Energy-Efficient and High-Performance Instruction Fetch using a Block-Aware ISA

Ahmad Zmily and Christos Kozyrakis
Electrical Engineering Department
Stanford University
zmily@stanford.edu, kozyraki@stanford.edu

ABSTRACT

The front-end in superscalar processors must deliver high application performance in an energy-effective manner. Impediments such as multi-cycle instruction accesses, instruction-cache misses, and mispredictions reduce performance by 48% and increase energy consumption by 21%. This paper presents a block-aware instruction set architecture (BLISS) that defines basic block descriptors in addition to the actual instructions in a program. BLISS allows for a decoupled front-end that reduces the time and energy spent on misspeculated instructions. It also allows for accurate instruction prefetching and energy efficient instruction access. A BLISS-based front-end leads to 14% IPC, 16% total energy, and 83% energy-delay-squared product improvements for wide-issue processors.

Categories and Subject Descriptors: C.1.1 **[Processor Architectures]**: Single Data Stream Architectures — Pipeline processors

General Terms: Design, Performance

Keywords: instruction set architecture, instruction delivery, basic blocks, decoupled architecture, energy efficiency.

1. INTRODUCTION

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[1]. 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 determines the performance and energy efficiency for a superscalar processor [2]. The rate and accuracy at which instructions enter the pipeline set an upper limit to sustained performance and determine the efficiency of energy use. Consequently, superscalar designs place increased demands on the processor *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 ILP (under-speculation). 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.

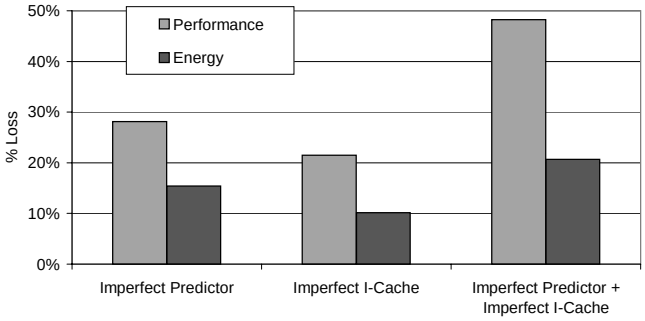


Figure 1: The percentage of performance and energy loss for a 4-way superscalar processor running SPEC CPU benchmarks due to prediction accuracy and instruction cache latency/misses.

In its effort to balance performance and energy efficiency, the front-end engine must handle three basic detractors: instruction cache misses that cause instruction delivery stalls; target and direction mispredictions for branches that send erroneous instructions to the execution core; and multi-cycle instruction cache accesses that cause additional uncertainty about the existence and direction of branches within the instruction stream. Figure 1 quantifies the performance and overall energy penalty due to the three problems for a 4-way superscalar processor running the SPEC CPU benchmarks [1]. Mispredictions alone cost 28% in performance and 15% in energy consumption. Multi-cycle instruction cache accesses and misses cost 22% in performance and 10% in energy consumption. Combined, front-end challenges reduce performance efficiency by up to 48% and overall energy efficiency by 21%.

In this paper, we address the front-end performance and energy challenges using a block-aware instruction set architecture (BLISS). 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 basic block, its potential target, and the number of instructions

[1]See Section 4 for the processor configuration and the evaluation methodology.

ISLPED'05, August 8–10, 2005, San Diego, California, USA

in the block. BLISS allows the processor front-end to access the software defined block descriptors through a small cache that replaces the common block target buffer (BTB).

BLISS enables significant performance and energy improvements at the front-end engine compared to a conventional ISA. 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 prefetch instructions and eliminate the impact of instruction cache misses. Furthermore, the control-flow information available in descriptors allows for judicious use of predictors, which reduces interference and training time and improves overall prediction accuracy. Finally, BLISS facilitates energy optimizations in the instruction cache, such as selective way/word access [3, 4] and serial access to data and tags [4], without sacrificing performance.

We demonstrate that a BLISS-based front-end design allows for a 14% performance improvement with a 4-way superscalar processor, while reducing its energy consumption by 16%. BLISS improves **both** performance and energy consumption. Moreover, we show that BLISS compares favorably to advanced, hardware-only schemes for decoupled front-ends. Overall, this work demonstrates the performance and energy potential of software-assisted superscalar execution using an expressive ISA.

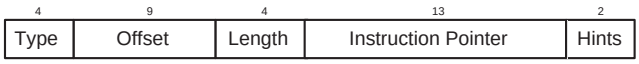
2. BLOCK-AWARE ISA

Our proposal is based on a *block-aware instruction set (BLISS)* that explicitly describes basic blocks. 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.

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 2 presents the format of a basic block descriptor (*BBD*). Each BBD defines the type of the control-flow operation that terminates the block. The BBD 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. The last BBD field contains optional compiler-generated hints. In this study, we make limited use of this field to convey branch prediction hints generated through profiling [5]. The overall BBD length is 32 bits.

BLISS treats each basic block as an atomic unit of execution. There is a single program counter and it only points within the code segment for BBDs. The execution of all 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). Precise exceptions are supported similar to [6].

The BBDs provide the processor front-end with architectural information about the program control-flow in a compressed and accurate manner. Since BBDs are stored separately from instructions, their information is available for front-end tasks before instructions are fetched and decoded. The sequential block target is always at 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 a PC-relative control-flow instructions (branches – `BR_B` and `BR_F`, jumps – `J` and `JAL`, loop – `LOOP`). For the remaining cases (jump register – `JR` and `JALR`, return – `RET`), the non-sequential target is provided by the last instruction in the block through a register. BBDs provide the branch



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

Figure 2: The 32-bit basic block descriptor format in BLISS.

condition when it is statically determined (all jumps, return, fall-through blocks). For conditional branches, the BBD provides type information (forward, backward, loop) and hints which can assist with dynamic prediction. The actual branch condition is provided by the last instruction in the block. Finally, the instruction pointer and length fields can be used for instruction prefetching.

Figure 3 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.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.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`.

Note that 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 ISAs. For example, the target register (`r3`) for the `jr` instruction in Figure 3 could be the destination register of a previous load instruction.

3. BLISS DECOUPLED FRONT-END

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 branch target buffer (BTB) with a *BB-cache* that caches the block descriptors in programs. 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

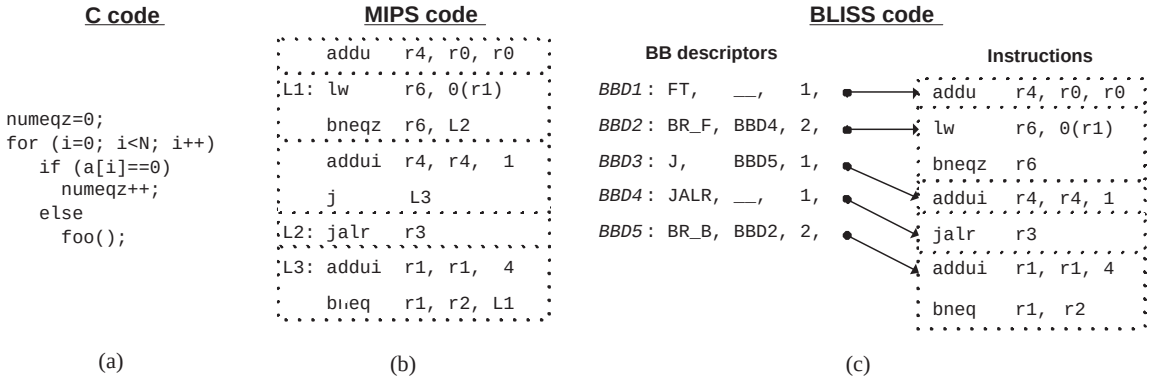


Figure 3: Example program in (a) C source code, (b) MIPS assembly code, and (c) BLISS assembly code. With both (b) and (c), the instructions in each basic block are identified with dotted-line boxes. Register `r3` contains the address for the first instruction (b) or first basic block descriptor (c) of function `foo`. For illustration purposes, the instruction pointers in basic block descriptors are represented with arrows.

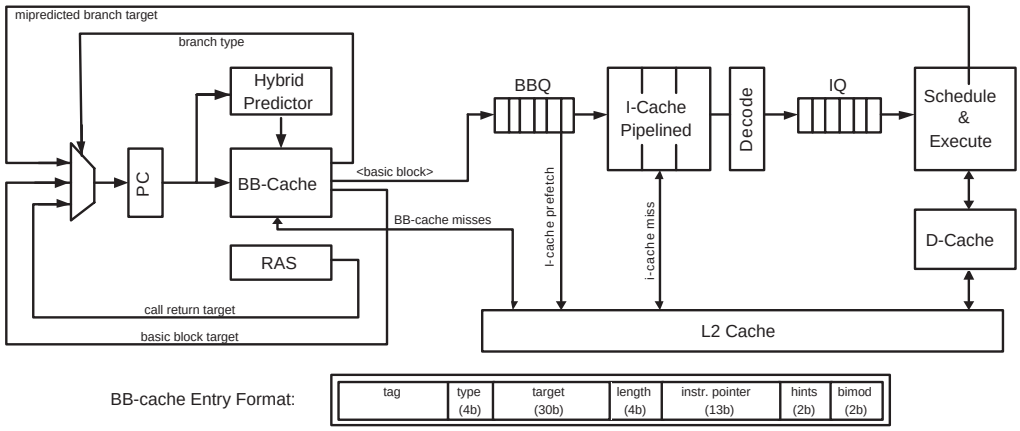


Figure 4: A decoupled front-end for a superscalar processor based on the BLISS ISA.

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.

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 direction is also verified by a tag-less, hybrid predictor. 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 BLISS front-end alleviates all performance and energy shortcomings of a conventional front-end. 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, there is no performance motivation for a single-cycle I-cache, which would be energy wasteful. 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, yet accurate, view into the instruction address stream and can be used for instruction prefetching [7] that hides instruction cache misses without wasting energy on unnecessary L2-cache accesses.

The availability of basic block descriptors also allows for energy optimizations in the instruction cache. Each basic block defines exactly the number of instructions needed from the I-cache. Using segmented word lines [3] for the data portion of the I-cache, we can fetch the necessary words while activating only the necessary sense-amplifiers in each case. Furthermore, front-end decoupling tolerates higher I-cache latency without loss in speculation accuracy. Hence, we can access first the tags for a set associative I-cache, and in subsequent cycles, access the data only in the way that hits [4]. Our results indicate the two optimizations combined save up to 40% of energy in the front-end compared to a conventional superscalar design. Finally, we can merge the I-cache accesses for sequential block in the BBQ that hit in the same I-cache line, in order to save decoding and tag access energy.

	Base	FTB	BLISS
Fetch Width	4 Inst./cycle	1 FB/cycle	1 BB/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		
D-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 1: The microarchitecture parameters for the simulations. The common parameters apply to all three models (base, FTB, BLISS).

A decoupled front-end similar to the one in Figure 4 can be implemented without the ISA support provided by BLISS. The FTB design [8, 9, 4] describes the latest of such design. The FTB detects basic block boundaries and targets dynamically in hardware and stores them in an advanced BTB called the fetch target buffer (FTB). Block boundaries are discovered by introducing large instruction sequential blocks which are later shortened when jumps are decoded (misfetch) or branches are taken (mispredict) within the block. The FTB enables I-cache energy optimizations and allows for instruction fetch decoupling and prefetching as described above. Furthermore, the FTB coalesces multiple continuous basic blocks into a single long fetch block in order to improve control-flow rate and better utilize the FTB capacity. Nevertheless, the simpler BLISS front-end outperforms the aggressive FTB design by providing a better balance between over- and under-speculation. With BLISS, block formation is statically done in software and it never introduces misfetches. In addition, the PC used to access the hybrid predictor for each block (branch) is the same. With FTB, as fetch blocks shrink dynamically when branches switch behavior, the PC used to index in the predictor and FTB for each branch changes dynamically, causing slower predictor training and additional interference.

4. METHODOLOGY

We simulate a 4-way superscalar processor to compare the BLISS-based front-end to conventional (base) and FTB-based front-ends. Table 1 summarizes the key architectural parameters. Note that the target prediction buffers in the three front-ends (BTB, FTB, and BB-cache) have exactly the same capacity for fairness. All other parameters are identical across the three models. We have also performed detailed experiments varying several of these parameters and the results are consistent (BTB size, I-cache size, etc.). For BLISS, we fully model contention for the L2-cache bandwidth between BB-cache misses and I-cache or D-cache misses. Our graphs present two sets of results for BLISS: without (BLISS) and with (BLISS-hints) the prediction hints in the BBDs. We do not discuss BLISS-hints in details due to space limitations.

We study 12 SPEC CPU2000 benchmarks using their reference datasets and compiled at the -O3 optimization level. We skip the first billion instructions and simulate another billion instructions for detailed analysis. We generated BLISS executables using a static binary translator, which can handle arbitrary programs written in any language. The generation of BLISS executable could also be done using a transparent, dynamic compilation framework. Despite introducing the block descriptors, BLISS executables are actually up to 16% smaller than the original binaries, as BLISS allows aggressive code size optimizations such as branch removal and common block elimination. The evaluation of code size optimizations is omitted due to space limitations.

Our simulation framework is based on the Simplescalar/PISA 3.0 toolset [10], 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 [11] (non-ideal, aggressive conditional clocking). Energy consumption was calculated for a 0.10μm process with a 1.1V power supply. The reported *Front-end Energy* includes I-cache, predictors, and BTB/FTB/BB-Cache. *Total Energy* includes all the processor components (front-end, execution core, and all caches). The BTB/FTB/BB-cache are 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 using CACTI 3.0.

5. EVALUATION

Figure 5 compares the IPC, front-end energy consumption, total energy consumption, and energy-delay-squared product (ED2P) improvements achieved for the 4-way processor configuration with the three front-ends. BLISS outperforms the base front-end for all benchmarks with an average IPC improvement of 14% as it allows for more accurate prediction and instruction prefetching. Compared to the base, BLISS reduces by 36% the number of pipeline flushes due to 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 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 I-cache latency and the accurate indexing and judicious use of the hybrid predictor. BLISS also enables I-cache prefetching though the BBQ which reduces the number of I-cache misses by 10% on average for the benchmarks studied. The remaining performance improvement suggested in Figure 1 can only be achieved with a much better branch predictor. The hardware-based FTB front-end outperforms the base for only half of the benchmarks and most of the 5% average IPC improvement is due to `vortex`. BLISS outperforms FTB for all benchmarks but `vortex`, with an average IPC advantage of 9% (up to 12% with BLISS-hints). Although FTB allows similar front-end decoupling with instruction prefetching, it suffers from higher number of control-flow mispredictions due to its aggressive over-speculation.

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). Both of the BLISS and FTB front-ends designs reduce significantly the energy consumption in the front-end structures through selective way/word accesses and serial data/tag accesses in the instruction cache. The FTB design saves 52% of the front-end energy consumed by a conventional front-end design. The BLISS design achieves even higher improvement (65%) as it reduces the number of mispredicted instructions fetched from the instruction cache.

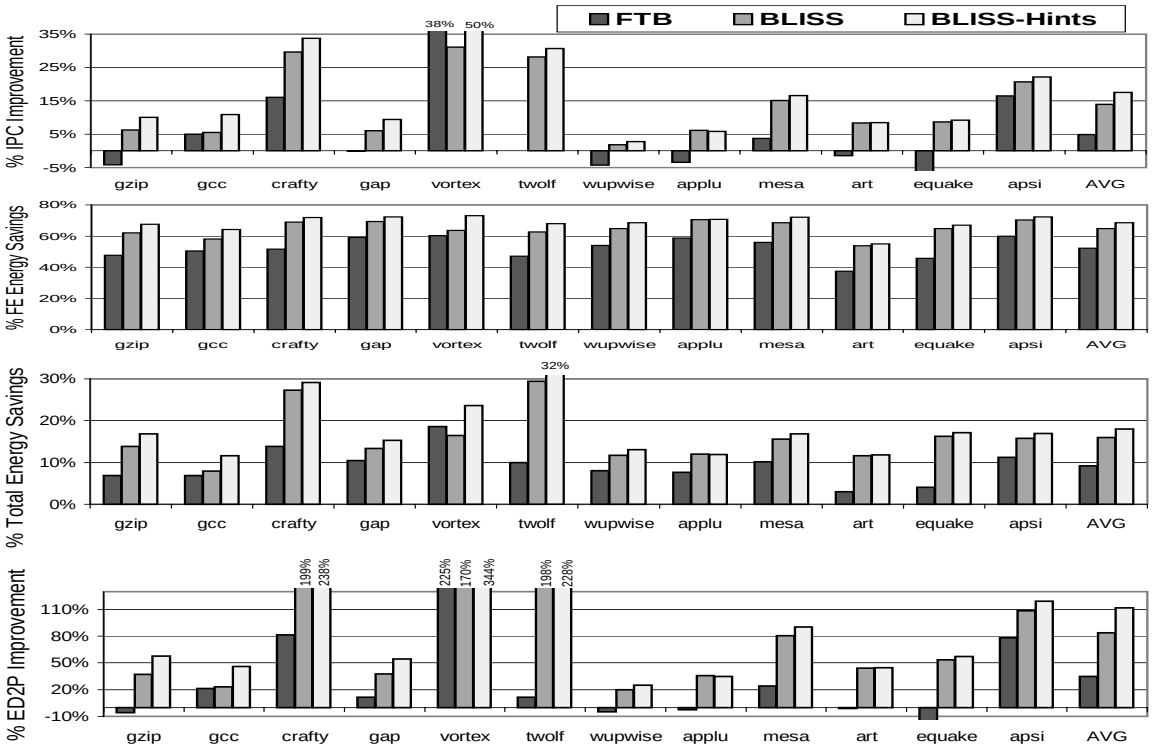


Figure 5: Evaluation of the 4-way processor configuration with the FTB and BLISS front-ends over the base design.

BLISS offers a 16% total energy advantage over the base design by significantly reducing front-end energy consumption and controlling over-speculation to limit the energy wasted for mispredicted instructions. BLISS-based design actually achieves 75% of the total energy improvement suggested in Figure 1. BLISS also provides a 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. BLISS compares even more favorably to FTB if one considers composite efficiency metrics such as the energy-delay-squared product, which is appropriate for high-performance, energy-efficient processors. The BLISS design achieves an 83% improvement in ED2P, while the FTB design has only 35% overall ED2P improvement over the base.

Figure 6 explains the basic difference in efficiency between the hardware-only approach (FTB) and our software-assisted approach (BLISS). It 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 BBQ/FTQ in each cycle. Looking at fetch IPC, the FTB design fetches more instructions per cycle than BLISS (3.83 versus 2.30 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. Nevertheless, in terms of commit IPC (instructions retired per cycle), the BLISS front-end has an advantage(1.64 versus 1.78). 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 number of erroneous instructions that lead to misfetches, mispredictions, and slow predictor training. The impact of mispredicted instructions is negative on both performance and energy consumption, especially in the back-end of the processor. Vortex is one of the few benchmarks for which the FTB design outperforms BLISS. This is due to the longer fetch blocks that the FTB front-end is capable of forming (8.1 instructions per FTB entry versus 5.0 instructions per BB-cache entry).

Although the BLISS L2-cache serves the BB-cache misses in addition to the I-cache and D-Cache misses, the number of L2-cache accesses and misses are slightly better than the numbers for the FTB design. BLISS has an 8% higher number of L2-cache accesses and a 3% lower number of L2-cache misses compared to the base design for the benchmarks studied. The increased number of L2-cache accesses for BLISS and FTB designs is mainly due to instruction prefetching.

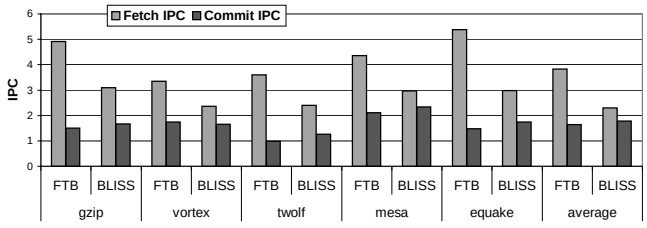


Figure 6: Fetch and commit IPC with the FTB and BLISS front-ends. We present data for a few representative cases, but the average refers to all 12 benchmarks. For BLISS, we present the data for the case without static branch hints.

6. RELATED WORK

Several researchers have 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 include reducing the instruction cache energy by way prediction[12], selective cache way access [13], and

sub-banking [3]. Others have focused on reducing energy consumption in predictors by using multi-banking [14] or selective prediction[15]. Confidence prediction and throttling [16] has been proposed as a way to control over-speculation to limit energy wasted on misspeculated instructions. Our proposal reduces the energy consumption in front-end and at the same time improves the performance by increasing instruction fetch accuracy.

Certain ISAs allow for basic blocks descriptors, interleaved with regular operations in the instruction stream (e.g. *prepare-to-branch* instructions in [17, 18]). They allow for target address calculation and instruction prefetching a few cycles before the branch instruction is decoded. The block-structured ISA (*BSA*) by Patt et al. [6] defines basic blocks of reversed ordered instructions as atomic execution units in order to simplify instruction renaming and scheduling. BLISS goes a step further by separating basic block descriptors from regular instructions which allows for instruction fetch bandwidth improvements. The benefits from BSA and BLISS are complimentary.

Block-based front-end architectures were introduced by Yeh and Patt [19], with basic block descriptors formed by hardware without any additional architectural support. Decoupled front-end techniques have been explored by Calder and Grunwald [20] and Stark et al. [21]. Reinman et al. combined the two techniques in a comprehensive front-end with prefetching capabilities [8, 9]. Our work improves their design using explicit ISA support for basic block formation. Significant amount of front-end research has also focused on trace caches [22, 23]. Trace caches have been shown to work well with basic blocks defined by hardware [24]. One can form streams or 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 stream or trace formation. Exploring such approaches is an interesting area for future work.

7. CONCLUSIONS

This paper proposed block-aware instruction set to address performance and energy challenges in the front-end of high-end superscalar processors. The ISA defines basic block descriptors in addition to and separately from the actual instructions. Software-defined basic blocks allow a decoupled front-end to avoid the wasteful over-speculation during hardware creation of fetch blocks and to achieve highly accurate control-flow speculation. They also allow energy optimizations in the instruction cache access. Through detailed simulation, we have shown that the proposed ISA allows for a 14% performance, 16% total energy, and 83% ED2P improvements over a conventional superscalar design. The ISA-supported front-end also outperforms (9% IPC, 7% energy, and 48% ED2P) advanced decouple front-ends that dynamically build fetch blocks in hardware. 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.

8. ACKNOWLEDGEMENTS

We would like to acknowledge Earl Kilian for his valuable input. This work was supported by a Stanford OTL grant.

9. REFERENCES

[1] W. M. Felter et al. On The Performance and Use of Dense Servers. *IBM J. RES. and DEV.*, 47(5/6), September 2003.

[2] R. Ronen, A. Mendelson, et al. Coming Challenges in Microarchitecture and Architecture. *Proceedings of the IEEE*, 89(3), March 2001.

[3] Kanad Ghose and Milind B. Kamble. Reducing Power in Superscalar Processor Caches Using Subbanking, Multiple Line Buffers and Bit-line Segmentation. In *Intl. Symposium on Low Power Electronics and Design*, San Diego, CA, August 1999.

[4] Glenn Reinman, Brad Calder, and Todd M. Austin. High Performance and Energy Efficient Serial Prefetch Architecture. In *Intl. Symposium on High Performance Computing*, Kansai Science City, Japan, May 2002.

[5] A. Ramirez, J. Larriba-Pey, and M. Valero. Branch Prediction Using Profile Data. In *EuroPar Conference*, Manchester, UK, August 2001.

[6] S. Melvin and Y. Patt. Enhancing Instruction Scheduling with a Block-structured ISA. *Intl. Journal on Parallel Processing*, 23(3), June 1995.

[7] T. Chen and J.L. Baer. A Performance Study of Software and Hardware Data Prefetching Schemes. In *Intl. Symposium on Computer Architecture*, Chicago, IL, April 1994.

[8] G. Reinman, B. Calder, and T. Austin. Fetch Directed Instruction Prefetching. In *Intl. Symposium on Microarchitecture*, Haifa, Israel, November 1999.

[9] G. Reinman, C. Calder, and T. Austin. Optimizations Enabled by a Decoupled Front-End Architecture. *IEEE TC*, 50(40), April 2001.

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

[11] D. Brooks, V. Tiwari, , and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *Intl. Symposium on Computer Architecture*, Vancouver, BC, Canada, June 2000.

[12] M. Powell et al. Reducing Set-Associative Cache Energy via Way Prediction and Selective Direct-Mapping. In *Intl. Symposium on Microarchitecture*, Austin, Texas, December 2001.

[13] David H. Albonesi. Selective Cache Ways: On-Demand Cache Resource Allocation. In *Intl. Symposium on Microarchitecture*, Haifa, Israel, November 1999.

[14] Dharmesh Parikh, Kevin Skadron, Yan Zhang, Marco Barcella, and Mircea R. Stan. Power Issues Related To Branch Prediction. In *Intl. Symposium on High-Performance Computer Architecture*, Boston, MA, February 2001.

[15] A. Baniasadi and A. Moshovos. Branch Predictor Prediction: A Power-Aware Branch Predictor for High-Performance Processors. In *Intl. Conference on Computer Design*, Freiburg, Germany, September 2002.

[16] J. Aragon, J. Gonzalez, and A. Gonzalez. Power-Aware Control Speculation Through Selective Throttling. In *Intl. Symposium on High-Performance Computer Architecture*, Anaheim, CA, February 2003.

[17] R. Wedig and M. Rose. The Reduction of Branch Instruction Execution Overhead Using Structured Control Flow. In *Intl. Symposium on Computer Architecture*, Ann Arbor, MI, June 1984.

[18] V. Kathail, M. Schlansker, and B. Rau. HPL PlayDoh Architecture Specification. Technical Report HPL-93-80, HP Labs, 1994.

[19] T. Yeh and Y. Patt. A Comprehensive Instruction Fetch Mechanism for a Processor Supporting Speculative Execution. In *Intl. Symposium on Microarchitecture*, Portland, OR, December 1992.

[20] B. Calder and D. Grunwald. Fast and Accurate Instruction Fetch and Branch Prediction. In *Intl. Symposium on Computer Architecture*, Chicago, IL, April 1994.

[21] J. Stark, P. Racunas, and Y. Patt. Reducing the Performance Impact of Instruction Cache Misses by Writing Instructions into the Reservation Stations Out-of-Order. In *Intl. Symposium on Microarchitecture*, Research Triangle Park, NC, December 1997.

[22] D. Friendly, S. Patel, and Y. Patt. Alternative Fetch and Issue Techniques from the Trace Cache Mechanism. In *Intl. Symposium on Microarchitecture*, Research Triangle Park, NC, December 1997.

[23] S. Patel, M. Evers, and Y. Patt. Improving Trace Cache Effectiveness with Branch Promotion and Trace Packing. In *Intl. Symposium on Computer Architecture*, Barcelona, Spain, June 1998.

[24] S. Jourdan et al. Extended Block Cache. In *Intl. Symposium on High-Performance Computer Architecture*, Toulouse, France, January 2000.