Simultaneously Improving Code Size, Performance, and Energy in Embedded Processors

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

Abstract

Code size and energy consumption are critical design concerns for embedded processors as they determine the cost of the overall system. Techniques such as reduced length instruction sets lead to significant code size savings but also introduce performance and energy consumption impediments such as additional dynamic instructions or decompression latency. In this paper, we show that a block-aware instruction set (BLISS) which stores basic block descriptors in addition to and separately from the actual instructions in the program allows embedded processors to achieve significant improvements in all three metrics: reduced code size **and** *improved performance* **and** *lower energy consumption.*

1 Introduction

Code density and energy consumption are critical efficiency metrics for embedded processors. Code size determines the amount and cost of on-chip or off-chip memory necessary for program storage. Instruction memory is often as expensive as the processor itself. Energy consumption dictates if the processor can be used in portable or deeply embedded systems for which battery size and lifetime are vital parameters. However, the big challenge with embedded processors is that code density and energy efficiency must be achieved in addition to high performance. Demanding applications such as image, voice, and video processing are increasingly common in modern embedded systems. Hence, high performance embedded processors are necessary to provide programmable support for the current and future demanding applications without the need for expensive and inflexible custom logic [19].

An effective technique for code size reduction is the use of short instruction sets such as the 16-bit MIPS-16 or Thumb-2 [10, 17]. However, a short instruction format implies access to a limited set of registers, limited number of opcodes, and a very short immediate and offset field. These limitations lead to an increased number of dynamic instructions and result in significant performance losses. Figure 1 shows the impact of using a 16-bit ISA on the average code size, execution time, and energy consumption for a simple embedded processor like the XScale PXA270 running the MediaBench applications. The 16-bit instructions lead to 41% code size savings at the cost of 11% and 13% higher execution time and energy consumption. It is possible to recover the performance and energy overhead by selectively using 16-bit instructions only for non-critical sections of the code using a few overhead instructions to specify switches between the two formats [6, 11]. As shown in Figure 1, selective use of short instructions maintains the code size savings and restores the performance and energy consumption of the original, 32-bit code.

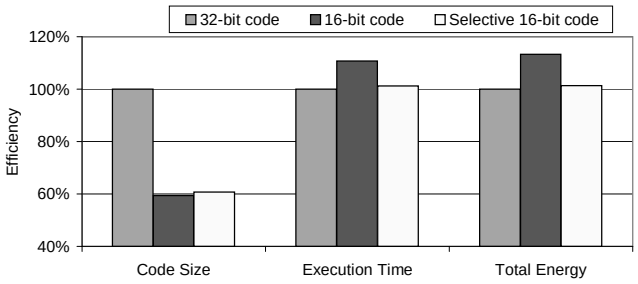
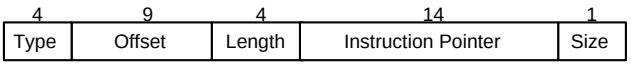


Figure 1: Code size, execution time, and total energy consumption for 32-bit, 16-bit, and selective 16-bit executables for a processor similar to Intel's XScale PXA270 processor running the MediaBench benchmarks. Lower bars present better results.

This paper examines the use of a block-aware instruction set (BLISS) to *improve all three efficiency metrics* for embedded processors at the same time: smaller code size **and** better performance **and** lower energy consumption. BLISS defines basic block descriptors in addition to and separately from the actual instructions in each program [21, 22]. 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. Even though it seems counter-intuitive, the use of additional block descriptors leads to significant reductions



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] are stored in the TLB

Size : flag to indicate the size of instructions in the block (16-bit or 32-bit)

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

in the code size. First, we can easily remove all instructions in a basic block 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. Second, we can 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 previous work we have shown the performance and energy advantages of BLISS [21, 22]. In this paper, we show that BLISS also enables significant code size optimizations without compromising its performance and energy benefits. We demonstrate that the code size optimizations allow BLISS to reach a compression ratio of 60% (40% reduction in code size) over a conventional 32-bit instruction set with a 10% average performance and 21% total energy improvements. The significant performance and energy improvements in addition to the code size savings allow BLISS to compare favorably to conventional techniques for selective use of 16-bit instructions.

2 BLISS Overview

Our proposal is based on a *block-aware instruction set (BLISS)* that explicitly describes basic blocks [21]. 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 32-bit 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 indicates if the actual instructions in the block use 16-bit or 32-bit encoding. With BLISS, 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 another descriptor (sequential or branch target). Precise exceptions are supported similar to [16].

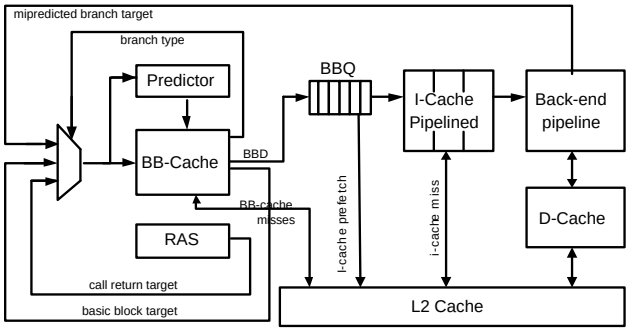


Figure 3: A decoupled front-end processor based on the BLISS ISA.

The BLISS ISA suggests a 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 [21]. The BLISS front-end operation is simple. On every cycle, the BB-cache is accessed using the PC. On a miss, the front-end stalls until the missing descriptor is retrieved from the memory hierarchy (L2-cache). On a hit, the BBD and its predicted direction/target are pushed in the *basic block queue (BBQ)*. The predicted PC is used to access the BB-cache in the following cycle. I-cache accesses use the instruction pointer, length and size fields in the descriptors available in the BBQ.

The BLISS front-end allows for several performance optimizations[21]. The contents of the BBQ provide an early view into the instruction address stream and can be used for instruction prefetching that hides the latency of I-cache misses. The BBQ decouples control-flow prediction from instruction fetching. Multi-cycle latency for a large I-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. Moreover, BLISS allows for energy optimizations in the processor front-end [22]. Each basic block defines exactly the number of instructions needed from the I-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. 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.

3 Code Size Optimizations

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

3.1 Basic Optimizations

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

3.2 Block Subsetting

BLISS facilitates the removal of repeated sequences of instructions [5]. All instructions in a basic block can be eliminated, if the exact sequence of the instructions can be found elsewhere in the binary. We maintain the separate descriptor for the block but change its instruction pointer to point to the unique location in the binary for that instruction sequence. We refer to this optimization as *Block Subsetting*. Block subsetting leads to significant code size improvements because programs frequently include repeated code patterns. Moreover, the compiler generates repeated patterns for tasks like function setup, stack handling, and loop setup. By removing jump and branch instructions, the basic code size optimizations expose more repeated instruction sequences that block subsetting can eliminate. Instruction similarity is also improved because BLISS stores branch offsets in the BBDs and not in regular instructions.

Block subsetting can affect performance both ways by interfering with the I-cache hit rate. It can reduce the hit rate as it decreases spatial locality in instruction references. Two sequential basic blocks may now point to instruction sequences in non-sequential locations. However, the BLISS front-end can tolerate higher I-cache miss rates as it allows for effective prefetching using information in the basic block descriptors (see Section 2). Block subsetting can also improve cache performance as it reduces the cache capacity wasted on repeated sequences.

3.3 Block-level Interleaving of 16/32-bit Code

Interleaving 16-bit and 32-bit instruction formats can lead to good code compression without performance loss. MIPS-16 allows mixing of 16-bit and 32-bit instructions at the function-level granularity. A special JALX instruction is used to switch between functions with 16-bit and 32-bit instructions. However, function-level granularity is restrictive as many functions contain both performance critical and non-critical code. Alternatively, one can interleave 16-bit and 32-bit code at instruction granularity [6, 11, 17]. Special instructions are still necessary to switch between the 16 and 32-bit sections, hence there is an overhead for each switch.

BLISS provides a flexible mechanism for interleaving 16-bit and 32-bit code at the granularity of basic blocks. This is significantly better than the function-level granularity in MIPS-16. It is also as flexible as the instruction-level granularity because either all instructions in a basic block are frequently executed (performance critical) or none of them is. The last field of the basic block descriptor provides a flag to specify if the block contains 16-bit or 32-bit instructions (see Figure 2). No new instructions are required to specify the switch between the 16-bit and 32-bit modes. Hence, frequent switches between the two modes incur no additional runtime penalty.

4 Methodology

Table 1 summarizes the key architectural parameters used for evaluation which is modeled after the Intel XScale PXA270 [8]. We have also performed detailed experiments for a high-end embedded core comparable to the IBM PowerPC 750GX [7] and the achieved results are consistent. For BLISS, we split the baseline I-cache capacity between regular instructions (3/4 for BLISS I-cache) and block descriptors (1/4 for BB-cache). The smaller BLISS I-cache does not incur more misses as 17% of the original 32-bit instructions are eliminated from the BLISS code by the simple code size optimizations. We fully model all contention for the L2-cache bandwidth between BB-cache misses and I-cache or D-cache misses.

Our simulation framework is based on the Simplescalar/PISA 3.0 toolset [4], which we modified to add the BLISS front-end model. For energy measurements, we use the Wattch framework with the cc3 power model [3]. 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). We study 10 MediaBench benchmarks [13] compiled at the -O2 optimization level using gcc and simulated to completion.

	XScale PXA270	
	Base	**BLISS**
Fetch Width	1 inst/cycle	1 BB/cycle
BTB	32-entry, 4-way	–
BB-cache	–	8 KBytes, 4-way 32B Blocks , 1-cycle access
I-cache	32 KBytes, 32-way 32B Blocks, 2-cycle access	24 KBytes, 24-way 32B Blocks, 2-cycle access
BBQ	–	4 entries
Execution	single-issue, in-order with 1 INT & 1 FP unit	
Predictor	256-entry bimod with 8 entry RAS	
IQ/RUU/LSQ	16/32/32 entries	
D-cache	32 KBytes, 4-way, 32B blocks, 1 port, 2-cycle access	
L2-cache	256 KBytes, 4-way, 64B blocks, 1 port, 5-cycle access	
Main memory	30-cycle access	

Table 1: The microarchitecture parameters for the simulations.

Benchmark	MIPS32	BLISS basic Optimization		Block Subsetting	Interleaving 16/32 Blocks	
	Code Size (KB)	# BBs	J/B Inst. removed	# Inst. eliminated	% of Inst. 16-bit	Extra Inst.
adpcm	36	2607	1671	2536	94%	730
epic	64	4345	2808	4771	96%	823
g721	42	2942	1920	3015	93%	750
gsm	69	4409	2866	4483	94%	948
jpeg	109	6609	4535	8033	96%	1322
mesa	430	24054	16692	36628	95%	6305
mpeg2.dec	77	5128	3514	5168	96%	1242
mpeg2.enc	104	6502	4390	6895	95%	1820
pegwit	74	4094	2735	5229	95%	1666
pgp	201	14273	10582	14889	96%	1242
rasta	226	13788	10628	19191	96%	1040

Table 2: Statistics for code size optimizations.

Our study uses a BLISS version of the MIPS ISA. The BLISS executables are generated from MIPS binaries using a static binary translator, which can handle arbitrary programs from high-level languages. BLISS executables could also be generated using a dynamic compilation framework [1]. We perform the basic code size optimizations during the base translation. Block subsetting is performed in an additional pass over the BLISS code. If all of the instructions of a basic-block appear elsewhere in the code stream, the instructions are eliminated and the descriptor pointer is updated. Although instruction rescheduling and register reallocation might help in identifying additional repetitions [5], they are not considered in this study.

To determine which basic blocks will use 16-bit encoding for their instructions, we employ the static profitability-based heuristic proposed in [6]. The heuristic makes the trade-off between increased register pressure and increased code size. Instructions in 16-bit format can only access 8 registers and may lead to performance loss due to register spilling. The heuristic tries to achieve similar code size reduction to what is possible with exclusive use of 16-bit instructions without impacting performance.

5 Evaluation

This section presents the code size, performance, and energy evaluation of BLISS.

5.1 Code Size

The top graph of Figure 4 presents the compression ratio achieved for the different BLISS executables compared to the MIPS-32 code size. Compression ratio is defined as the percentage of the compressed code size over the original code size with 32-bit instructions. Lower ratio means smaller code size.

Direct translation with basic-optimizations (*Basic-Optimizations* bar) of MIPS-32 code leads to an increase in code size with a 106% average compression ratio. Block subsetting (*Block-Subset* bar) yields an average compression ratio of 77%. Mixing 16- and 32-bit instruction sets makes the BLISS executables 29% less than the MIPS-32 code (71% compression ratio). Combining the two optimizations leads to 60% compression ratio. Note that when the two optimizations are enabled, the individual reductions in code size do not add up. This is due to two reasons. First, block subsetting is only performed within the blocks of the same instruction size: 16-bit instruction blocks can only be considered for block subsetting with other 16-bit blocks and the same applies to 32-bit blocks. Hence, the opportunity for removing repeated sequences is less. Second, the saving from eliminating 16-bit instructions is half the saving from eliminating 32-bit instructions. Table 2 presents additional detailed statistics on the code size optimizations studied. Extra instructions are required when interleaving 16 and 32-bit blocks due to register spilling as Instructions in 16-bit format can only access 8 registers.

5.2 Performance Analysis

The middle graph of Figure 4 compares the percentage of IPC improvement achieved using the different BLISS executables for the XScale processor configurations over the base design. The original BLISS with basic optimizations provides an 11% average IPC improvement over the base design. BLISS provides very similar IPC improvements even with block subsetting. The additional I-cache misses due to reduced instruction locality are well tolerated through prefetching using the contents of the BBQ. The elimination of repeated instruction sequences allows for more unique instructions to fit in the I-cache at any point in time. Hence, certain applications observe an overall higher I-cache hit rate.

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

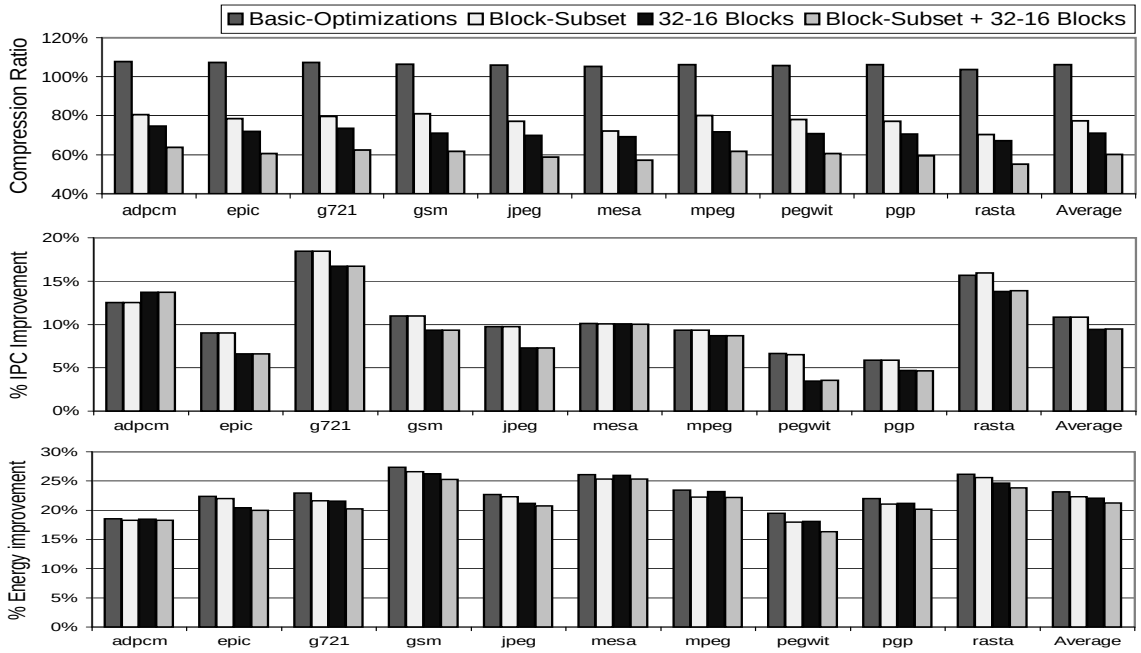


Figure 4: Evaluation of the XScale processor configuration for the different BLISS binaries over the base design.

register spilling and long offsets for load and store instructions. On average, 1.4% more instructions are executed compared to the original code. The net effect is a small degradation in average performance compared to the original BLISS. Nevertheless, for benchmarks like mesa which stresses the I-cache capacity, the net effect is a small performance improvement. Using block subsetting in addition to interleaved 16-bit/32-bit instructions results in a similar performance as the one observed when the optimization is enabled on the original BLISS code.

5.3 Energy Analysis

The bottom graph of Figure 4 compares the percentage of total energy improvement achieved using the different BLISS executables over the base design. BLISS offers a 23% total energy advantage over the base design. The BLISS advantage is due a number of factors: reduced energy spent on mispredicted instructions, selective word access in the I-cache, merging of I-cache accesses for sequential blocks, and judicious access to the branch predictor. 16-bit encodings introduce some energy consumption due to the additional instructions. Nevertheless, BLISS with mixed instruction widths and subsetting provides a 21% total energy advantage over the base design.

5.4 Comparison to Selective Use of 16-bit Code

Figure 5 compares BLISS with block subsetting and selective use of 16-bit blocks to selective use of 16-bit instructions with a conventional ISA like Thumb-2 and rISA (*Sel16*) [17, 6, 11]. Note that the same profitability heuristic

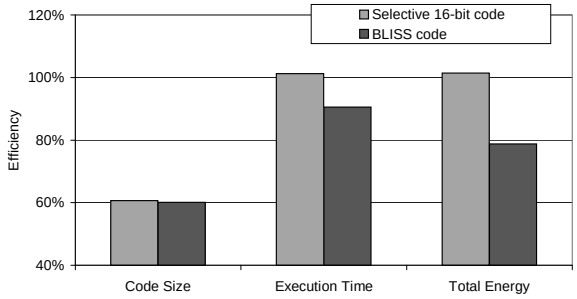


Figure 5: Average Code size, execution time, and total energy consumption for selective 16-bit and BLISS (with block-subset and 32/16 blocks) executables for the XScale processor configuration over the base. Lower bars present better results.

is used with both ISAs to select which instructions or blocks to encode with 16 bits. The base XScale configuration with the full-sized I-cache is used for Sel16.

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

between 16-bit and 32-bit code. Overall, BLISS improves upon Sel16 by offering similar code density at superior performance and energy consumption.

6 Related Work

Many code size reduction techniques have been proposed and widely used in embedded systems [2]. Most techniques store the compressed program code in memory and decompression happens on I-cache misses [20, 15, 9] or inside the processor [14]. Compression is typically dictionary based. Such techniques reduce memory footprint and the off-chip bandwidth requirements for instruction accesses. When decompression occurs in the core, additional latency is introduced for instruction execution. When decompression occurs on cache refills, additional pressure is placed on the I-cache capacity. BLISS reduces code size, places no additional pressure on I-cache capacity, and improves on execution time. BLISS can be combined with a dictionary compression scheme behind the I-cache for further code size improvements.

Cooper proposed a compiler framework for discovering and eliminating repeated instruction sequences [5]. The echo instruction has been proposed to facilitate elimination of such redundancies [12]. An echo instruction is used in the place of repeated sequences and points back to the unique location for that code. Using echo instructions, 84% compression ratio is reported in [12]. BLISS facilitates redundancy elimination with block subsetting, which on its own leads to a 77% compression ratio. Moreover, BLISS allows for significant performance improvements in addition to code compression, which is not the case with previous proposals.

The BLISS instruction set builds on the block-structured instruction set that defines block boundaries within the regular instruction stream [16]. Decoupled processor frontends for ordinary ISAs have been proposed and analyzed in [18]. The advantages that BLISS introduces over this work are analyzed in [21, 22].

7 Conclusions

This paper evaluated the use of a block-aware instruction set (BLISS) to achieve code size, performance, and energy improvements for embedded processors. The BLISS ISA defines basic block descriptors in addition to and separately from the actual instructions. It enables code size optimizations by removing redundant sequences of instructions across basic blocks and by allowing a fine-grain interleaving of 16-bit and 32-bit instructions without overhead instructions. We showed that BLISS allows for 40% code size reduction over a conventional RISC ISA and simultaneously achieves 10% performance and 21% total energy improvements. Hence, BLISS improves concurrently the performance and cost of embedded systems.

References

[1] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A Transparent Dynamic Optimization System. In *Conference on Programming Language Design and Implementation*, 2000.

[2] A. Beszedes et al. Survey of Code-Size Reduction Methods. *ACM Comput. Surv.*, 35(3), 2003.

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

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

[5] K. Cooper and N. McIntosh. Enhanced Code Compression for Embedded RISC Processors. In *Conference on Programming Language Design and Implementation*, 1999.

[6] A. Halambi et al. An Efficient Compiler Technique for Code Size Reduction Using Reduced Bit-Width ISAs. In *Conference on Design, automation and test in Europe*, 2002.

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

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

[9] M. B. Jr. and R. Smith. Enhanced Compression Techniques to Simplify Program Decompression and Execution. In *Intl. Conference on Computer Design*, 1997.

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

[11] A. Krishnaswamy and R. Gupta. Profile Guided Selection of ARM and Thumb Instructions. In *Joint Conference on Languages, Compilers and Tools for Embedded Systems*, 2002.

[12] J. Lau et al. Reducing Code Size With Echo Instructions. In *Intl. Conference on Compilers, Architecture and Synthesis for Embedded Systems*, 2003.

[13] C. Lee, M. Potkonjak, and W. Mangione-Smith. MediaBench: a Tool for Evaluating and Synthesizing Multimedia and Communications systems. In *Intl. Symposium on Microarchitecture*, 1997.

[14] C. Lefurgy, P. Bird, I.-C. Chen, and T. Mudge. Improving Code Density Using Compression Techniques. In *Intl. symposium on Microarchitecture*, 1997.

[15] H. Lekatsas, J. Henkal, and W. Wolf. Code Compression for Low Power Embedded System Design. In *Conference on Design Automation*, 2000.

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

[17] R. Phelan. Improving ARM Code Density and Performance. Technical report, Advanced RISC Machines Ltd, 2003.

[18] G. Reinman, T. Austin, and C. Calder. A Scalable Front-End Architecture for Fast Instruction Delivery. In *Intl. Symposium on Computer Architecture*, 1999.

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

[20] A. Wolfe and A. Chanin. Executing Compressed Programs on An Embedded RISC Architecture. In *Intl. Symposium on Microarchitecture*, 1992.

[21] A. Zmily, E. Killian, and C. Kozyrakis. Improving Instruction Delivery with a Block-Aware ISA. In *EuroPar Conference*, 2005.

[22] A. Zmily and C. Kozyrakis. Energy-Efficient and High-Performance Instruction Fetch using a BlockAware ISA. In *Intl. Symposium on Low Power Electronics and Design*, 2005.