides a detailed analysis of BLISS performance advantage. We only present data for a representative subset of benchmarks, but the average refers to all 12 benchmarks in this study.

Figure 7 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 average). The FTB advantage is because of 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 fetches the descriptors from the L2

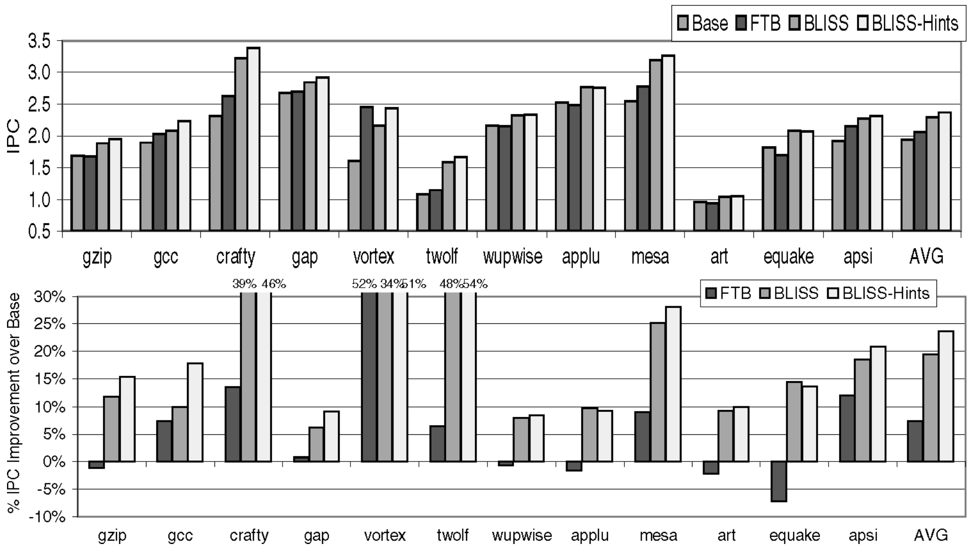


Fig. 6. Performance comparison for the 8-way processor configuration with the base, FTB, and BLISS front-ends. The top graph presents raw IPC and the bottom one shows the percentage of IPC improvement over the base for FTB and BLISS.

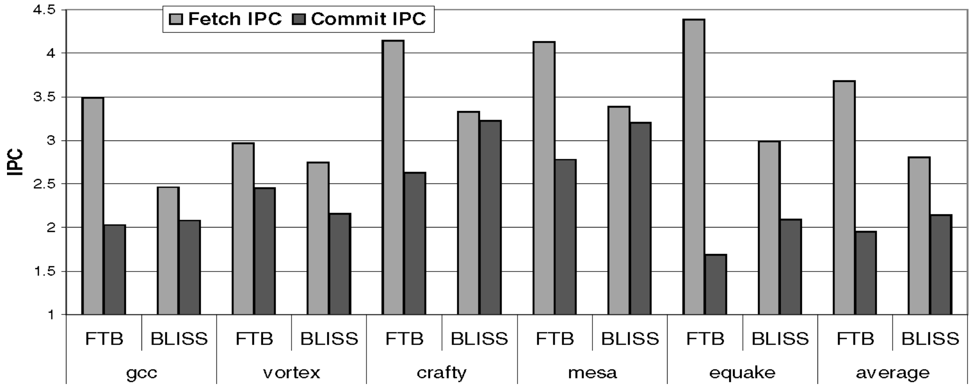


Fig. 7. Fetch and commit IPC for the eight-way processor configuration with the FTB and BLISS front-ends. We present data for a representative subset of benchmarks, but the average refers to all benchmarks in this study. For BLISS, we present the data for the case without static branch hints.

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.

Figure 8 further explains the basic performance advantage of BLISS over the base and FTB designs. Compared to the base, BLISS reduces by 41% the

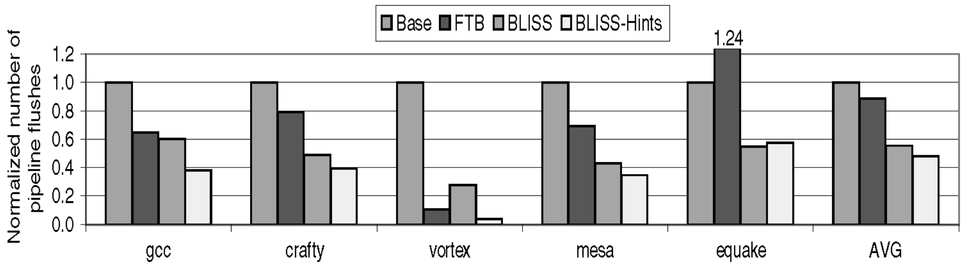


Fig. 8. Normalized number of pipeline flushes because of direction and target mispredictions for the eight-way processor configuration with the base, FTB, BLISS, and BLISS-HINTS front-ends. We present data for a representative subset of benchmarks, but the average refers to all benchmarks in this study.

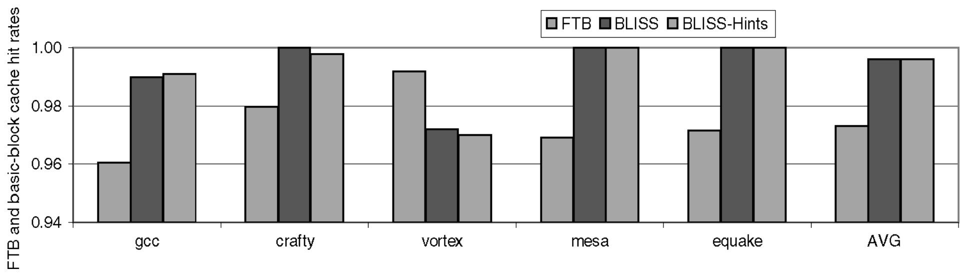


Fig. 9. FTB and BB-cache hit rates for the eight-way processor configuration. We present data for a representative subset of benchmarks, but the average refers to all benchmarks in this study.

number of pipeline flushes because of target and direction mispredictions. These flushes have a severe performance impact as they empty the full processor pipeline. Flushes in BLISS are slightly more expensive than in the base design because of the longer pipeline, but they are less frequent. The BLISS advantage is because of 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. Although the FTB front-end achieves a higher prediction accuracy compared to the base design, 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 because of longer training and increased interference. When static branch hints are in use (*BLISS-Hints*), the branch prediction accuracy is improved on average by 1.2%, from 93.4% without hints to 94.6% with hints, and results in an additional 10% reduction in the number of pipeline flushes.

Figure 9 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 by 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 FTB as the BB-cache avoids block splitting and recreation that occur when branches change

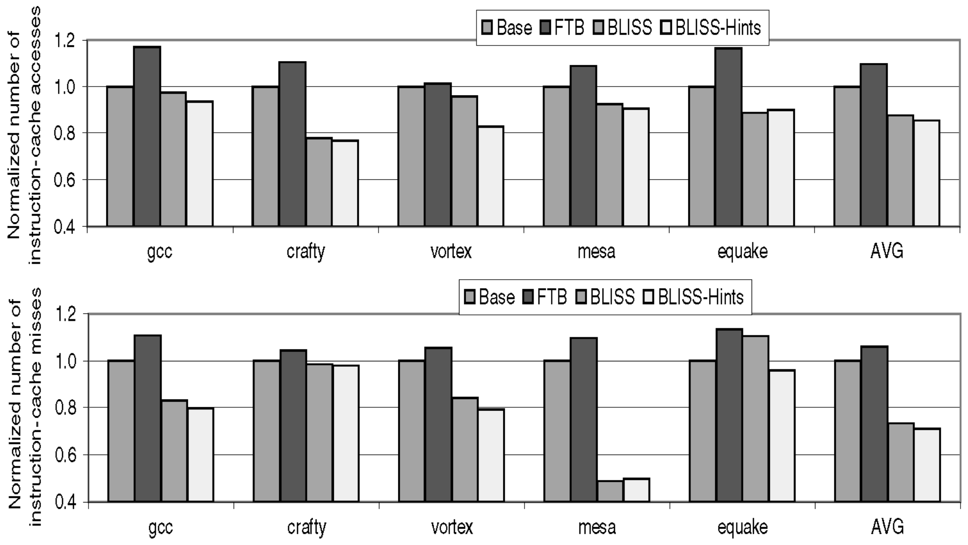


Fig. 10. Instruction cache comparison for the eight-way processor configuration with the base, FTB, BLISS, and BLISS-HINTS front-ends. The top graph compares the normalize number of instruction-cache accesses; the bottom one shows the normalized number of instruction-cache misses. We present data for a representative subset of benchmarks, but the average refers to all benchmarks in this study.

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).

Figure 10 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 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 because of the more accurate prediction as shown in Figure 8 and the reduced number of instructions by the basic optimizations described in Section 2.2. 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.

Figure 11 compares the normalized number of the L2-cache accesses and misses for the FTB and BLISS front-ends. As expected, both of the FTB and BLISS front-ends have a higher number of L2-cache accesses because of 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. BLISS design has a 10% higher average number of L2-cache accesses than the

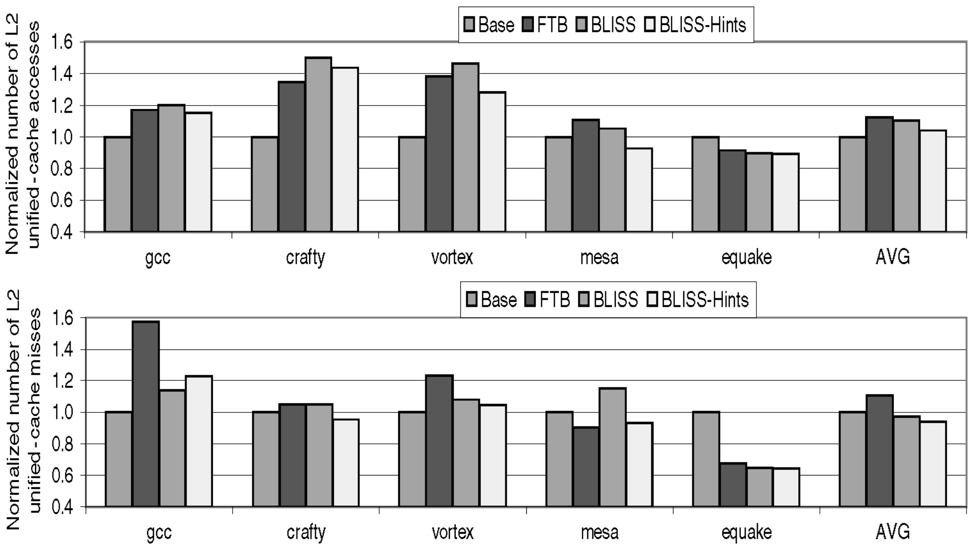


Fig. 11. L2 cache comparison for the eight-way processor configuration with the base, FTB, BLISS, and BLISS-HINTS front-ends. The top graph compares the normalize number of L2-cache accesses and the bottom one shows the normalized number of L2-cache misses. We present data for a representative subset of benchmarks, but the average refers to all benchmarks in this study.

base design, while the FTB design has a 12% higher average. FTB also exhibits a 10% higher L2-cache misses; this is because of the large number of erroneous instructions that are fetched 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 because of the better and accurate prediction as shown in Figure 8.

5.4 Detailed Comparison to FTB

To further explain the performance advantage of the BLISS design, we evaluated two FTB design variants. 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. We refer to this less aggressive FTB design as FTB-simple. This design fixes a 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 the FTB-simple design come at an additional cost. First, the increased number of misfetches from the biased not-taken branches may have a negative impact on performance. In addition, the fetch blocks are smaller than the blocks created in the original FTB design as biased not-taken branches are no longer

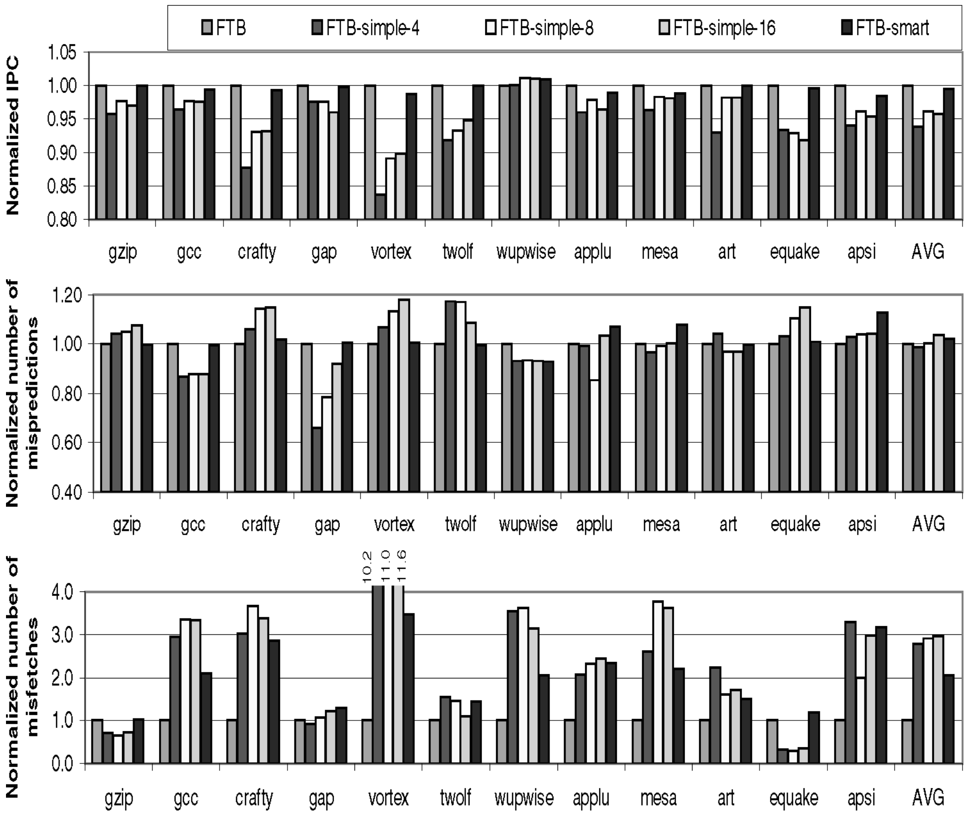


Fig. 12. FTB-simple and FTB-smart evaluation for the eight-way processor configuration. The top graph presents the normalized IPCs, the middle graph compares the normalized number of mispredictions, and the bottom graph shows the normalized number of misfetches.

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 a different interference patterns with the 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 execution 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. However, this is only true if the prediction is accurate. If it is not, then this would cost an additional misfetch, mispredict, and an extra fetch block that could cause increased contention for the finite capacity of the FTB.

Figure 12 compares the normalized IPC, number of mispredictions, and number of misfetches for the FTB-simple and the FTB-smart designs over the

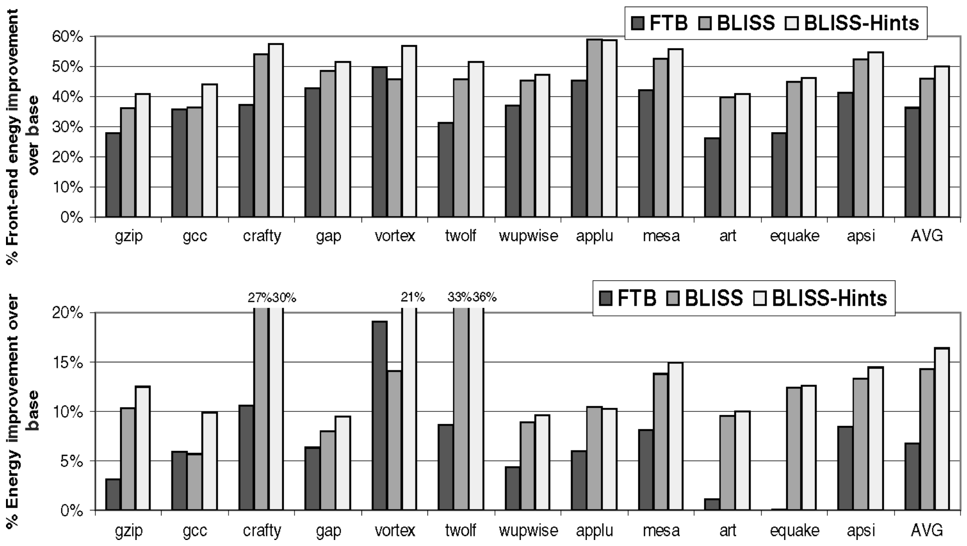


Fig. 13. Energy comparison for the 8-way processor configuration with the base, FTB, and BLISS front-ends. The top graph presents the percentage of front-end energy improvement and the bottom one shows the percentage of total energy improvement over the base for FTB and BLISS.

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. For FTB-simple, we see a consistent IPC degradation as a result of the negative impact of the additional misfetches that is not compensated for by the small change in the prediction accuracy. Although the number of misfetches decreases with smaller fetch-blocks, however, 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. For FTB-smart, we get a consistent increase in the number of misfetches. However, in terms of IPC and number of mispredictions we see a very small change. The average fetch-block length for FTB-smart also slightly decreases to 8.2 instructions. Overall, the original FTB design outperforms all of the other FTB configurations and, as shown in Figure 6, the BLISS-based design outperforms the original FTB design as it balances over- and underspeculation with better utilization of the BB-cache capacity.

5.5 Energy Comparison

The top graph of Figure 13 compares the front-end energy consumption improvement achieved for the eight-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 (instruction cache, BTB, predictors). Both the BLISS and FTB front-end designs significantly reduce the energy consumption in the front-end structures through selective word accesses in the instruction cache and by providing a higher number of instructions per prediction. The FTB design saves 36% of the front-end energy consumed by a conventional front-end design. The BLISS design achieves even higher improvement (46%) as it reduces the number of mispredicted instructions fetched from the instruction cache. When static branch hints are in use (*BLISS-Hints*), BLISS saves some additional energy in the hybrid predictor. On average, the static hints reduce the energy consumption in the predictor by 50%.

The bottom graph of Figure 13 presents the total energy savings achieved for the eight-way processor configuration using the FTB and BLISS front-ends over the base design. As expected, the energy savings track the IPC improvements shown in Figure 6. BLISS improves average energy consumption by 14 and 16% with and without the static branch hints, respectively. BLISS energy advantage over the base design is because of the significant reduction of the front-end energy consumption and the reduction in the energy wasted for mispredicted instructions as a result of the accurate prediction. By reducing execution time, the BLISS front-end also saves on the energy consumed by the clock tree and the processor resources even when they are stalling or idling. BLISS also provides a 7% total energy advantage over FTB as dynamic fetch block creation in the FTB front-end leads to the execution of misspeculated instructions that waste energy. The energy advantage of BLISS over FTB is mainly the high ratio of retired to fetched instructions. In other words, BLISS reduces overspeculation in the front-end, which, in turn, reduces the energy wasted for mispredicted instructions in the execution core.

5.6 Sensitivity Study

This section compares the performance for the FTB and BLISS designs when key architectural parameters of the front-end are varied. 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.

5.6.1 *BB-Cache/FTB Size and Associativity.* The performance with both of the decoupled front-ends depends heavily on the miss rate of the FTB and BB-cache, respectively. As we showed in Figure 9, the high BB-cache miss rate for `vortex` leads to a performance advantage for the FTB design. Figure 14 presents the average IPC across all benchmarks for the eight-way processor configuration with the FTB and BLISS 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 eight entries per cache lines in all cases.

Figure 14 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. The increasing number of entries eliminates stalls because of BB-cache misses for BLISS and reduces the inaccuracies introduced by fetch block recreation because of FTB misses in the FTB design. Associativity is less critical for both front-ends. With 512 or 1K entries, four-way associativity is preferred, but with a larger FTB or BB-cache, two-way is sufficient.

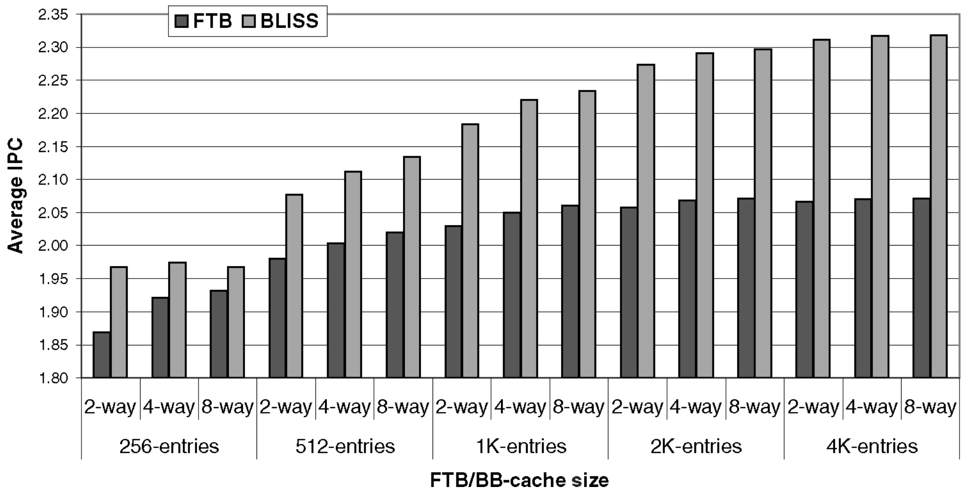


Fig. 14. Average IPC for the eight-way processor configuration with the FTB and BLISS front-ends as we scale the size and associativity of the FTB and BB-cache structures. For the BLISS front-end, we assume that static prediction hints are not available in this case.

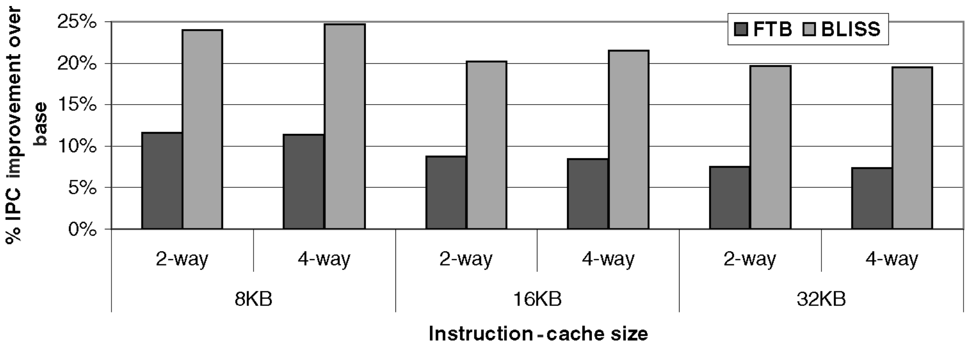


Fig. 15. Average percentage of IPC improvement with the FTB and BLISS front-ends over the base design as we vary the size and associativity of the instruction cache. We simulate an eight-way execution core, two-cycle instruction cache latency, and 2K entries in the BTB, FTB, and BB-cache, respectively.

5.6.2 *Instruction-Cache Size.* The use of a small, fast instruction cache was one of the motivations for the FTB front-end [Reinman et al. 2001]. The instruction prefetching enabled by the FTQ can compensate for the increased miss rate of a small instruction cache.

Figure 15 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 eight-way processor configuration. 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 32KB (from 12 to 7% for FTB, from 24 to 20% for BLISS). The BLISS front-end maintains a 13% IPC lead over the FTB design for all instruction-cache sizes and associativities.

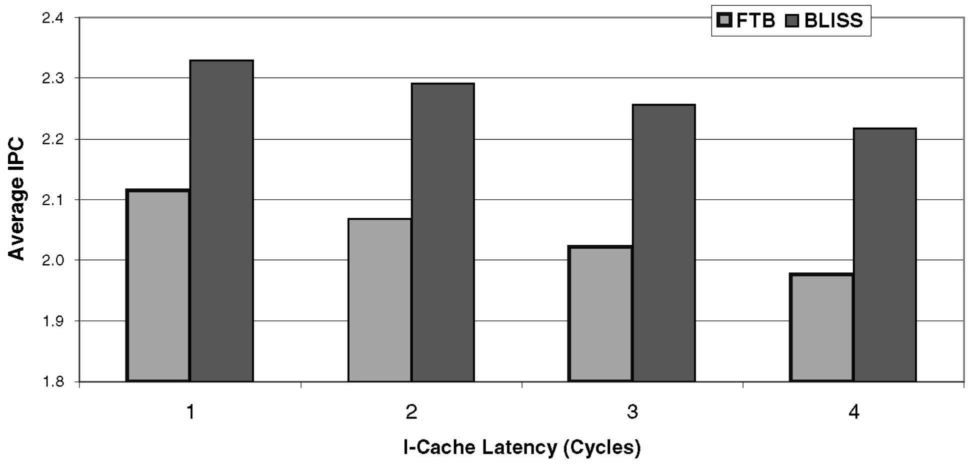


Fig. 16. Average IPC with the FTB and BLISS front-ends as we vary the latency of the instruction cache from 1 to 4 cycles. We simulate an 8-way execution core, 32KB pipelined instruction cache, and 2K entries in the BTB, FTB, and BB-cache respectively.

5.6.3 *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. Tolerance of 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 and power budget. Larger instruction caches are desirable for enterprise applications that tend to have larger instruction footprints [Barroso et al. 1998].

Figure 16 presents the average IPC for the eight-way processor configuration with the FTB and BLISS front-ends as we scale the instruction-cache latency from one to four cycles. With both front-ends, IPC decreases approximately 5% between the two end points, which shows good tolerance to instruction-cache latency. The performance loss is mainly because of the higher cost of recovery from mispredictions and misfetches. Increased instruction-cache latency does not change the performance advantage of the BLISS front-end over the FTB design.

5.6.4 *Four-Way Processor Performance Comparison.* The eight-way processor configuration analyzed in Section 5.3 and 5.5 represents an aggressive design point, where the execution core is designed for minimum number of back-end stalls. Figure 17 shows the impact of the front-end selection on the four-way execution core configuration described in Table I, which represents a practical commercial implementation.

Figure 17 shows that the performance comparison with the four-way execution core is nearly identical to that with the eight-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

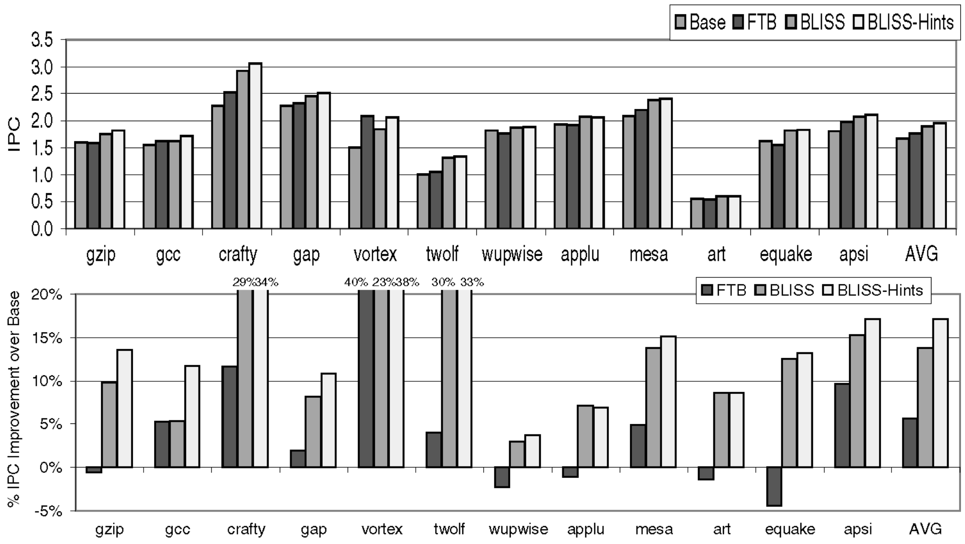


Fig. 17. Performance comparison for the 4-way processor configuration with the base, FTB, and BLISS front-ends. The top graph presents raw IPC and the bottom one shows the percentage of IPC improvement over the base for FTB and BLISS.

values for the improvements are lower than with the eight-way core because of some additional stalls in the execution core that mask stalls in the front-end.

The energy consumption comparison for the three front-ends with the four-way execution core is virtually identical to the one for the eight-way core in Section 5.5.

6. RELATED WORK

Some instruction sets allow for basic blocks descriptors, interleaved with regular instructions within the application binary code. *Prepare-to-branch* instructions specify the length and target of a basic block at its very beginning [Wedig and Rose 1984; Kathail et al. 1994; Schlansker and Rau 1999]. They allow for target address calculation and instruction prefetching a few cycles before the branch instruction is executed. The block-structured ISA (*BSA*) defines basic blocks of reversed-ordered instructions in order to simplify instruction renaming and scheduling [Hao et al. 1996; Melvin and Patt 1995]. The proposed instruction set goes a step further by separating basic block descriptors from regular instructions. The separation allows for instruction delivery optimizations in addition to the basic benefits of BSA.

Certain instruction sets allow for compiler-generated hints with individual branch and load/store instructions [Schlansker and Rau 1999]. BLISS provides a general mechanism for software hints at basic block granularity. The mechanism can support a variety of software-guided optimizations, as discussed in Section 2.3.

The decoupled control-execute (*DCE*) architectures use a separate instruction set with distinct architectural state for control-flow calculation [Topham and McDougall 1995; Manohar and Heinrich 1999]. DCE instruction sets allow for 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.

Block-based front-end engines were introduced by Yeh and Patt to improve prediction accuracy [Yeh and Patt 1992], with basic block descriptors formed by hardware without any additional ISA support. Decoupled front-end techniques have been explored by Calder and Grunwald [1994] and Stark et al. [1997]. Reinman et al. [1999a, 1999b, 2001] combined the two techniques in a comprehensive front-end, which detects basic block boundaries and targets dynamically in hardware and stores them in an advanced BTB called the fetch target buffer (FTB). The FTB coalesces multiple continuous basic blocks into a single fetch block in order to improve control-flow rate and better utilize the FTB capacity. Our work simplifies the FTB design using explicit ISA support for basic block formation. Despite the lack of block coalescing, BLISS outperforms the FTB front-end design by 13% in performance and 7% in total energy consumption. Ramirez et al. [2002] applied an FTB-like approach to long sequential instructions streams created with code layout optimizations and achieved 4% performance improvement.

Significant front-end research has also focused on trace caches [Rotenberg et al. 1996; Friendly et al. 1997], trace predictors [Jacobson et al. 1997], and trace construction [Patel et al. 1998]. Trace caches have been shown to work well with basic blocks defined by hardware [Black et al. 1999; Jourdan et al. 2000]. 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. Exploring such approaches is an interesting area for future work.

Other research in front-end architectures has focused on multiblock prediction [Seznec et al. 1996, 2003], control-flow prediction [Pnevmatikatos et al. 1993; Dutta and Franklin 1995], and parallel fetching of no contiguous instruction streams [Oberoi and Sohi 2003; Santana et al. 2004]. Such techniques are rather orthogonal to the block-aware ISA and can be used with a BLISS-based front-end engine. An investigation of the issues and benefits of such an effort is beyond the scope of this paper.

Several researchers have also worked 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 of the techniques focused on reducing the instruction cache energy [Powell et al. 2001; Albonesi 1999; Ghose and Kamble 1999]. Others have focused on reducing energy consumption in predictors [Parikh et al. 2002; Baniasadi and Moshovos 2002] and branch target buffer [Petrov and Orailoglu 2003; Shim et al. 2005]. Confidence prediction and throttling [Aragon et al. 2003] has been proposed as a way to control overspeculation to limit energy wasted on misspeculated instructions. Our proposal reduces the energy consumption in the front-end and, at the same time, improves the performance by increasing instruction fetch accuracy.

7. CONCLUSIONS

This paper proposed a block-aware instruction set (BLISS) that addresses basic challenges in the front-end of wide-issue, high-frequency superscalar processors. The ISA defines basic block descriptors in addition to and separately from the actual instructions. The software-defined descriptors allow a decoupled front-end that removes the instruction-cache latency from the prediction critical path, allows for instruction-cache prefetching, and allows for judicious use and training of branch predictors. We demonstrate that BLISS leads to 20% performance improvement and 14% total energy savings over conventional superscalar designs. Even though basic block formation and the resulting optimizations can be implemented in hardware without instruction set support, we show that the proposed ISA leads to significant advantages. By providing a balance between over and underspeculation, BLISS leads to 13% performance improvements and 7% energy savings over an aggressive front-end that dynamically builds fetch blocks in hardware. Finally, we show that the performance advantages of our proposal are robust across a wide range of design parameters for wide-issue processors.

Overall, this work demonstrates the advantages of using an expressive instruction set to address microarchitectural challenges in superscalar processors. Unlike techniques that rely solely on larger and more complex hardware structures, 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. Moreover, BLISS provides a flexible mechanism to communicate software-generated hints in order to address a range of performance, power consumption, and code density challenges.

ACKNOWLEDGMENTS

We would like to acknowledge Earl Kilian for his valuable input. This work was supported by the Stanford University, Office of Technology Licensing.

REFERENCES

Albonesi, D. H. 1999. Selective Cache Ways: On-Demand Cache Resource Allocation. In *Intl. Symposium on Microarchitecture*. IEEE Computer Society, Haifa. 248–259.

Aragon, J., Gonzalez, J., and Gonzalez, A. 2003. Power-Aware Control Speculation Through Selective Throttling. In *Intl. Symposium on High-Performance Computer Architecture*. IEEE Computer Society, Anaheim, CA. 103–112.

Bala, V., Duesterwald, E., and Banerjia, S. 2000. Dynamo: A Transparent Dynamic Optimization System. In *Conference on Programming Language Design and Implementation*. ACM Press, New York. 1–12.

Baniasadi, A. and Moshovos, A. 2002. Branch Predictor Prediction: A Power-Aware Branch Predictor for High-Performance Processors. In *Intl. Conference on Computer Design*. IEEE Computer Society, Freiburg. 458–461.

Barroso, L. et al. 1998. Memory System Characterization of Commercial Workloads. In *Intl. Symposium on Computer Architecture*. IEEE Computer Society, Barcelona. 3–14.

Black, B., Rychlik, B., and Shen, J. 1999. The Block-Based Trace Cache. In *Intl. Symposium on Computer Architecture*. IEEE Computer Society, Atlanta, GA. 196–207.

Brooks, D., Tiwari, V., and Martonosi, M. 2000. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *Intl. Symposium on Computer Architecture*. ACM Press, New York. 83–94.

BURGER, D. AND AUSTIN, M. 1997. Simplescalar Tool Set, Version 2.0. Tech. Rep. CS-TR-97-1342, University of Wisconsin, Madison. June.

CALDER, B. AND GRUNWALD, D. 1994. Fast and Accurate Instruction Fetch and Branch Prediction. In *Intl. Symposium on Computer Architecture*. IEEE Computer Society Press, Chicago, IL. 2–11.

CHEN, T. AND BAER, J. 1994. A Performance Study of Software and Hardware Data Prefetching Schemes. In *Intl. Symposium on Computer Architecture*. IEEE Computer Society Press, Chicago, IL. 223–232.

COOPER, K. AND MCINTOSH, N. 1999. Enhanced Code Compression for Embedded RISC Processors. In *Conference on Programming Language Design and Implementation*. ACM Press, New York. 139–149.

DUTTA, S. AND FRANKLIN, M. 1995. Control Flow Prediction with Tree-Like Subgraphs for Superscalar Processors. In *Intl. Symposium on Microarchitecture*. IEEE Computer Society Press, Ann Arbor, MI. 258–263.

FRIENDLY, D., PATEL, S., AND PATT, Y. 1997. Alternative Fetch and Issue Techniques from the Trace Cache Mechanism. In *Intl. Symposium on Microarchitecture*. IEEE Computer Society, Research Triangle Park, NC. 24–33.

GHOSE, K. AND KAMBLE, M. B. 1999. Reducing Power in Superscalar Processor Caches Using Subbanking, Multiple Line Buffers and Bit-line Segmentation. In *Intl. Symposium on Low Power Electronics and Design*. ACM Press, New York. 70–75.

HALAMBI, A., SHRIVASTAVA, A., BISWAS, P., DUTT, N., AND NICOLAU, A. 2002. An Efficient Compiler Technique for Code Size Reduction Using Reduced Bit-Width ISAs. In *Conference on Design, Automation and Test in Europe*. IEEE Computer Society, Paris. 402–408.

HAO, E., CHANG, P., EVERS, M., AND PATT, Y. 1996. Increasing the Instruction Fetch Rate via Block-Structured Instruction Set Architectures. In *Intl. Symposium on Microarchitecture*. IEEE Computer Society, Paris. 191–200.

HENNING, J. 2000. SPEC CPU2000: Measuring Performance in the New Millennium. *IEEE Computer 33,* 7 (July), 28–35.

JACOBSON, Q., ROTENBERG, E., AND SMITH, J. 1997. Path-based Next Trace Prediction. In *Intl. Symposium on Microarchitecture*. IEEE Computer Society, Research Triangle Park, NC. 14–23.

JOURDAN, S. ET AL. 2000. Extended Block Cache. In *Intl. Symposium on High-Performance Computer Architecture*. IEEE Computer Society, Toulouse. 61–70.

KATHAIL, V., SCHLANSKER, M., AND RAU, B. 1994. HPL PlayDoh Architecture Specification. Tech. Rep. HPL-93-80, HP Labs.

MANOHAR, R. AND HEINRICH, M. 1999. The Branch Processor Architecture. Tech. Rep. CSL-TR-1999-1000, Cornell Computer Systems Laboratory. Nov.

MAY, C. ET AL. 1994. *The PowerPC Architecture*. Morgan Kaufman, San Mateo, CA.

MCFARLING, S. 1993. Combining Branch Predictors. Tech. Rep. TN-36, DEC WRL. June.

MELVIN, S. AND PATT, Y. 1995. Enhancing Instruction Scheduling with a Block-structured ISA. *Intl. Journal on Parallel Processing 23,* 3 (June), 221–243.

MOSHOVOS, A., PNEVMATIKATOS, D. N., AND BANIASADI, A. 2001. Slice-Processors: An Implementation of Operation-Based Prediction. In *Intl. Conference on Supercomputing*. ACM Press, New York. 321–334.

OBEROI, P. AND SOHI, G. 2003. Parallelism in the Front-End. In *Intl. Conference on Computer Architecture*. ACM Press, New York. 230–240.

PARIKH, D., SKADRON, K., ZHANG, Y., BARCELLA, M., AND STAN, M. R. 2002. Power Issues Related To Branch Prediction. In *Intl. Symposium on High-Performance Computer Architecture*. IEEE Computer Society, Boston, MA. 233–246.

PATEL, S., EVERS, M., AND PATT, Y. 1998. Improving Trace Cache Effectiveness with Branch Promotion and Trace Packing. In *Intl. Symposium on Computer Architecture*. ACM Press, New York. 262–271.

PATT, Y. 2001. Requirements, Bottlenecks, and Good Fortune: Agents for Microprocessor Evolution. *Proceedings of the IEEE 89,* 11 (Nov.), 1153–1159.

PETROV, P. AND ORAILOGLU, A. 2003. Low-power Branch Target Buffer for Application-Specific Embedded Processors. In *Euromicro Symposium on Digital Systems Design*. IEEE Computer Society, Washington, DC. 158–165.

PNEVMATIKATOS, D. N., FRANKLIN, M., AND SOHI, G. S. 1993. Control Flow Prediction for Dynamic ILP Processors. In *Intl. Symposium on Microarchitecture*. IEEE Computer Society Press, Austin, TX. 153–163.

POWELL, M. ET AL. 2001. Reducing Set-Associative Cache Energy via Way Prediction and Selective Direct-Mapping. In *Intl. Symposium on Microarchitecture*. IEEE Computer Society, Austin, TX 54–65.

RAMIREZ, A., LARRIBA-PEY, J., AND VALERO, M. 2001. Branch Prediction Using Profile Data. In *EuroPar Conference*. Springer-Verlag, New York. 386–393.

RAMIREZ, A., SANTANA, O., LABRIBA, J., AND VALERO, M. 2002. Fetching Instruction Streams. In *Intl. Symposium on Microarchitecture*. IEEE Computer Society Press, Istanbul. 371–382.

REINMAN, G., AUSTIN, T., AND CALDER, C. 1999a. A Scalable Front-End Architecture for Fast Instruction Delivery. In *Intl. Symposium on Computer Architecture*. IEEE Computer Society, Atlanta, GA. 234–245.

REINMAN, G., CALDER, B., AND AUSTIN, T. 1999b. Fetch Directed Instruction Prefetching. In *Intl. Symposium on Microarchitecture*. IEEE Computer Society, Haifa. 16–27.

REINMAN, G., CALDER, C., AND AUSTIN, T. 2001. Optimizations Enabled by a Decoupled Front-End Architecture. *IEEE TC 50,* 40 (Apr.), 338–335.

REINMAN, G., CALDER, B., AND AUSTIN, T. M. 2002. High Performance and Energy Efficient Serial Prefetch Architecture. In *Intl. Symposium on High-Performance Computing*. Springer-Verlag, New York. 146–159.

RONEN, R., MENDELSON, A., ET AL. 2001. Coming Challenges in Microarchitecture and Architecture. *Proceedings of the IEEE 89,* 3 (Mar.), 325–340.

ROTENBERG, E., BENNET, S., AND SMITH, J. 1996. Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching. In *Intl. Symposium on Microarchitecture*. IEEE Computer Society, Austin, TX. 24–34.

SANTANA, O. J., RAMIREZ, A., LARRIBA-PEY, J. L., AND VALERO, M. 2004. A Low-Complexity Fetch Architecture for High-Performance Superscalar Processors. *ACM Transactions on Architecture and Code Optimization 1,* 2, 220–245.

SCHLANSKER, M. AND RAU, B. 1999. EPIC: An Architecture for Instruction-Level Parallel Processors. Tech. Rep. HPL-99-111, HP Labs.

SEZNEC, A. ET AL. 1996. Multiple-Block Ahead Branch Predictors. In *Intl. Conference on Architectural Support for Programming Languages and Operating Systems*. ACM Press, New York. 116–127.

SEZNEC, A. ET AL. 2003. Effective Ahead Pipelining of Instruction Block Address Generation. In *Intl. Conference on Computer Architecture*. ACM Press, New York. 241–252.

SHIM, S., KWAK, J.-W., KIM, C.-H., JHANG, S.-T., AND JHON, C.-S. 2005. Power-aware Branch Logic: A Hardware Based Technique for Filtering Access to Branch Logic. In *Intl. Embedded Computer Systems: Architectures, Modeling and Simulation Workshop*. Springer-Verlag, New York. 162–171.

SHIVAKUMAR, P. AND JOUPPI, N. 2001. Cacti 3.0: An Integrated Cache Timing, Power, Area Model. Tech. Rep. 2001/02, Compaq Western Research Laboratory. Aug.

STARK, J., RACUNAS, P., AND PATT, Y. 1997. Reducing the Performance Impact of Instruction Cache Misses by Writing Instructions into the Reservation Stations Out-of-Order. In *Intl. Symposium on Microarchitecture*. IEEE Computer Society, Research Triangle Park, NC. 34–43.

TOPHAM, N. AND MCDOUGALL, K. 1995. Performance of the Decoupled ACRI-1 Architecture: the Perfect Club. In *Intl. Conference on High-Performance Computing and Networking*. Springer-Verlag, New York. 472–480.

WEDIG, R. AND ROSE, M. 1984. The Reduction of Branch Instruction Execution Overhead Using Structured Control Flow. In *Intl. Symposium on Computer Architecture*. ACM Press, New York. 119–125.

YEH, T. AND PATT, Y. 1992. A Comprehensive Instruction Fetch Mechanism for a Processor Supporting Speculative Execution. In *Intl. Symposium on Microarchitecture*. IEEE Computer Society Press, Portland, OR. 129–139.

YEH, T. AND PATT, Y. 1993. A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History. In *Intl. Symposium on Computer Architecture*. IEEE Computer Society Press, Philadelphia, PA. 257–266.

ZMILY, A. AND KOZYRAKIS, C. 2005. Energy-Efficient and High-Performance Instruction Fetch using a Block-Aware ISA. In *Intl. Symposium on Low Power Electronics and Design*. ACM Press, New York. 36–41.

ZMILY, A., KILLIAN, E., AND KOZYRAKIS, C. 2005. Improving Instruction Delivery with a Block-Aware ISA. In *EuroPar Conference*. Springer-Verlag, Lisbon. 530–539.

Received August 2005; revised December 2005; accepted April 2006