Improving Instruction Delivery with a Block-Aware ISA

Ahmad Zmily, Earl Killian, and Christos Kozyrakis

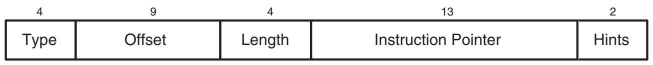
Electrical Engineering Department
Stanford University
{zmily,killian,kozyraki}@stanford.edu

Abstract. Instruction delivery is a critical component for wide-issue 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, multi-cycle instruction cache accesses, and target or direction mispredictions for control-flow operations. This paper introduces a block-aware ISA (BLISS) that helps accurate instruction delivery by defining basic block descriptors in addition to and separate from the actual instructions in a program. We show that BLISS allows for a decoupled front-end that tolerates cache latency and allows for higher speculation accuracy. This translates to a 20% IPC and 14% energy improvements over conventional front-ends. We also demonstrate that a BLISS-based front-end outperforms by 13% decoupled front-ends that detect fetched blocks dynamically in hardware, without any information from the ISA.

1 Introduction

Effective instruction delivery is vital for superscalar processors [1]. The rate and accuracy at which instructions enter the 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 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.

To overcome these problems in high performance yet energy efficient way, we propose 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 block, its potential target, and the number of instructions in the basic block. BLISS allows the processor front-end to access the software defined block descriptors through a small cache that replaces the block 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 prefetch instructions and eliminate the impact of instruction cache misses. Furthermore, the control-flow information available in descriptors allows for judicious use of branch predictors, which



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.

reduces interference and training time and improves overall prediction accuracy. We demonstrate that for an 8-way superscalar processor, a BLISS-based front-end allows for 20% performance improvement and 14% overall energy savings over a conventional front-end engine.

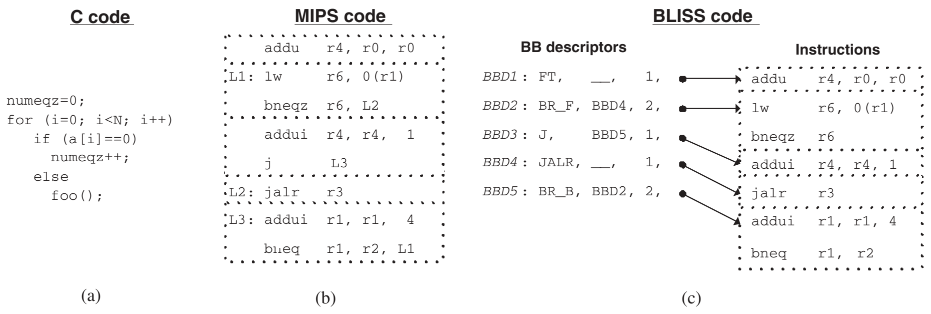
Moreover, BLISS compares favorably to advanced, hardware-based schemes for decoupled front-end engines such as the fetch-block-buffer (FTB) design [2, 3]. The FTB performs aggressive block coalescing to increase the number of instructions per control-flow prediction and increase the utilization of the BTB. The BLISS-based front-end provides higher control-flow accuracy than the FTB by removing over-speculation with block fetching and coalescing. Our experiments show that a BLISS-based 8-way processor provides 13% higher performance and 7% overall energy savings over the FTB design.

Overall, we demonstrate the potential of delegating portions of instruction delivery (accurate fetch block formation) to software using an expressive ISA.

2 Block-Aware Instruction Set Architecture

Our proposal for addressing front-end performance 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 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 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 [4]. The overall BBD length is 32 bits.



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

```
     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
```

```
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
```

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.

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 [5].

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 block 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 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, instruction pointer and length fields can be used for instruction (pre)fetching.

Figure 2 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 2.b). The 4 control-flow operations define 5 basic blocks. All branch conditions and targets are defined by the branch and jump instructions. With the BLISS equivalent of MIPS (Figure 2.c), the program requires 5 basic block descriptors and 7 instructions. All PC-relative offsets for branch and jump operations are available in BBDs. Compared to the original code, we have eliminated the `j` instruction. The corresponding descriptor (BBD3) defines both the control-flow type (`J`) and the offset, hence the jump instruction itself is redundant. However, we cannot eliminate

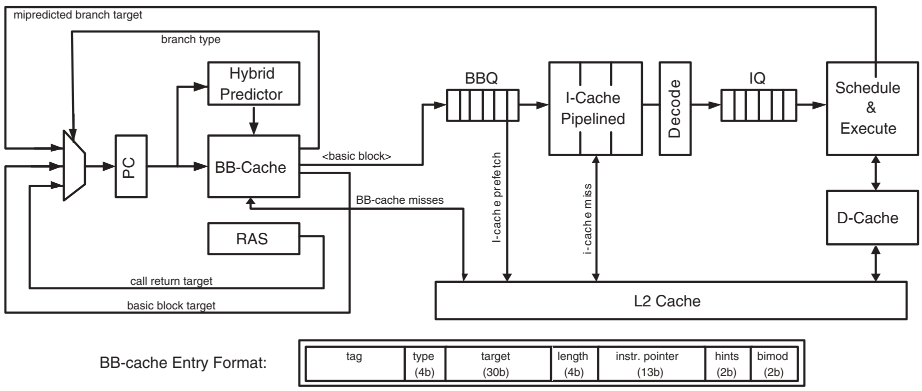


Fig. 3. A decoupled front-end for a superscalar processor based on the BLISS ISA

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

3 Decoupled Front-End for the Block-Aware ISA

The BLISS ISA suggests a superscalar front-end that fetches BBDs and the associated instructions in a decoupled manner. Figure 3 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 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 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 length, 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 [6].

A decoupled front-end similar to the one in Figure 3 can be implemented without the ISA support provided by BLISS. The FTB design [2, 3] 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 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 an 8-way superscalar processor in order 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 (4-way processor, BTB size, I-cache latency, 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) using 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 [7]. The benchmarks are compiled at the -O3 optimization level. In all cases, we skip the first

Table 1. The microarchitecture parameters for the simulations. The common parameters apply to all three models (base, FTB, BLISS).

	Base	**FTB**	**BLISS**
Fetch Width	8 instructions/cycle	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
I-cache	32 KBytes, 4-way, 64B blocks, 1 port, 2-cycle access pipelined
Issue/Commit Width	8 instructions/cycle
IQ/RUU/LSQ Size	64/128/128 entries
FUs	8 INT & 6 FP
D-cache	64 KBytes, 4-way, 64B blocks, 2 ports, 2-cycle access pipelined
L2 cache	1 MByte, 8-way, 128B blocks, 1 port, 12-cycle access, 4-cycle repeat rate
Main memory	100-cycle access

billion instructions and simulate another billion instructions for detailed analysis. We generate 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 [8]. 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 [9], 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 [10]. Energy consumption was calculated for a 0.10μm process with a 1.1V power supply. The reported *Total Energy* includes all the processor components (front-end, execution core, and all caches).

5 Evaluation

Figure 4 presents IPC and IPC improvement for the BLISS front-end over the base and FTB front-ends for the 8-way superscalar processor. BLISS outperforms the base front-end for all benchmarks with an average IPC improvement of 20%. The hardware-based FTB front-end outperforms the base for only half of the benchmarks and most of the 7% average IPC improvement is due to `vortex`. BLISS outperforms FTB for all benchmarks but `vortex`, with an average IPC advantage of 13% (up to 18% with BLISS-hints).

Figure 4 also presents total energy savings. BLISS provides a 14% total energy improvement over the base design. The advantage is mostly due to the elimination of a

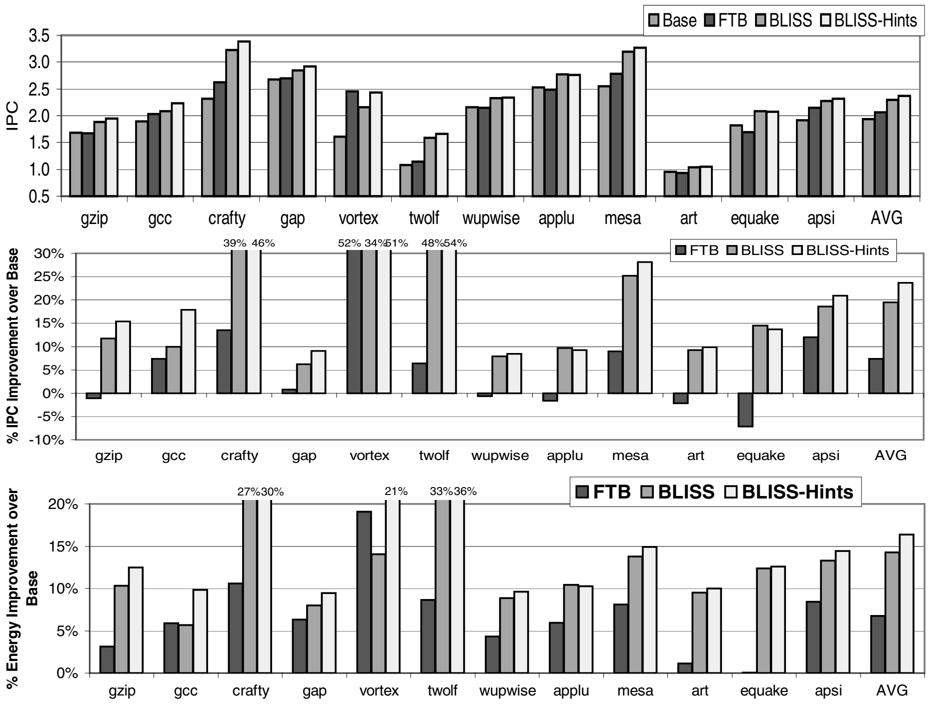


Fig. 4. IPC, percentage of IPC improvement, and percentage of total energy improvement for the FTB and BLISS front-ends over the base fornt-end design.

significant number of pipeline flushes due to control-flow misprediction. BLISS offers a 7% energy advantage over FTB which allows similar energy optimizations in the front-end but suffers from higher number of control-flow mispredictions. It is important to note from Figure 4 that BLISS provides **both** performance and energy advantages over the base and FTB.

Figure 5 explains the basic performance advantage of BLISS over the base and FTB design. 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. The FTB front-end has a significantly higher number of pipeline flushes compared to the BLISS front-end as block recreation affects the prediction accuracy of the hybrid predictor due to longer training and increased interference. Both BLISS and FTB allow for a decoupled front-end with instruction prefetching. BLISS enables I-cache prefetching though the BBQ which reduces the number of I-cache misses by 24% on average for the benchmarks studied. Although the BLISS L2-cache serves an additional type of misses from the BB-cache, BLISS number of

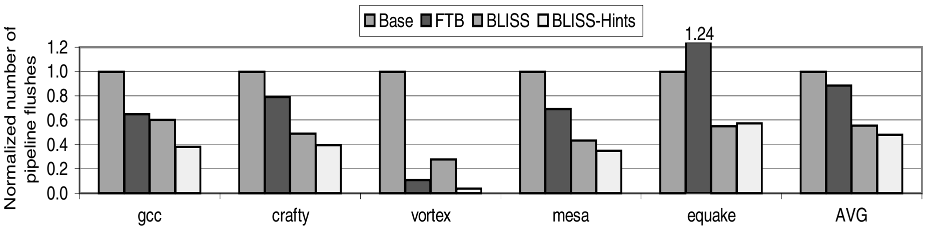


Fig. 5. Normalized number of pipeline flushes for the base, FTB, BLISS for representative benchmarks. The average is across all 12 benchmarks.

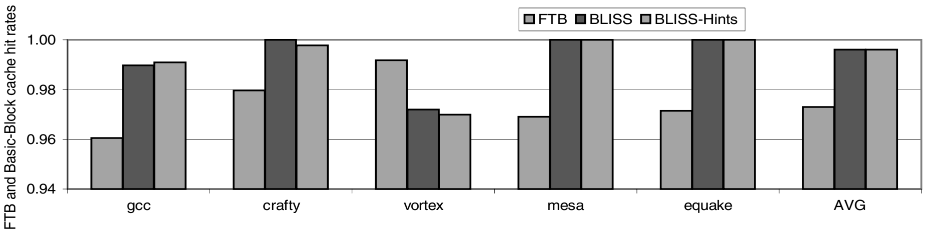


Fig. 6. Normalized FTB and BB-cache hit rates for representative benchmarks. The average is across all 12 benchmarks.

L2-cache accesses and misses are slightly better than the numbers for the FTB design. BLISS has a 10% higher number of L2-cache accesses, and 2% 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 shows the BB-cache and FTB hit rates to evaluate the effectiveness of the FTB in forming fetch-blocks and the BB-cache in delivering BBDs. Since the FTB returns a fall-through block address even when it misses in order to avoid storing the fall-through blocks, we define its miss rate as the number of misfetches divided over the number of FTB accesses. A misfetch occurs when the decoder detects that the block fetched from the FTB is wrong and needs to be updated and a new block to be fetched. 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 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).

6 Related Work

Certain ISAs allow for basic blocks descriptors, interleaved with regular operations in the instruction stream (e.g. *prepare-to-branch* instructions in [11, 12]). 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. [5] 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. The decoupled control-execute architectures use a separate ISA with distinct architectural state for control-flow calculation [13, 14]. The BBDs in BLISS are not a stand-alone ISA and do not define any state, eliminating the deadlock scenarios with decoupled control-execute ISAs.

Block-based front-end architectures were introduced by Yeh and Patt [15], with basic block descriptors formed by hardware without any additional architectural support. Decoupled front-end techniques have been explored by Calder and Grunwald [16] and Stark et al. [17]. Reinman et al. combined the two techniques in a comprehensive front-end with prefetching capabilities [2, 3]. 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 [18–20]. Trace caches have been shown to work well with basic blocks defined by hardware [21]. 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

We present a block-aware ISA that addresses basic challenges in the front-end of wide 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 with highly accurate control-flow speculation, which leads to 20% IPC and 14% energy advantages over conventional designs. The ISA-supported front-end also outperforms (13% IPC and 7% energy) 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.

Acknowledgements

This work was supported by a Stanford OTL grant.

References

1. R. Ronen, A. Mendelson, et al. Coming Challenges in Microarchitecture and Architecture. *Proceedings of the IEEE*, 89(3), March 2001.
2. G. Reinman, B. Calder, and T. Austin. Fetch Directed Instruction Prefetching. In *Intl. Symposium on Microarchitecture*, Haifa, Israel, November 1999.
3. G. Reinman, C. Calder, and T. Austin. Optimizations Enabled by a Decoupled Front-End Architecture. *IEEE TC*, 50(40), April 2001.

4. A. Ramirez, J. Larriba-Pey, and M. Valero. Branch Prediction Using Profile Data. In *EuroPar Conference*, Manchester, UK, August 2001.
5. S. Melvin and Y. Patt. Enhancing Instruction Scheduling with a Block-structured ISA. *Intl. Journal on Parallel Processing*, 23(3), June 1995.
6. 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.
7. J. Henning. SPEC CPU2000: Measuring Performance in the New Millennium. *IEEE Computer*, 33(7), July 2000.
8. V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A Transparent Dynamic Optimization System. In *the Proceedings of the Conference on Programming Language Design and Implementation*, Vancouver, Canada, June 2000.
9. D. Burger and M. Austin. Simplescalar Tool Set, Version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.
10. 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.
11. 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.
12. V. Kathail, M. Schlansker, and B. Rau. HPL PlayDoh Architecture Specification. Technical Report HPL-93-80, HP Labs, 1994.
13. N. Topham and K. McDougall. Performance of the Decoupled ACRI-1 Architecture: the Perfect Club. In *Intl. Conference on High-Performance Computing and Networking*, Milan, Italy, May 1995.
14. R. Manohar and M. Heinrich. The Branch Processor Architecture. Technical Report CSL-TR-1999-1000, Cornell Computer Systems Laboratory, November 1999.
15. 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.
16. B. Calder and D. Grunwald. Fast and Accurate Instruction Fetch and Branch Prediction. In *Intl. Symposium on Computer Architecture*, Chicago, IL, April 1994.
17. 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.
18. 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.
19. Q. Jacobson, E. Rotenberg, and J. Smith. Path-based Next Trace Prediction. In *Intl. Symposium on Microarchitecture*, Research Triangle Park, NC, December 1997.
20. 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.
21. S. Jourdan et al. Extended Block Cache. In *Intl. Symposium on High-Performance Computer Architecture*, Toulouse, France, January 2000.