A Low Power Front-End for Embedded Processors Using a Block-Aware Instruction Set

Ahmad Zmily and Christos Kozyrakis
Electrical Engineering Department, Stanford University
Stanford, CA 94305, USA
zmily@stanford.edu, kozyraki@stanford.edu

ABSTRACT

Energy, power, and area efficiency are critical design concerns for embedded processors. Much of the energy of a typical embedded processor is consumed in the front-end since instruction fetching happens on nearly every cycle and involves accesses to large memory arrays such as instruction and branch target caches. The use of small front-end arrays leads to significant power and area savings, but typically results in significant performance degradation. This paper evaluates and compares optimizations that improve the performance of embedded processors with small front-end caches. We examine both software techniques, such as instruction re-ordering and selective caching, and hardware techniques, such as instruction prefetching, tagless instruction cache, and unified caches for instruction and branch targets. We demonstrate that, building on top of a block-aware instruction set, these optimizations can eliminate the performance degradation due to small front-end caches. Moreover, selective combinations of these optimizations lead to an embedded processor that performs significantly better than the large cache design while maintaining the area and energy efficiency of the small cache design.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems; C.0 [**General**]: Hardware/software interface

General Terms
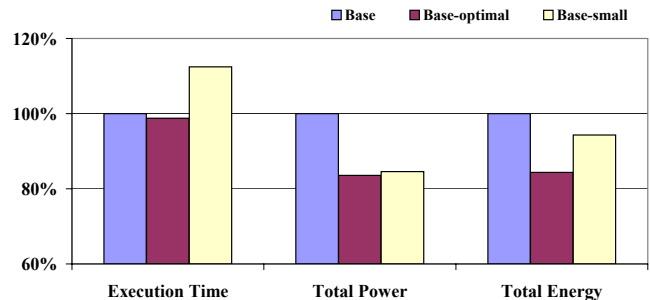
Design, Performance

Keywords

low power front-end, instruction re-ordering, software hints, instruction prefetching, tagless instruction cache, unified instruction cache and BTB

1. INTRODUCTION

Energy, power, and area efficiency are important metrics for embedded processors. Die area and power consumption determine the

Figure 1: Normalized execution time, total power, and total energy consumption for the base design (32-KByte, 32-way I-Cache, 64-entry BTB), the base design with optimal I-Cache and BTB, and the base design with small front-end arrays (2-KByte, 2-way I-Cache, 16-entry BTB). The processor core is similar to Intel's XScale PXA270 and is running benchmarks from the MediaBench and SpecCPU2000 suites. Lower bars present better results.

cost to manufacture, package, and cool the chip. Energy consumption determines if the processor can be used in portable systems. Moreover, embedded processors must also meet the increasing performance requirements of demanding applications such as image, voice, and video processing that are increasingly common in consumer products [25]. Hence, area, power, and energy efficiency must be achieved without compromising performance.

Embedded processors consume a large fraction of their energy in the front-end of their pipeline. The front-end contains several large SRAM structures such as the instruction cache, the branch target buffer (BTB), and the branch predictor, that are accessed on nearly every clock cycle. Such memory arrays are sized to hold a large amount of data in order to obtain good overall performance. For example, the Intel XScale PXA270 processor uses a 32-KByte, 32-way instruction cache and a 128-entry BTB [10]. Nevertheless, different programs exhibit different locality and memory access patterns and even a single program may not need all the available storage at all times. If the processor is executing a tight loop, for example, most of the instruction cache is underutilized as smaller cache could provide the same performance but with lower area, power, and energy requirements. Figure 1 quantifies the total energy and power wasted in the PXA270 processor due to sub-optimal instruction cache and BTB sizing for MediaBench and SpecCPU2000 applications. The optimal configuration is found using a method similar to [26] where a continuum of cache sizes and configurations are simulated. During each cycle, the cache with the lowest power from among those that hit is selected. On average 16% total power

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

Table 1: Normalized power dissipation, area, and access time for different instruction cache configurations over the XScale 32-KByte instruction cache configuration.

and 17% total energy are wasted if the processor uses larger than needed instruction cache and BTB.

Reducing the instruction cache and BTB capacity of embedded processors by a factor of 4 or 8 leads to direct die area and power savings. Table 1 presents the normalized power dissipation, area, and access time for different smaller instruction cache configurations over the 32-KByte, 32-way instruction cache of the PXA270 processor using Cacti [27]. A 2-KByte instruction cache dissipates only 8.4% of the power dissipated by the 32-KByte cache and uses only 4.6% of its area. While the use of smaller arrays reduces die area and power dissipation, several applications will now experience additional instruction cache and BTB misses that will degrade performance and increase energy consumption. Figure 1 quantifies the performance penalty with the smaller instruction cache and BTB sizes (13% on average). Furthermore, the energy savings from accessing smaller arrays are nearly canceled from the cost of operating the processor longer due to the performance degradation.

This paper studies optimization techniques that improve the performance of embedded processors with small front-end arrays. Our goal is to reach or exceed the performance of embedded processors with large caches, while maintaining energy and power consumption close to the optimal design. We evaluate both hardware and software based techniques such as *instruction prefetching* and *re-ordering*, *unified instruction cache and BTB structures*, *tagless instruction caches*, and various forms of *software hints*. Instruction prefetching hides the latency of extra cache misses by fetching instructions ahead of time. Instruction re-ordering attempts to densely pack frequently used instruction sequences in order to improve the locality in instruction cache and BTB accesses. Unifying the instruction cache and the BTB allows a program to flexibly use the available storage as needed without the limitations of a fixed partitioning. Alternatively, the BTB and instruction cache could be organized in such away that the instruction cache tags are no longer required; hence, their area and power overhead can be saved. Finally, compiler generated hints can improve the instruction cache performance by guiding the hardware to wisely use the limited resources.

We explore these front-end optimizations using a block-aware instruction set architecture (BLISS). Previous work [33] has shown that BLISS leads to significant performance and code size advantages for processors with conventionally sized front-end caches. BLISS defines basic block descriptors in addition to and separately from the actual instructions in each program. A descriptor provides the type of the control-flow operation that terminates the basic block, its potential target, the number of instructions in the block, and a pointer to the actual instructions.

In this paper, we explore the front-end optimizations that improve the performance of embedded processors with small front-end caches using the BLISS ISA. BLISS provides a flexible substrate to implement the optimizations efficiently because the de-

scriptors are directly visible to software, provide accurate information for prefetching, and can carry software hints. Hence, BLISS allows significant reorganization of the front-end without modifying the software model. While some of the optimizations can also be implemented with a conventional instruction set, they lead to lower performance benefits and are typically more complex.

Overall, this paper provides the insights and analysis necessary to design the front-end for efficient embedded designs. The specific contributions are:

- we demonstrate that a block-aware architecture allows the implementation of a wide-range of front-end optimizations in a simple and efficient manner.

- we evaluate the front-end optimizations and analyze how they allow an embedded processor with small front-end caches to perform similarly to one with larger structures.

- we demonstrate that combinations of these optimizations further improve the performance and allow the front-end of the processor to achieve power and energy consumption levels close to the optimal design. The best performing configuration allows an embedded processor with small front-end caches to be 9% faster and consume 14% less power and 19% less energy than a similar pipeline with large front-end structures.

- While some optimizations can be implemented using a conventional instruction set, we demonstrate that they are typically more complex and may lead to lower energy and performance benefits compared to BLISS. We compare BLISS with the front-end optimizations to the Filter cache design with similar optimizations. We show that BLISS provides similar power reduction and at the same time provides significant performance and energy improvements.

The rest of the paper is organized as follows. Section 2 provides a brief overview of the BLISS architecture. In Section 3, we present the different front-end optimizations. Section 4 explains the methodology used for evaluation. In Section 5, we evaluate the performance, cost, and total energy benefits of the different front-end optimizations for a configuration similar to the XScale processor. In Section 6, we discuss the related research that this work is based on. Section 7 provides a summary.

2. BLISS OVERVIEW

Before we describe the front-end optimizations, we provide a brief overview of the BLISS architecture. For further details, we refer readers to [33].

The BLISS instruction set provides explicit basic block descriptors (*BBD*) 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 basic blocks, while the second section lists the actual instructions in each block. Figure 2 shows the descriptor format. Each block descriptor 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.

Figure 2 also shows an embedded processor with a BLISS-based front-end that uses a cache for basic block descriptors (BB-cache)
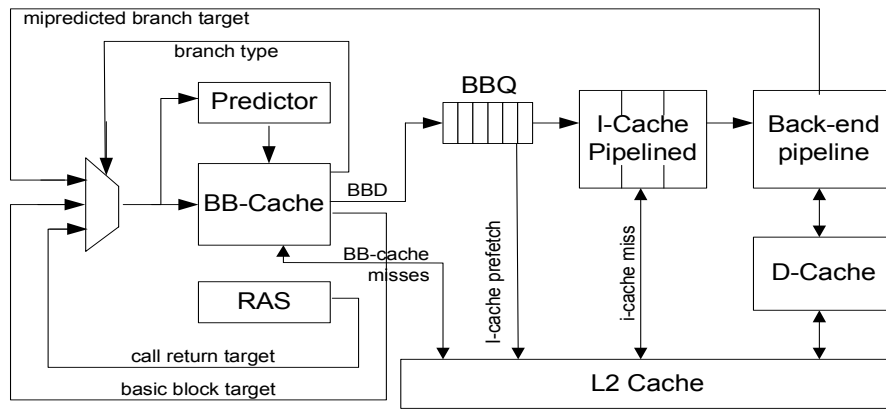
BB Descriptor Format:

Type (4)	Offset (8)	Length (4)	Instruction Pointer (13)	Hints (3)

Type: Basic Block type (type of terminating branch):
   - FT, B, J, JAL, JR, JALR, RET, LOOP
Offset: displacement for PC-relative branches and jumps.
Length: number of instructions in the basic block (0..15)
Instruction pointer: address of the 1st instruction in the block
   bits [15:2]. bits [31:16] in TLB
Size: optional compiler-generated hints used for cache hints
   in this study

Figure 2: The BLISS architecture. The top graph presents an embedded processor based on the BLISS ISA. The bottom graph shows the 32-bit basic block descriptor format.

as a replacement for the BTB. The front-end operates in a decoupled manner. On every cycle, the BB-cache is accessed using the PC. On a miss, the front-end stalls until the missing descriptor is retrieved from the memory hierarchy (L2-cache). On a hit, the BBD and its predicted direction/target are pushed in the *basic block queue (BBQ)*. The predicted PC is used to access the BB-cache in the following cycle. Instruction cache accesses use the instruction pointer and length fields of the descriptors available in the BBQ to retrieve the instructions in the block. If all instructions are in a single cache line, a single cache access per block is sufficient.

Previous work has shown that BLISS leads to significant improvements in performance, energy consumption, and code size [33]. Performance is improved because BLISS tolerates instruction cache latency and improves control-flow prediction [31]. The BBQ decouples control-flow prediction from instruction fetching. Multi-cycle latency for a 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. Since the PC in the BLISS ISA always points to basic block descriptors (i.e. a control-flow instruction), the predictor is only used and trained for PCs that correspond to branches which reduces interference and accelerates training in the predictor.

The improved control-flow prediction accuracy reduces the energy wasted by mispredicted instructions. In addition, energy consumption is further reduced because BLISS allows for energy optimizations in the processor front-end [32]. Each basic block defines exactly the number of instructions needed from the instruction cache. Using segmented word lines for the data portion of the cache, we can fetch the necessary words while activating only the necessary sense-amplifiers in each case. Front-end decoupling tolerates higher instruction cache latency without loss in speculation

accuracy. Hence, we can access first the tags for a set associative instruction cache, and in subsequent cycles, access the data only in the way that hits We can also merge the instruction accesses for sequential blocks in the BBQ that hit in the same cache line, in order to save decoding and tag access energy. Finally, the branch predictor is only accessed after the block descriptor is decoded; hence, predictor accesses for fall-through or jump blocks can be eliminated.

BLISS improves code density by removing redundant sequences of instructions across basic blocks and flexible interleaving of 16-bit and 32-bit instructions at basic block granularity [34]. All instructions in a basic block can be eliminated if the same sequence is present elsewhere in the code. Correct execution is facilitated by adjusting the instruction pointer in the basic block descriptor to point to the unique location in the binary for that instruction sequence. We can also aggressively interleave 16-bit and 32-bit instructions at basic-block boundaries without the overhead of additional instructions for switching between 16-bit and 32-bit modes. The block descriptors identify if the associated instructions use the short or long instruction format.

In this paper, we go beyond previous BLISS studies by introducing hardware and software optimizations that improve efficiency with small front-end structures. While the base BLISS approach is not sufficient to address the performance challenges in such systems, its instruction set and front-end organization provide significant benefits for hardware and software optimizations that can bridge the gap. Many of the optimization techniques covered in this study have been already proposed for conventional instruction sets. In this paper, we demonstrate that BLISS provides a flexible substrate to implement the optimizations efficiently which translates to higher performance and energy improvement compared to

implementing them using conventional instruction sets. We explain the synergy in Section 3.

3. FRONT-END OPTIMIZATIONS

Reducing the instruction cache and BTB capacity of embedded processors by a factor of 4 or 8 leads to direct die area and power savings. However, several applications will now experience additional instruction cache and BTB misses that will degrade performance and increase energy consumption (see Figure 1). This section discusses hardware and software techniques that can reduce the performance degradation. The hardware-based techniques include *instruction prefetching*, *unified instruction cache and BTB structures*, and *tagless instruction caches*. Instruction prefetching hides the latency of extra cache misses by fetching instructions ahead of time. Unifying the instruction cache and the BTB allows a program to flexibly use the available storage as needed without the limitations of a fixed partitioning. Alternatively, the BTB and the instruction cache could be organized in such away that the instruction cache tags are no longer required; hence, their area and power overhead can be saved. The software-based techniques include *instruction re-ordering* and various forms of *software hints*. Instruction re-ordering attempts to densely pack frequently used instruction sequences in order to improve the locality in instruction cache and BTB accesses. Finally, compiler-generated hints can improve the instruction cache performance by guiding the hardware to wisely use the limited resources. The following sections will explain each optimization technique and how it can be easily supported by BLISS.

3.1 Instruction Prefetching (Hardware)

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

Most modern processors only support very basic hardware sequential prefetchers. With a sequential or stream-based prefetcher, one or more sequential cache lines after the currently requested one are prefetched [28, 19]. Stream prefetching only helps with misses on sequential instructions. An alternative approach is to initiate prefetches for cache lines on the predicted path of execution [6]. The advantage of such a scheme is that it can prefetch potentially useful instructions even for non-sequential access patterns as long as branch prediction is sufficiently accurate. Prefetched instructions are typically stored in a separate buffer until the data is used at least once to avoid cache pollution. Most prefetching methods filter out useless prefetches by probing the cache to save bandwidth and power. To avoid adding an additional port, probing is performed only when the instruction cache port is idle.

BLISS supports efficient execution-based prefetching using the contents of the BBQ. The BBQ decouples basic block descriptor accesses from fetching the associated instructions. The predictor typically runs ahead, even when the instruction cache experiences temporary stalls due to a cache miss or when the instruction queue is full. The contents of the BBQ provide an early, yet accurate

view into the instruction address stream and are used to lookup further instructions in the instruction cache. Prefetches are initiated when a potential miss is identified. BLISS also improves prediction accuracy since the PC always points to basic block descriptors and the predictor is only used and trained for PCs that correspond to branches which reduces interference and accelerates training in the predictor [31]. The improved prediction accuracy makes the execution-based prefetching scheme even more effective. Prefetching, if not accurate, leads to additional L2-cache accesses that can increase the L2-cache power dissipation.

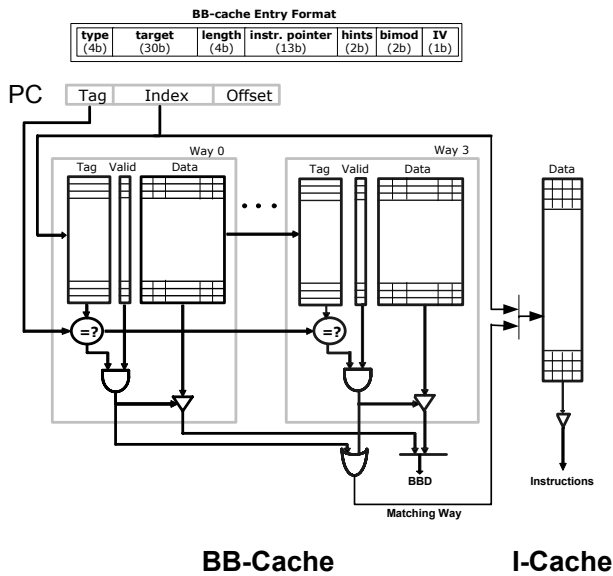## 3.2 Unified I-Cache and BTB (Hardware)

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

The BLISS front-end can be configured with a unified instruction cache and BB-cache storage as both instructions and descriptors are part of the architecturally-visible binary code. Each line in the unified cache holds either a few basic block descriptors or a few regular instructions. The unified cache can be accessed by both the descriptor fetch and the instruction fetch units using a single access port. Instruction fetch returns multiple instructions per access (up to a full basic block) to the back-end pipeline and does not need to happen on every cycle. On the remaining cycles, we perform descriptor fetches. For the embedded processors we studied in Section 5, sharing a single port for instruction and descriptor fetches had a negligible impact on performance.

With a conventional architecture, on the other hand, storing BTB and instruction cache entries in a single structure is more challenging as the same program counter is used to access both structures. This implies that extra information is required to be stored in the unified cache to differentiate between BTB and instruction entries. In addition, the two entries map to the same cache set, causing more conflicts. The BTB and instruction cache are also accessed more frequently as basic block boundaries are not known until instruction decoding. Hence, sharing a single port is difficult.

3.3 Tagless I-Cache (Hardware)

In previous work [32], we showed that we could eliminate the data access for all but the way that hits by accessing the tag arrays first then the data array in a subsequent cycle. Now we will focus on eliminating the instruction cache tags altogether (storage and access). BLISS provides an efficient way to build an instruction cache with no tag accesses by exploiting the tags checks performed on descriptor accesses. This improves instruction cache access time, reduces its energy consumption significantly, and eliminates the area overhead of tags. The new tagless instruction cache is organized as a direct mapped cache, with only the data component. Figure 3 illustrates the organization of this cache. For each basic block descriptor in the BB-cache, there is only one entry in the tagless instruction cache which can hold a certain number of instructions, 4 in our experiments. A flag bit is used in each descriptor in the BB-cache entry to indicate if the corresponding entry in the tagless instruction cache has valid instructions or not. This flag is initialized during BB-cache refill from L2-cache and is set after the instructions are fetched from the L2-cache and placed in the tagless instruction cache. Moreover, the flag that indicates if the entry

BB-cache Entry Format						
type (4b)	target (30b)	length (4b)	instr. pointer (13b)	hints (2b)	bimod (2b)	IV (1b)

Figure 3: The organization of the tagless instruction cache with BLISS.

in the tagless cache is valid or not can be used by the prefetching logic. This eliminates the need to probe the cache and improves the overall performance of the prefetcher.

The operation of the BLISS front-end with the tagless cache is very similar to what we explained in Section 2 except the way the instruction cache is accessed. On a BB-cache miss, the missing descriptors are retrieved from the L2-cache. At that stage, the instruction valid bits (**IV**) are initialized for those descriptors indicating that their associated instruction cache entries are invalid. The instruction fetch unit uses the valid bit to determine how to access the instruction cache. If the instruction valid bit is not set, the instructions are retrieved from the L2-cache using the instruction pointer available from the descriptor. Once the instructions are retrieved and placed in the instruction cache, the valid bit for the corresponding descriptor is set. If the instruction valid bit is set, the instructions are retrieved from the instruction cache using the index field of the PC and the index of the matching BB-cache way. For basic blocks larger than 4 instructions, only the first four instructions are stored in the instruction cache. In the applications studied in Section 5, 68% of the executed basic blocks include 4 instructions or less. Similar to the victim cache, we use a 4-entry fully associative cache to store the remaining instructions. This victim cache is accessed in a subsequent cycle and is tagged using the PC. In a case of a miss, the instructions are brought from the L2-cache.

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

3.4 Instruction Re-ordering (Software)

Code re-ordering at the basic block level is a mature method that tunes a binary to a specific instruction cache organization and im-

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

Pettis and Hansen suggested a bottom-up block-level positioning algorithm [20]. In their approach, they split each procedure into two procedures, one with the commonly used basic blocks and one with the rarely used basic blocks ("fluff"). The infrequently executed code is replaced with a jump to the relocated code. Additionally, a jump is inserted at the end of the relocated code to transfer control back to the commonly executed code. Within each of the two procedures, a control-flow graph is used to from chains of basic blocks based on usage counts. The chains are then placed making fall through the likely case after a branch.

Basic block re-ordering is easily supported by BLISS using the explicit block descriptors. Blocks of instructions can be freely re-ordered in the code segment in any desired way as long as we update the instruction pointers in the corresponding block descriptors. Figure 4 presents an example to illustrate this optimization. The two instructions in the second basic block in the original code are rarely executed. Therefore, they can be moved to the end of the code as long as the instruction pointer for BBD2 is updated. Compared to re-ordering with conventional architectures, this provides two major benefits. First, there is no need to split the procedure or introduce additional jump instructions for control transfers between the commonly and the less commonly used code (fewer static and dynamic instructions). The pointers in the block descriptors handle control transfers automatically. Second, re-ordering basic blocks does not affect branch prediction accuracy for BLISS, as the vital information for speculation is included in the basic block descriptors available through the BB-cache (block type, target offset). On a conventional architecture, re-ordering blocks may change the number of BTB entries needed and the conflicts observed on BTB accesses.

3.5 Cache Placement Hints (Software)

Conventional caches are designed to be managed purely by hardware. Hardware must decide where to place the data and which data to evict during cache replacement. A consequence is that the cache resources may not be optimally utilized for a specific benchmark, leading to poor cache hit rate. Compilers and profile-based tools can help the processor with selecting the optimal policies in order to achieve the highest possible performance using the minimal amount of hardware. Hints can indicate at which cache levels it is profitable to retain data based on their access frequency, excluding infrequent data from the first level cache. Hints can also guide the hardware placing data in the cache to avoid conflicts, or improve the cache replacement decisions by keeping data with higher chance of reuse.

A compiler can attach hints to executable code at various granularities, with every instruction, basic block, loop, function call, etc. BLISS provides a flexible mechanism for passing compiler-generated hints at the granularity of basic blocks. The last field of the basic block descriptor contains optional compiler-generated hints. Specifying hints at the basic block granularity allows for fine-grain information without increasing the length of all instruction encodings or requiring additional, out-of-band, instructions that carry the hints. Hence, hints can be communicated without modifying the conventional instruction stream or affecting static

BB descriptors

```
BBD1: BR_F, BBD3, ,  ●────→    beq r8, r1
BBD2: JAL , foo,  2, ●────→    add r3, r2, r8
BBD3: BR_B, BBD1, 3, ●         addiu r17, r0, 1
BBD4:             ●            lw r6,1492(r30)
                              addu r4, 0, r2
                              sub r3, r2, r8
                              addiu r17, r0, 1
```

(a)

BB descriptors

```
BBD1: BR_F, BBD3, ,  ●────→    beq r8, r1
BBD2: JAL , foo,  2, ●         lw r6,1492(r30)
BBD3: BR_B, BBD1, 3, ●         addu r4, 0, r2
BBD4:             ●            sub r3, r2, r8
                              addiu r17, r0, 1
                              add r3, r2, r8
                              addiu r17, r0, 1
```
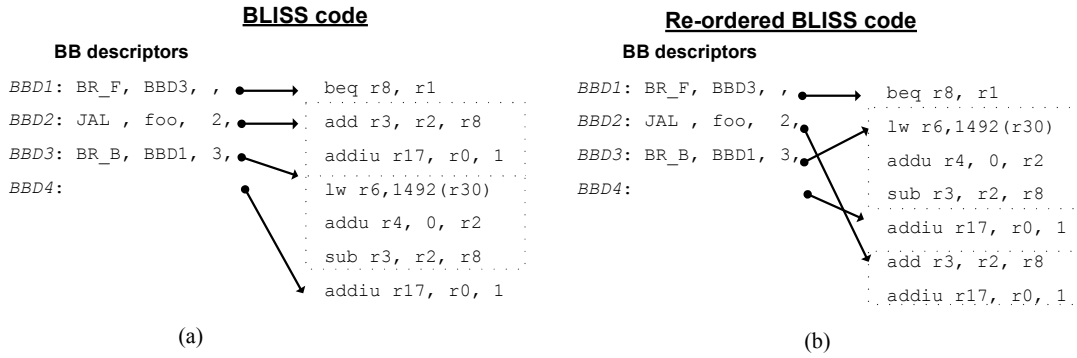
(b)

Figure 4: Example to illustrate the instruction re-ordering optimization with BLISS. (a) Original BLISS code. (b) Re-ordered BLISS code. For illustration purposes, the instruction pointers in basic block descriptors are represented with arrows.

Front-End Parameters		
	XScale PXA270	
	Base	**BLISS**
Fetch Width	1 inst/cycle	1 BB/cycle
Regular I-cache	32 KBytes, 32-way, 32B blocks	
	1 port, 2-cycle access	
Small I-cache	2 KBytes, 2-way, 32B blocks	
	1 port, 2-cycle access	
BTB/BB-cache		
Regular	64-entry, 4-way	64-set, 4-way
Small	16-entry, 2-way	16-set, 2-way
BBQ	–	4 entries
Common Processor Parameters		
	XScale PXA270	
Execution	single-issue, in-order with 1 INT & 1 FP unit	
Predictor	256-entry bimod with 8 entry RAS	
D-cache	32 KBytes, 4-way, 32B blocks	
	1 port, 2-cycle access	
L2-cache	128 KBytes, 4-way, 64B blocks	
	1 port, 5-cycle access	
Main memory	30-cycle access	

Table 2: The microarchitecture parameters for base and BLISS processor configurations used for power and area optimization experiments.

or dynamic instruction counts. Furthermore, since descriptors are fetched early in the pipeline, the hints can be useful with decisions with most pipeline stages, even before instructions are decoded.

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

4. METHODOLOGY

Table 2 summarizes the key architectural parameters for the base and BLISS processor configurations used for evaluation. Both are modeled after the Intel XScale PXA270 [10]. We fully model all contention for the L2-cache bandwidth between BB-cache misses and instruction cache or data cache misses. For fair energy comparison, the base design is modeled with serial instruction tag and data accesses to eliminate the data access for all but the way that hits. We have also performed experiments with a high-end embedded core comparable to the IBM PowerPC 750GX and the achieved results are consistent.

Our simulation framework is based on the Simplescalar/PISA 3.0 toolset [5], which we modified to add the BLISS front-end model. All front-end optimizations explained in Section 3 are fully modeled in the simulations. For energy measurements, we use the Wattch framework at the cc3 power model [4], which we also modified to accurately capture all of the optimizations. Energy consumption was calculated for a $0.10\mu m$ process with a 1.1V power supply. The reported *Total Energy* includes all the processor components (front-end, execution core, and all caches). For performance, we report IPC, ignoring the fact that processors with smaller caches may be able to run at higher clock frequencies than processors with larger caches. We study 10 benchmarks form Media-Bench and SpecCPU2000 suites. The selected benchmarks have relatively high instruction cache or BTB miss rates. The benchmarks are compiled at the -O2 optimization level using gcc. We did not include the code size optimizations in [34]. MediaBench programs are simulated to completion and for the SpecCPU2000 programs we skipped 1 billion instructions and simulated 1 billion instructions for detailed analysis. For benchmarks with multiple datasets, we run all of them and calculate the average.

5. EVALUATION

This section presents the performance, total energy, and cost evaluation results for the different front-end optimizations using BLISS.

5.1 Performance Analysis

Figure 5 compares the IPC of BLISS with small caches and the various optimizations to that of the base design with large caches (IPC of 1.0). We only present a single combination of optimizations, the best performing one (prefetching + instruction re-ordering + unified cache + redistribute cache hints). For reference, the average normalized IPC for various other configurations is: 0.87 for the base design with small caches, **0.91** for the base design with small
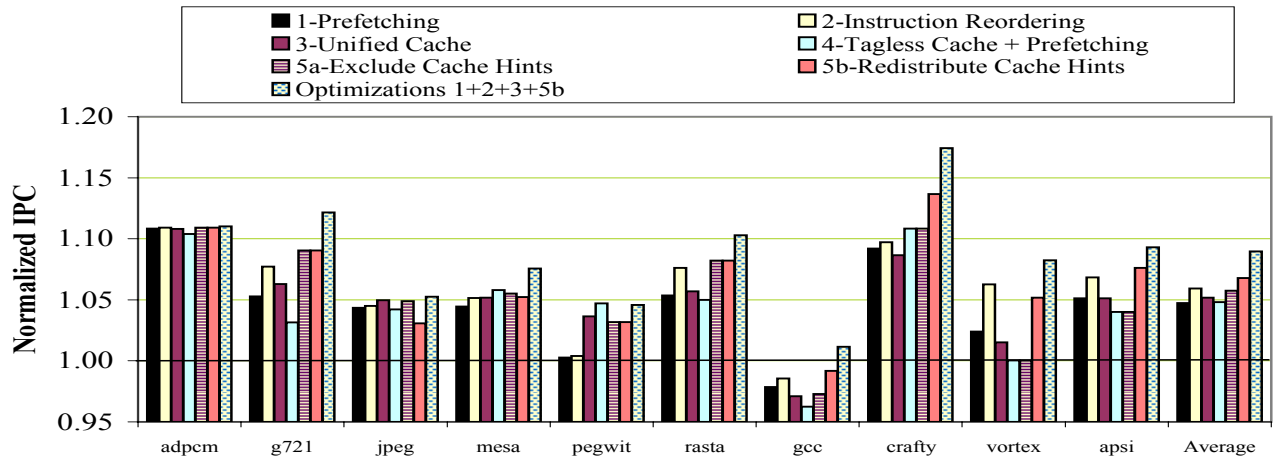
Figure 5: Normalized IPC for BLISS with the different front-end optimizations over the base. The BLISS design uses the small I-cache and BB-cache. The base design uses the regular I-cache and BTB. The 1.0 line presents the base design. Higher bars present better performance.
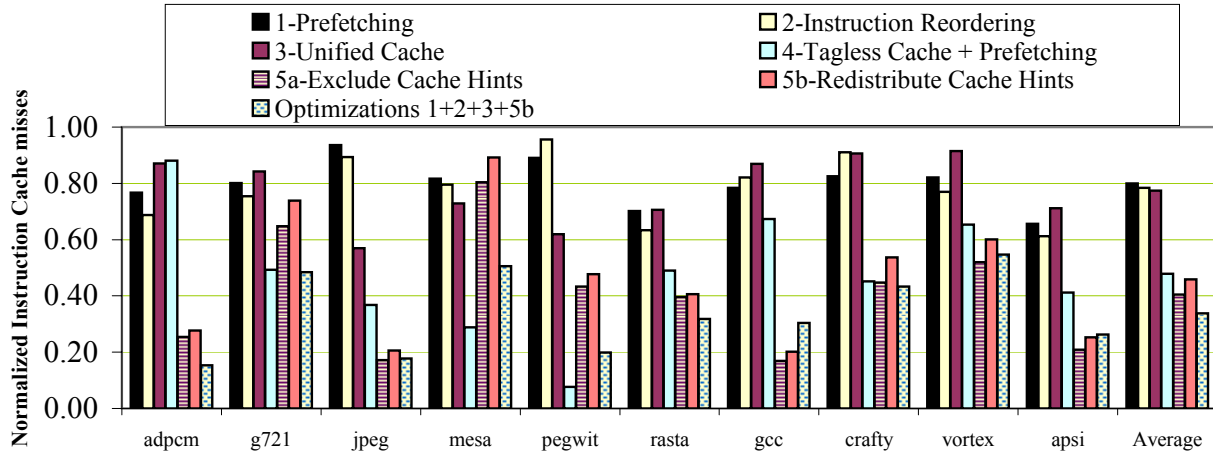

Figure 6: Normalized number of instruction cache misses for BLISS with the different front-end optimizations over the base. The BLISS design uses the small I-cache and BB-cache. The base design uses the small I-cache and BTB. Lower bars present better results.

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

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

with prefetching to deal with conflict misses. Software hints tend to provide a consistent improvement for all of the benchmarks. The redistribute cache hints achieve slightly better performance than the exclude cache hints.

To understand the effectiveness of each technique in reducing the performance impact of the small instruction cache, we look at the instruction cache miss rates for the different optimizations. Figure 6 presents the normalized number of instruction cache misses for BLISS with the different front-end optimizations over the base design with the small instruction cache. The reduction in instruction cache misses with prefetching, instruction re-ordering, and unified cache is consistent across most benchmarks with a 20% average. For the tagless instruction cache + prefetching, the decrease varies and largely depends on the basic block average size. Both of the software cache placement hints with prefetching significantly reduce the number of cache misses with an average of 58%. Finally, the best combination of the optimizations (prefetching + instruction re-ordering + unified cache + redistribute cache hints) achieves 66% reduction.
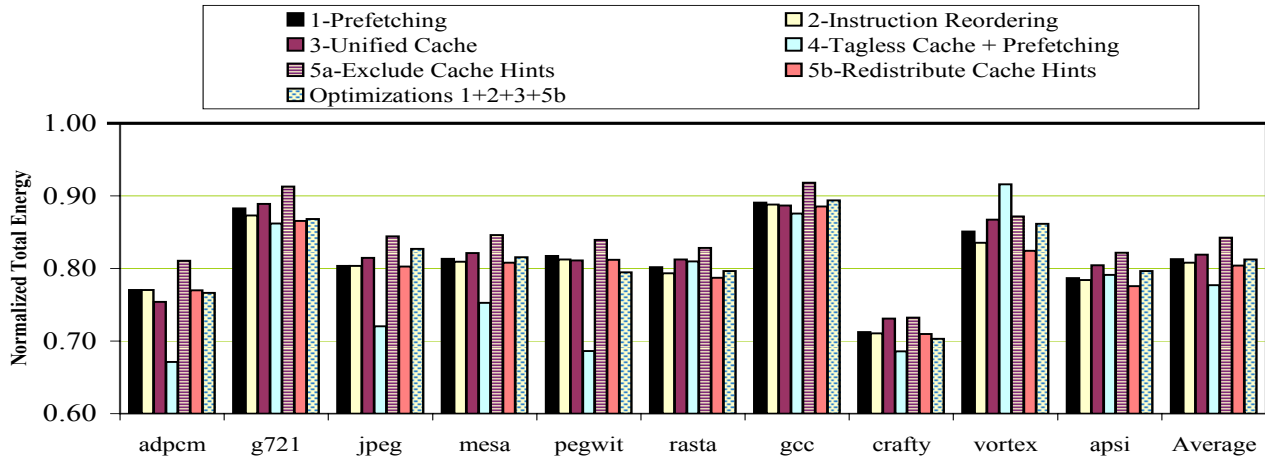
Figure 7: Normalized total energy comparison for BLISS with the different front-end optimizations over the base. The BLISS design uses the small I-cache and BB-cache. The base design uses the regular I-cache and BTB. The 1.0 line presents the base design. Lower bars present better results.

5.2 Cost Analysis

Power and die area determine the cost to manufacture and package the chip. The front-end consumes a large fraction of the power budget as it includes large memory structures that are accessed nearly every cycle. For example, the Intel XScale PXA270 processor consumes 20% of it power budget in the front-end itself. Table 3 quantifies the potential cost reduction of using small front-end structures for the XScale PXA270 processor in Table 2. It presents the normalized power and area of the small front-end structures over the large structures for the XScale configuration. We also report the normalized access times for the small front-end structures. However, we ignore the fact that the processor with the small caches can run at higher clock frequency. The small instruction cache only dissipates 8.4% of the power dissipated by the large cache and uses only 4.6% of its area. The small predictor tables dissipate 75.4% of the power dissipated by the larger structures and use only 47.5% of the area. The small instruction cache access time is also half of the access time for the large cache.

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

Table 3: Normalized power dissipation, area, and access time for the small instruction cache and predictor tables over the large structures of the XScale configuration.

5.3 Total Energy Analysis

Figure 7 compares the total energy of BLISS with small caches and the various optimizations to that of the base design with large caches (energy of 1.0). Lower energy is better. For reference, the average total energy for other configurations is: 0.95 for the base design with small caches, **0.93** for the base design with small caches and prefetching, and **0.88** for BLISS with large caches.

With all optimizations, BLISS with small caches consumes less energy than the base with small or large caches. The combined optimizations lead to an energy consumption of 81%. The tagless instruction cache configuration provides significant energy benefits for several benchmarks (`adpcm`, `jpeg`, `mesa`, `pegwit`) as it

eliminates redundant tag accesses. However, for `vortex`, the tagless instruction cache has the highest energy consumption. This is due to the fact that `vortex` has large basic blocks that will require to be prefetched and placed in the small victim cache. For the remaining optimizations, energy consumption tracks the IPC behavior.

5.4 Comparison Analysis to Hardware-based Techniques

Many techniques have been proposed to save the front-end power without the need for a new ISA. One such example is the Filter cache design proposed by Kin et al. [13]. A Filter cache is a tiny cache introduced as the first level of memory in the instruction memory hierarchy.

Many of the front-end optimizations that are presented in Section 3 can also be implemented with a conventional instruction set using the Filter cache. Figure 8 summarizes the comparison between BLISS with the combined optimizations (unified cache + prefetching + instruction re-ordering + redistribute hints) to the base design with (Filter cache + prefetching + instruction re-ordering + selective caching hints). Note that similar front-end optimizations and cache sizes are used with both designs. The base XScale configuration with the full-sized instruction cache and BTB is shown as a reference. We also show the results for the base design with optimally-sized caches. We use a method similar to [26] to quantify the amount of energy wasted due to sub-optimal cache sizes. A continuum of cache sizes and configurations are simulated. During each cycle, the cache with the lowest power from among those that hit is selected.

BLISS with the front-end optimizations provides similar total power reduction to the Filter cache design and the base design with optimally-sized caches (14% savings). It also provides similar total energy savings to the optimally-sized design (19% reduction). The small advantage is due to the more efficient access of instruction cache in the BLISS base model [31]. More important, the power and energy savings do not lead to performance losses as it is the case for the base design with the Filter cache. BLISS provides a 9% performance improvement over the base design with large caches and a 12% performance improvement over the base design with Filter cache and the combined front-end optimizations. The performance advantage is due to two reasons. First, the efficient imple-
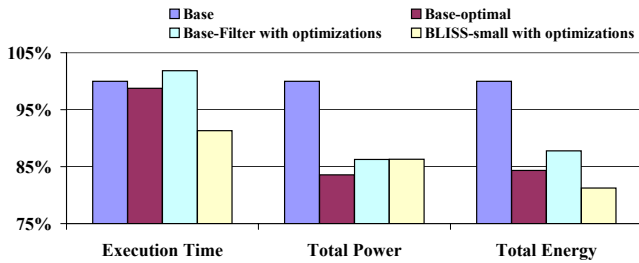
Figure 8: Average execution time, total power, and total energy consumption for base design (with large caches), base design (with optimal caches), base design (with Filter cache and a combination of front-end optimizations), and BLISS (with small caches and a combination of front-end optimizations). Lower bars present better results.

mentation of front-end optimizations mitigates the negative effects of the small instruction cache and BTB. Second, the block-aware architecture allows for higher prediction accuracy that provides the additional performance gains [31]. In addition, BLISS provides 7% energy improvement over the base design with Filter cache and the combined front-end optimizations. Overall, BLISS with small caches and front-end optimizations improves upon the Filter cache with comparable front-end optimizations by offering similar power reduction at superior performance and energy consumption (12% performance and 7% total energy improvements).

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

6. RELATED WORK

Significant amount of front-end research focused on instruction cache optimizations of microprocessor-based systems because of the cache's high impact on system performance, cost, and power. The use of a tiny (Filter) cache to reduce power dissipation was proposed by Kin et al. [13]. Bellas et al. [2] proposed using a profile-aware compiler to map frequent loops into the Filter cache to reduce the performance overhead. BLISS provides similar power reduction as the Filter cache design and at the same time improves performance and energy consumption. Lee et al. [14] suggested using a tiny tagless loop cache with a controller that dynamically detect loops and fill the cache. The loop cache is an alternative to the first level of memory which is only accessed when a hit is guaranteed. Since the loop cache is not replacing the instruction cache, their approach improves the energy consumption with small performance, area, and total power overhead. Rose et al. [8] evaluated different small cache designs.

Many techniques have been proposed to reduce the instruction cache energy. Some of the techniques include way prediction [22], selective cache way access [1], sub-banking [7], and tag comparison elimination [18], Other research has focused on reconfigurable caches [23] where a subset of the ways in a set-associative cache or a subset of the cache banks are disabled during periods of modest cache activity to reduce power. Using a unified reconfigurable cache has also shown to be effective in providing greater levels of hardware flexibility [16]. Even though reconfigurable caches are effective in reducing energy consumption, they have negligible effect on reducing the peak power or the processor die area.

Many prefetching techniques have been suggested to hide the latency of cache misses. The simplest technique is the sequential prefetching [28, 19]. In this scheme, one or more sequential cache lines that follow the current fetched line are prefetched. History-based schemes [29, 12] use the patterns of previous accesses to initiate the new prefetches. The execution-based scheme has been proposed as an alternative approach [24, 6]. In this scheme, the prefetcher uses the predicted execution path to initiate accesses. Other types of prefetching schemes include wrong path prefetching [21] and software cooperative approach [15]. In the later scheme, the compiler inserts software prefetches for non-sequential misses. BLISS enables highly accurate execution-based prefetching using the contents of the BBQ.

Much research exists at exploring the benefit of code re-ordering [30]. Most of the techniques use a variation of the code positioning algorithm suggested by Pettis and Hansen [20]. Several researchers have also worked on using software-generated hints to improve the performance and power of caches [11, 17, 3, 9]. BLISS efficiently enables instruction re-ordering with no extra overhead and no impact on speculation accuracy. Moreover, the architecturally visible basic block descriptors allow communicating software hints without modifying the conventional instruction stream or affecting its instruction code footprint.

7. CONCLUSIONS

This paper evaluated several front-end optimizations that improve the performance of embedded processors with small front-end caches. Small caches allow for an area and power efficient design but typically lead to performance challenges. The optimizations included instruction prefetching and re-ordering, selective caching, tagless instruction cache, and unified instruction and branch target caches. We built these techniques on top of the block-aware instruction set (BLISS) architecture that provides a flexible platform for both software and hardware front-end optimizations. The best performing combined optimizations (prefetching + instruction re-ordering + unified caches + redistribute cache hints) allow an embedded processor with small front-end caches to be 9% faster and consume 14% less power and 19% less energy than a similar pipeline with large front-end structures. While some of the optimizations can also be implemented with a conventional instruction set, they lead to lower performance benefits and are typically more complex. The BLISS ISA-supported front-end outperforms (12% IPC and 7% energy) a front-end with a conventional ISA with Filter cache and similar front-end optimizations. Overall, BLISS allows for low power and low cost embedded designs in addition to performance, energy, and code size advantages. Therefore, it can be a significant design option for embedded systems.

8. REFERENCES

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

[2] N. Bellas, I. Hajj, C. Polychronopoulos, and G. Stamoulis. Energy and Performance Improvements in Microprocessor Design using a Loop Cache. In *The Proceedings of Intl. Conference on Computer Design*, pages 378–383, Washington, DC, October 1999.

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

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

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

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

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

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

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

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

[11] P. Jain, S. Devadas, D. Engels, and L. Rudolph. Software-Assisted Cache Replacement Mechanisms for Embedded Systems. In *The Proceedings of Intl. Conference on Computer-Aided Design*, pages 119–126, San Jose, CA, November 2001.

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

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

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

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

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

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

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

[19] G.-H. Park, O.-Y. Kwon, T.-D. Han, S.-D. Kim, and S.-B. Yang. An Improved Lookahead Instruction Prefetching. In

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

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

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

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

[24] G. Reinman, B. Calder, and T. Austin. Fetch Directed Instruction Prefetching. In *The Proceedings of Intl. Symposium on Microarchitecture*, pages 16–27, Haifa, Israel, Nov. 1999.

[25] C. Rowen. *Engineering the Complex SOC*. Prentice Hall, 2004.

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

[27] P. Shivakumar and N. P. Jouppi. Cacti 3.0: An Integrated Cache Timing, Power, Area Model. Technical Report 2001/02, Compaq Western Research Laboratory, Aug. 2001.

[28] J. E. Smith and W.-C. Hsu. Prefetching in Supercomputer Instruction Caches. In *The Proceedings of Conference on Supercomputing*, pages 588–597, Minneapolis, MN, November 1992.

[29] V. Srinivasan, E. S. Davidson, G. S. Tyson, M. J. Charney, and T. R. Puzak. Branch History Guided Instruction Prefetching. In *The Proceedings of Intl. Symposium on High-Performance Computer Architecture*, pages 291–300, Nuevo Leone, Mexico, January 2001.

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

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

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

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

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

The Proceedings of High-Performance Computing on the Information Superhighway, pages 712–715, Seoul, South Korea, May 1997.