Due to the type 3 fonts used, please increase the magnification to view

Cached-Code Compression for Energy Minimization in Embedded Processors

Luca Benini
Università di Bologna
Bologna, ITALY 40136
lbenini@deis.unibo.it

Alberto Macii
Politecnico di Torino
Torino, ITALY 10129
amacii@athena.polito.it

Alberto Nannarelli
Università di Roma Tor Vergata
Roma, ITALY 00133
nannarelli@ing.uniroma2.it

ABSTRACT

This paper contributes a novel approach for reducing static code size and instruction fetch energy for cache-based core processors running embedded applications. Our implementation of the decompression unit guarantees fast and low-energy, on-the-fly instruction decompression at each cache lookup. The decompressor is placed outside the core boundaries; therefore, processor architecture does not need any modification, making the proposed compression approach suitable to IP-based designs. Viability of our solution is assessed through extensive benchmarking performed on a number of typical embedded programs.

1. INTRODUCTION

Embedded processors are often the main computational engines for modern system-on-chip (SoC) architectures. Processors for embedded applications have traditionally been extremely simple (8-bit or 16-bit CPUs), because of tight cost constraints coupled with loose performance demand. The increasing level of integration and computational speed requirements, fueled by the new generation of embedded computing tasks (e.g., DSP, high-bandwidth data transfer, etc.) have changed the picture. Currently, many embedded processors are based on high-performance RISC architectures, with on-chip cache and full support for complex memory systems and peripheral controllers. Such processors, and their software development environments, are usually purchased by system integrators from third-party companies that specialize in embedded core design.

One of the key challenges in designing a complex system around a high-performance embedded RISC processor is to ensure sufficient instruction fetch bandwidth to keep the execution pipeline busy. The regularity of RISC instruction sets eases application and compiler development, but hinders code compaction. For this reason, designers and researchers have put significant effort in devising techniques for improving code density and reducing instruction-related costs, in terms of speed, area and energy.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISLPED'01, August 6-7, 2001, Huntington Beach, California, USA.
Copyright 2001 ACM 1-58113-371-5/01/0008 ...$5.00.

Numerous code compression techniques have been proposed for reducing instruction memory size in low-cost embedded applications [1]. The basic idea is to store programs in compressed form and decompress them on-the-fly at execution time. Later, researchers have realized that code compression can be beneficial for energy as well, because it reduces the energy consumed in reading instructions from memory and communicating them to the processor core [2, 3, 4]. Code compression leverages well-known lossless data compression techniques, but it is characterized by two constraints. First, it must be possible to decompress a program in relatively small blocks, as instructions are fetched, and starting from several points inside the program (i.e., branch destinations). Hence, techniques that decompress a stream starting from a single initial point are not applicable without changes. Second, the decompressor should be small, fast and energy efficient, because its hardware cost must be fully amortized by the corresponding savings in memory size and energy, without compromising performance.

For simple processors with no instruction cache, the decompressor is either merged with the processor core itself, or placed between program memory and processor. The first solution has been implemented in several commercial core processors, in form of a "dense" instruction set, with short instructions (e.g., ARM Thumb and MIPS16 instruction sets). The second solution has been investigated in several papers [2, 5, 6, 3]. Supporting restricted instruction sets requires changes to the core architecture, while an external decompressor does not. Furthermore, with an external decompressor it is possible to aggressively tailor code compression to a specific embedded application. Hence, external decompression is well-suited for embedded designs employing third-party cores, which are the focus of this paper.

In more advanced architectures containing an instruction cache, the decompressor can be placed either between the I-cache and the main memory (*decompress on cache refill*, or DCR architecture), or it can be placed between the processor and the I-cache (*decompress on fetch*, or DF architecture). Both alternatives have been investigated in the recent literature [4, 7]. In [4], it was shown that from an energy and performance viewpoint, the DF architecture is superior to the DCR architecture, when decompressor overhead is small. The main reason for this effect is that instructions are stored in cache in a compressed fashion, effectively increasing cache capacity. However, current silicon implementations of code compression are based on the DCR architecture, indicating that reducing decoding overhead is a non-trivial task that still entails significant challenges.

The main issue with the DF approach is that decompression is performed on every instruction fetch. In other words, the decompressor is on the critical path for the execution of every instruction, not only for cache refills. If its delay is not small, it may significantly slow down execution. Furthermore, it consumes energy on every instruction fetch, while in the DCR architecture it can be activated only on cache refills. Careful implementation of the decompression unit is thus key for making DF applicable in practice.

This paper proposes a novel DF architecture that focuses on reducing decoding overhead on energy and performance. First, our technique *guarantees* that storage requirements for the compressed program *always* decrease. Second, the compression algorithm has been designed specifically for fast and low-energy decoding during cache lookup. Compressed instructions are always aligned to cache line boundaries, branch destinations are word-aligned and instruction decompression is based on a single lookup into a small (and fast) memory buffer. Third, we do not limit our analysis to the architecture level, but present a complete implementation of the cache-decompressor block, including detailed analysis of its energy and delay.

We have benchmarked the proposed compression technique on a number of programs implementing functions typical of embedded computations. The achieved code size reductions, averaged over all the experiments, are around 28%. From the energy point of view the improvements vary a lot depending on cache size, original and compressed code size, dynamic memory access profile, and kind of adopted program memory (i.e., on-chip vs. off-chip). For example, for a 4Kbyte cache and an on-chip program memory, average energy savings are around 30%. This value grows to 50% for a system with a cache of the same size but an off-chip program memory. Cache performance are also very sensitive to cache and code size. For a 4Kbyte cache, the average hit ratio improves by 19%.

2. DECOMPRESSION ON FETCH

2.1 Previous Work

We begin this section by outlining the characteristics of two previously published DF approaches, by Larin *et al.* [7], and by Lekatsas *et al.* [4]. Larin's approach targets VLIW processors and compresses instructions using the Huffman algorithm. Basic blocks of compressed instructions are transferred and stored into the I-cache atomically. Compressed instructions are not aligned to cache line boundaries. On a cache access, two consecutive cache lines are decompressed and stored in a level-zero buffer. The following instructions are fetched in sequence from the buffer, until it is emptied, or a branch is executed. The paper by Larin does not report any data on energy, speed or area of the decoder, but its high hardware cost is apparent. Fetching and decoding two cache lines at a time imposes parallel Huffman decoding of multiple instructions in one clock cycle (even single-instruction Huffman decoding requires a quite large hardware block). Furthermore, branch targets addresses in compressed code are stored in a dedicated address re-mapping memory that must be accessed on every taken branch.

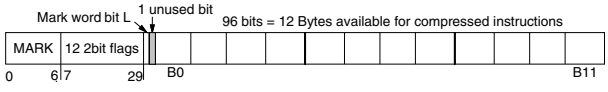
Lekatsas approach clusters instructions in four groups (*instructions with immediates*, *branches*, *fast dictionary* instructions and *uncompressed* instructions) identified for decoding purpose by a unique prefix. Instructions with immediate are compressed using arithmetic coding. For branches, only destination displacement is variable-length coded. Fast dictionary instructions, with no immediate are compressed to a one-byte code and decompressed using a 256-entries lookup memory. Uncompressed instructions are left intact, and extended with a 3-bit preamble. Even though Lekatsas reports on decoding energy consumption, no information is provided in [4] on decoder area, speed and its interaction with cache (for instance, cache line unpacking, and branch destination re-mapping in the case of non word-aligned branch destinations, are not described). Furthermore, appending a 3-bit preamble to uncompressed instructions may cause run-time decompression inefficiency, if many uncompressed instructions are fetched during program execution.

2.2 A Low-Overhead DF Architecture

The compression algorithm described in the sequel is tailored for efficient hardware implementation. The decompressor is merged with the cache controller, and instructions are decompressed on the fly as they are extracted from the cache. Instructions are compressed in groups with the size of one cache line. In describing the algorithm, we will assume a direct-mapped cache with 4-word lines ($L = 128$ bits) and one-word ($W = 32$ bits) uncompressed instructions. However, our approach can be extended to caches with any associativity and line size. We will use the DLX as target core processor, because it has a simple 32-bit RISC architecture with several open-source synthesizable implementations, and an open-source, widely known software development environment. Moreover, the DLX is similar to several commercial RISC processors, such as the simplest cores in the ARM and MIPS families.

Compression is targeted to a specific program thanks to execution profiling. The code is initially profiled and the subset $\mathcal{S}_N$ of the N most frequently executed instructions is obtained. Without loss of generality, we assume in the following $N = 256$. Similarly to [3] and [4] (fast dictionary instruction), $log_2 N$-long compressed codes (8 bits, in our case) are assigned to the N most frequent instructions. However, the decision of compressing an instruction i, even if it belongs to set $\mathcal{S}_N$ *depends on the neighboring instructions* that would fit in the same cache line. More specifically, instruction i is compressed only if it belongs to a group of instructions that can be stored in a *compressed line*. The latter is a group of more than four adjacent instructions which, after compression, will be stored in four consecutive words. The size (four words) and the memory alignment of a compressed line is such that it fits in a single cache line, when it is cached. The detailed structure of a compressed line is shown in Figure 1.

Figure 1: Compressed Line Structure.



The first word of the line contains a *mark* and a set of *flag bits*. The mark is an unused instruction opcode (in DLX, we have 6-bit opcodes), while the flag bits are divided in 12 groups of 2 bits each, one group for each of the bytes of the remaining three words of the line. One additional flag bit `L` is reserved at the end of the flags.

The flags values are assigned as follows: 00 if the corresponding byte contains a compressed instruction; 01 if the corresponding byte contains 8 bits of an uncompressed instruction (a compressed line may contain an uncompressed instruction); 11 if the corresponding byte is left empty for alignment reasons (see below). Flag value 10 is used to signal the last compressed instruction in the line, in case of lines containing compressed instructions. Finally, the last flag bit `L` marks if the last instruction in the line is compressed (bit set to 0) or not (bit set to 1).

By construction, a compressed line stores between a minimum of 5 instructions, and a maximum of 12 instructions. Thus, the best compression achievable is a factor of 4.

The compression algorithm analyzes the code sequentially, starting from the first instruction (assuming that it is aligned to cache line boundary), and tries to pack adjacent instructions in compressed lines. If the compressed line contains more than 4 words, all instructions belonging to $\mathcal{S}_{\mathcal{N}}$ are compressed to their $log_2 N$ codes, the mark and flags are inserted in the first word of the compressed line and empty bytes are inserted when required for alignment reasons. Otherwise, instructions are not compressed even if they belong to $\mathcal{S}_{\mathcal{N}}$.

The compression procedure guarantees that compressed code is never larger than the original code. Furthermore, the number of bits transferred from memory to cache when executing compressed code is never larger than for the uncompressed case.

To facilitate an efficient hardware implementation of the decoder, we guarantee two types of alignment:

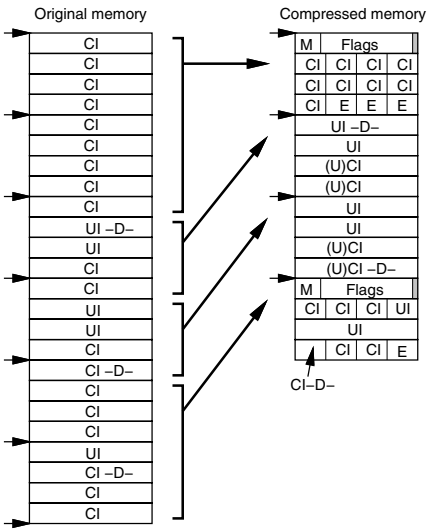
- No instruction is allowed to go across a cache line boundary. This rule ensures that only a single cache line needs to be accessed at every instruction fetch, thus avoiding expensive double-line accesses, which require dual-bank caches [7].

- Instructions at branch destinations are always word-aligned. This rule allows us to eliminate the address translation table, since branch addresses in original code can be replaced with legal, word-aligned branch destination addresses in the compressed code.

Alignment constraints imply some losses in code density, because a few bytes in compressed lines may be left empty. However, this effect is taken into account when deciding whether a group of instructions can be compressed or not.

EXAMPLE 1. *Figure 2 shows code compression at work. In the original memory map, instructions are marked with* `CI` *if they belong to* $\mathcal{S}_{\mathcal{M}}$ *(i.e., candidate for compression), with* `UI` *if they do not. Tag* `-D-` *is placed on instructions which are at a branch target. The arrows on the left of the memory maps represent cache line alignment (i.e., the words between two arrows are cached on the same cache line). The compression algorithm moves from the top of the original memory map, and tries to pack as many instructions as possible in a compressed line. The first group of 9* `CI` *instructions fits, after compression, into a compressed line. The following instruction is a* `UI` *(and a branch destination), which does not fit in the remaining two bytes of the compressed line. Hence, the last three bytes are left empty. The flags for the first compressed line are:* `00.00.00.00.00.00.00.00.10.11.11.11.1`. *The algorithm then tries to fit the following instructions into a compressed line, but it cannot, because after two consecutive UIs, compression would be convenient only if we had at least three consecutive CIs, but we have only two. Hence, the* `UI-D` *and the following three instructions are stored uncompressed in memory. Notice that this choice implies that two* `CI`*s are not compressed. In compressed memory, this case is marked as* `(U)CI`. *A similar decision is taken for the next four instructions, which are, again, stored in uncompressed form. Finally, the last 7 instructions are stored in a compressed line. Notice that: (i) The compressed line contains an uncompressed instruction; (ii) The first empty byte is due to the branch alignment constraint (branch destinations must be word-aligned). Flags for the second compressed line are:* `00.00.00.01.01.01.01.11.00.00.10.11.1`. *For this example, compression reduced code size from 24 words to 16.*

Figure 2: Code Compression Example.



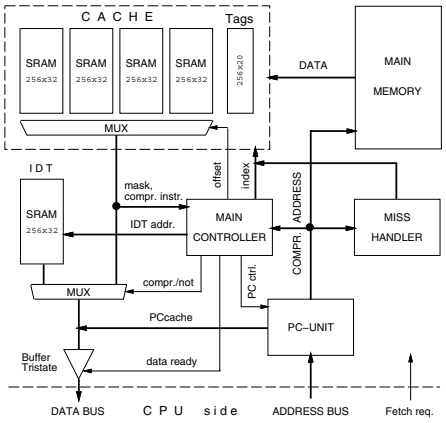
During program execution, on every cache miss, a new cache line is fetched from memory. The line may either contain four uncompressed instruction, or a first word containing mark and flags, followed by three words containing five or more compressed instructions. The cache controller examines the first word of the line. If a mark is detected, it proceeds to fetch instructions according to the indications provided by the flag bits. Decompression is performed by addressing the decompression table (a fast RAM containing 256 32-bit words) with the 8-bit compressed instruction code. A detailed description of the hardware decompression engine is provided in the next section. More information on the implementation of the compression algorithm and the related profiling and instruction selection flow is provided in Section 4.

3. DECOMPRESSION HARDWARE

The hardware which performs the on-the-fly decompression is embedded in the cache controller. The cache controller is divided into a number of units: A master controller, called "main controller" in the sequel, which handles the communication with the CPU, a slave controller, called "miss handler" in what follows, which supervises the transfers from main memory to the cache when a read miss occurs, and a program counter update unit (PC-unit).

The decompression-cache system, sketched in Figure 3, is completed by the following units: The cache array of 256 × 4 words, plus tags of 20 bits. A MUX placed at the output of the cache array. A memory holding the compressed instructions (IDT). A MUX between the cache and the data bus, driven by the main controller, which selects either the output of the cache array (for uncompressed instructions) or the output of the IDT (for compressed instructions).

Figure 3: Decompression Unit.



When a memory read access is issued by the CPU, the address (index) is passed to the cache and, when the datum is ready, the main controller performs tag checking. On a tag mismatch (i.e., a miss), the main controller enables the miss handler, puts its command and address buses toward the cache in high-impedance and sets itself in idle state.

On a read miss, the miss handler is woken up by the main controller and the requested block (4 words) is transfered to the cache. Upon completion, the miss handler signals to the main controller the end of the transfer, puts its command and address buses toward the cache in high-impedance and sets itself in idle state. In case of a cache hit, the main controller executes the following tasks:

1. It reads the first word of the block (or cache line) to check if such block contains at least a compressed instruction. This is done by matching the first 6 bits (opcode) with the pattern indicating a compressed line.

2. According to the result of the matching, it selects the appropriate instruction by setting the control lines of cache and data bus MUXes.

If the cache line contains at least one compressed instruction, the main controller performs some additional work: *i)* It stores the flag bits of the mark in a mask register on the first visit to the cache line. *ii)* It checks the flag bits corresponding to the requested instruction. If it is compressed, the main controller issues the IDT address to be accessed; otherwise, it selects the appropriate word by operating the cache MUX. Finally, it executes task 2 above.

To keep track of the instruction to be fetched, in case of compression, the main controller updates a 4-bit counter. Every time a new cache line is requested or the current instruction is the target of a branch, the main controller resets the counter to the appropriate value, and then it increments it by 1, if the current instruction is compressed, or by 4, if the current instruction is not compressed.

The CPU does not know if the instruction under execution is compressed or not. Therefore, the Program Counter (PC) is incremented by 4 for each instruction, assuming sequential code. However, compressed instructions in the cache occupy 1 byte instead of 4, and this causes a problem of misalignment between the PC and the actual address of the instruction to fetch, or in other words, the program counter tends to *"run forward"*.

To solve this problem, which affects relative jumps, we can opt for two solutions:

- If the CPU core is modifiable, we can stop the PC increment when reading inside the same cache word multiple compressed instructions. This can be accomplished with modest hardware additions.

- If the CPU core is sealed-off, we have to replicate the PC logic inside the decompressor-cache. Realignment of CPU PC and cache PC is done when a jump to a register is executed (*"jr register"* in the DLX instruction set).

The second solution is obviously more expensive in terms of hardware requirements: A 32 bit adder plus two 32 bit registers (one to hold the PC, one to hold the possible target of a jump) have to be placed in the decompressor (PC-unit). Moreover, an additional problem arises in case of subroutine calls. The DLX instruction *"jal address"* performs two tasks upon execution:

1. The PC (return address) is stored in register 31.

2. The CPU jumps to *"address"*.

To get the correct return address from the subroutine, we store in register 31 the cache PC instead of the CPU PC. This is done by changing instruction *"jal address"* (during compression) with two instructions:

"lw r31, (don't care)"
"j address"

and by sending the value of the cache PC on the data bus at the right time. In this way, when the subroutine ends (instruction *"jr $31"*), the correct address of the compressed program is restored and the CPU PC is realigned.

The decompressor introduces an overhead on the timinig as well. In normal cache operation, the time required to fetch the instruction, in case of hit, is:

$$t_{fetch-wo} = t_{cache} + t_{match} + t_{busEN}$$

where t_{cache} is the time necessary to access the cache and transfer the data through the cache MUX, t_{match} is the time needed for tag-matching, and t_{busEN} is the time to enable the tristate buffer if a hit occurred. In the decompressor, the time required to fetch a compressed instruction (worst case) is:

$$t_{fetch-dec} = t_{cache} + t_{gen} + t_{IDT} + t_{mux2:1} + t_{busIN}$$

where t_{gen} is the time for IDT address generation, t_{IDT} is the IDT access time, $t_{mux2:1}$ is the in-to-out delay of the 2:1 MUX, and t_{busIN} is the transfer time through the tristate buffer.

We used a $0.25\mu m$, $2.5V$ library of standard cells from ST to implement our decompressor and found the values for the timing and energy per cycle reported in Tables 1 and 2. Synthesis was performed using Synopsys Design Compiler, and energy estimation using Synopsys Power Compiler.

Table 1: Delays of Blocks on the Critical Path.

Delay	W/O Dec. [ns]	With Dec. [ns]
t_{cache}	2.2	2.2
t_{match}	0.6	
t_{gen}		2.0
t_{IDT}		1.5
$t_{mux2:1}$		0.2
t_{busEN}	1.9	
t_{busIN}		0.3
$t_{fetch-wo}$	4.7	
$t_{fetch-dec}$		6.2

Table 2: Energy per Cycle of Blocks in Figure 3.

Block	W/O Dec. [J]	With Dec. [J]
Main Controller	9.1e-12	3.1e-11
Miss Handler	2.1e-11	2.1e-11
PC Unit		4.2e-11
Cache Mux	1.1e-11	1.8e-11
Sparse Logic	0.4e-11	2.6e-11
IDT (256×32)		7.0e-10
Cache	3.2e-9	3.2e-9

4. EXPERIMENTAL RESULTS

In this section, we report data on the use of the proposed compression scheme. Software programs we considered for the experiments are some of the C benchmarks distributed in the `Ptolemy` [8] package; they implement functions that are widely exploited in embedded systems for DSP. Data collection has been done using the SuperDLX [9] compilation and simulation environment.

We have used, as program memory, a 512Kbyte SRAM from ST, organized in eight banks of $32K \times 16$ and built in $0.25\mu m$ technology at 2.5V. The energy access cost, in read mode, is 17.2 nJ per memory block of four, 32-bit words. The line capacitance we used for determining the address and data bus energy is $0.6pF$.

Table 3 summarizes all the results for the case of a 4Kbyte cache. In particular, for each benchmark program and for both original and compressed execution, it reports static code size (column *Size*), cache hit ratio (column *HR*) and total energy consumption (column *Energy*). The total number of executed instructions is provided on the left of the table (column *Exec. Instr.*), while the percentage of variation introduced by the compression on code size, hit ratio and energy is summarized in column Δ.

Average code size reduction is around 28%, with a peak value around 61% for program `chaos`. Cache performance improve, on average, by 19%, with a maximum increase in the hit ratio of 34% for benchmark `integrator`. Finally, energy decreases, on average, by 30%, with peak reductions for programs `chaos` and `interp` around 48% and 53%, respectively.

Energy figures are further detailed in Table 4, where a breakdown of all the contributors to the total values (i.e., cache including the decompressor, address and data buses, background memory) is shown. Clearly, the energy consumed in the cache system increases substantially w.r.t. the case of uncompressed code, since it includes the energy for instruction decompression which is required for each fetched instruction (coming from both cache and memory). Obviously, decrease in bus and memory energy well compensates the overhead of the decompressor.

The results of Tables 3 and 4 refer to a system with a 4Kbyte cache. In Tables 5 and 6 we explore how cache performance and energy consumption change as the cache size changes (we consider the case of 8Kbyte and 2Kbyte caches). We observe some contradictory effects that yield results which are not always intuitive. For example, a larger cache increases the hit ratio, thus limiting the number of times the background memory is accessed also for the case of uncompressed programs. This causes a substantial decrease in bus and memory energy for both uncompressed and compressed execution. In addition, cache access cost is higher and the decompression overhead is no longer compensated due to the lower absolute contribution of bus and main memory to total energy. As a consequence, the usefulness of compressing the code decreases; average energy savings are, in fact, around 8%, and there are programs for which energy even increases. For a smaller cache, the hit ratio tends to decrease. This causes a strong increase in bus and memory energy, especially for the uncompressed code. In addition, cache access cost decreases. As a result, energy reductions are more sizable. We can then conclude that the relationship between energy, compression ratio and cache hit ratio is very complex; as such, intensive and exhaustive code profiling and dynamic simulation are the basic ingredients for achieving a satisfactory trade-off.

We close with some results for the case where accesses to memory go off chip. The memory is a 4Mbit FLASH at 2.5V from ST; the energy cost for accessing a 4-word location is 21.4 nJ. The bus load we have assumed is $8pF$ per line. Tables 7 and 8 show data similar to those in Tables 3 and 4. Although compression ratios and hit ratios remained unchanged, larger energy savings are observed (i.e., the average is around 50%) due to the much more penalizing effect introduced by off-chip bus and memory accesses.

5. CONCLUSIONS

We have proposed a new approach for reducing static code size and instruction fetch energy for embedded processors with cache. The method pairs efficiency in the compression with a low-overhead implementation of the decompressor. Experiments on several benchmarks have shown average code compression results around 28% and average energy savings of about 30%.

6. REFERENCES

[1] C. Lefurgy, *Efficient Execution of Compressed Programs*, Doctoral Dissertation, University of Michigan, 2000.

[2] Y. Yoshida, B.-Y. Song, H. Okuhata, T. Onoye, I. Shirakawa, "An Object Code Compression Approach to Embedded Processors," *ISLPED-97*, pp. 265-268, 1997.

[3] L. Benini, A. Macii, E. Macii, M. Poncino, "Selective Instruction Compression for Memory Energy Reduction in Embedded Systems," *ISLPED-99*, pp. 206-211, 1999.

[4] H. Lekatsas, J. Henkel, W. Wolf, "Code Compression for Low Power Embedded Systems Design," *DAC-00*, pp. 294-299, 2000.

[5] S. Y. Liao, S. Devadas, K. Keutzer, "Code Density Optimization for Embedded DSP Processors Using Data Compression Techniques," *IEEE Trans. on CAD*, Vol. 17, No. 7, pp. 601-608, 1998.

[6] H. Lekatsas, W. Wolf, "Code Compression for Embedded Systems," *DAC-00*, pp. 516-521, 1998.

[7] S. Larin, T. Conte, "Compiler-Driven Cached Code Compression Schemes for Embedded ILP Processors," *MICRO-32*, 1999.

[8] J. Davis II *et al.*, *Overview of the Ptolemy Project*, UCB/ERL Tech. Report No. M99/37, UC Berkeley, 1999.

[9] C. Moura, *SuperDLX: A Generic Superscalar Simulator*, ACAPS Technical Memo 64, McGill Univ., 1993.

Table 3: Compression Results: 4Kbyte Cache and On-Chip SRAM.

Bench	*Exec. Instr.* [#]	*Without Compression*			*With Compression*			Δ		
		Size [Kbyte]	*HR* [%]	*Energy* [J]	*Size* [Kbyte]	*HR* [%]	*Energy* [J]	*Size* [%]	*HR* [%]	*Energy* [%]
adaptFilter	5.32e5	4530	90.79	2.93e-5	3784	98.68	2.38e-5	-16.47	8.69	-18.77
butterfly	2.50e5	8048	79.20	2.09e-5	4292	91.60	1.56e-5	-46.67	15.66	-25.25
chaos	4.91e5	9266	77.80	4.27e-5	3660	98.57	2.21e-5	-60.50	26.70	-48.20
dft	8.81e6	11212	81.05	6.97e-4	9742	90.69	5.64e-4	-13.11	11.90	-19.01
iirDemo	2.31e5	10168	70.56	2.44e-5	8112	90.48	1.49e-5	-20.22	28.22	-38.85
integrator	4.02e5	6224	87.06	2.60e-5	4108	96.27	2.04e-5	-33.99	10.57	-21.44
interp	9.35e5	8312	71.87	9.48e-5	6544	97.43	4.45e-5	-21.27	35.56	-53.02
scramble	4.23e6	13296	77.78	3.68e-4	11828	87.71	3.02e-4	-11.04	12.77	-17.94
Average			79.51			93.93		-27.90	18.76	-30.31

Table 4: Energy Break-Down: System with 4Kbyte Cache and On-Chip SRAM.

Bench	*Energy Without Compression* [J]			*Energy With Compression* [J]			Δ [%]		
	Cache	*Bus*	*SRAM*	*Cache*	*Bus*	*SRAM*	*Cache*	*Bus*	*SRAM*
adaptFilter	1.59e-5	3.78e-6	9.60e-6	2.17e-5	5.69e-7	1.51e-6	36.45	-84.94	-84.29
butterfly	6.52e-6	4.06e-6	1.03e-5	9.47e-6	1.68e-6	4.46e-6	45.19	-58.46	-58.46
chaos	1.26e-5	8.50e-6	2.16e-5	2.00e-5	5.76e-7	1.53e-6	59.05	-93.23	-92.94
dft	2.35e-4	1.30e-4	3.31e-4	3.30e-4	6.40e-5	1.70e-4	40.48	-50.84	-48.75
iirDemo	5.37e-6	5.37e-6	1.36e-5	8.64e-6	1.72e-6	4.55e-6	60.96	-68.02	-66.66
integrator	1.15e-5	4.07e-6	1.03e-5	1.60e-5	1.20e-6	3.18e-6	38.80	-70.49	-69.25
interp	2.21e-5	2.05e-5	5.21e-5	3.77e-5	1.88e-6	4.98e-6	70.18	-90.84	-90.45
scramble	1.08e-4	7.33e-5	1.86e-4	1.53e-4	4.07e-5	1.08e-4	41.56	-44.47	-42.11
Average							49.08	-70.16	-69.11

Table 5: Compression Results: 8Kbyte Cache and On-Chip SRAM.

Bench	*Exec. Instr.* [#]	*Without Compression*			*With Compression*			Δ		
		Size [Kbyte]	*HR* [%]	*Energy* [J]	*Size* [Kbyte]	*HR* [%]	*Energy* [J]	*Size* [%]	*HR* [%]	*Energy* [%]
adaptFilter	5.32e5	4530	98.87	1.97e-5	3784	99.06	2.40e-5	-16.47	0.19	21.32
butterfly	2.50e5	8048	96.40	1.09e-5	4292	97.60	1.23e-5	-46.67	1.24	13.28
chaos	4.91e5	9266	86.76	3.27e-5	3660	99.39	2.18e-5	-60.50	14.55	-33.34
dft	8.81e6	11212	87.63	5.67e-4	9742	96.25	4.57e-4	-13.11	9.84	-19.46
iirDemo	2.31e5	10168	84.41	1.67e-5	8112	96.97	1.16e-5	-20.22	14.87	-30.57
integrator	4.02e5	6224	97.26	1.66e-5	4108	98.51	1.88e-5	-33.99	1.28	12.76
interp	9.35e5	8312	90.37	5.40e-5	6544	99.25	4.17e-5	-21.27	9.82	-22.70
scramble	4.23e6	13296	86.99	2.81e-4	11828	92.67	2.57e-4	-11.04	6.52	-8.34
Average			91.09			97.46		-27.90	7.29	-8.38

Table 6: Compression Results: 2Kbyte Cache and On-Chip SRAM.

Bench	*Exec. Instr.* [#]	*Without Compression*			*With Compression*			Δ		
		Size [Kbyte]	*HR* [%]	*Energy* [J]	*Size* [Kbyte]	*HR* [%]	*Energy* [J]	*Size* [%]	*HR* [%]	*Energy* [%]
adaptFilter	5.32e5	4530	60.34	6.85e-5	3784	91.73	3.24e-5	-16.47	52.02	-52.68
butterfly	2.50e5	8048	45.20	4.14e-5	4292	64.40	3.20e-5	-46.67	42.48	-22.79
chaos	4.91e5	9266	48.47	7.74e-5	3660	82.07	4.15e-5	-60.50	69.33	-46.38
dft	8.81e6	11212	51.42	1.33e-3	9742	65.95	3.44e-4	-13.11	28.26	-74.02
iirDemo	2.31e5	10168	51.95	3.45e-5	8112	69.26	2.67e-5	-20.22	33.33	-22.43
integrator	4.02e5	6224	73.63	3.88e-5	4108	87.31	2.90e-5	-33.99	18.58	-25.29
interp	9.35e5	8312	58.18	1.25e-4	6544	92.83	5.44e-5	-21.27	59.56	-56.55
scramble	4.23e6	13296	67.85	4.67e-4	11828	80.14	3.78e-4	-11.04	18.12	-19.09
Average			57.13			79.21		-27.90	40.21	-39.90

Table 7: Compression Results: 4Kbyte Cache and Off-Chip FLASH.

Bench	*Exec. Instr.* [#]	*Without Compression*			*With Compression*			Δ		
		Size [Kbyte]	*HR* [%]	*Energy* [J]	*Size* [Kbyte]	*HR* [%]	*Energy* [J]	*Size* [%]	*HR* [%]	*Energy* [%]
adaptFilter	5.32e5	4530	90.79	7.79e-5	3784	98.68	3.11e-5	-16.47	8.69	-18.77
butterfly	2.50e5	8048	79.20	7.31e-5	4292	91.60	3.73e-5	-46.67	15.66	-25.25
chaos	4.91e5	9266	77.80	1.52e-4	3660	98.57	2.95e-5	-60.50	26.70	-80.60
dft	8.81e6	11212	81.05	2.37e-3	9742	90.69	1.39e-3	-13.11	11.90	-41.50
iirDemo	2.31e5	10168	70.56	9.35e-5	8112	90.48	3.70e-5	-20.22	28.22	-60.41
integrator	4.02e5	6224	87.06	7.84e-5	4108	96.27	3.59e-5	-33.99	10.57	-54.26
interp	9.35e5	8312	71.87	3.59e-4	6544	97.43	6.88e-5	-21.27	35.56	-80.85
scramble	4.23e6	13296	77.78	1.31e-3	11828	87.71	8.27e-4	-11.04	12.77	-37.03
Average			79.51			93.93		-27.90	18.76	-49.83

Table 8: Energy Break-Down: System with 4Kbyte Cache and Off-Chip FLASH.

Bench	*Energy Without Compression* [J]			*Energy With Compression* [J]			Δ [%]		
	Cache	*Bus*	*FLASH*	*Cache*	*Bus*	*FLASH*	*Cache*	*Bus*	*FLASH*
adaptFilter	1.59e-5	5.03e-5	1.17e-5	2.17e-5	7.58e-6	1.82e-6	36.45	-84.93	-84.41
butterfly	6.52e-6	5.41e-5	1.25e-5	9.47e-6	2.25e-5	5.39e-6	45.19	-58.46	-57.01
chaos	1.26e-5	1.13e-4	2.63e-5	2.00e-5	7.68e-6	1.84e-6	59.05	-93.23	-92.99
dft	2.35e-4	1.74e-3	4.03e-4	3.30e-4	8.54e-4	2.05e-4	40.48	-50.84	-49.12
iirDemo	5.37e-6	7.16e-5	1.66e-5	8.64e-6	2.29e-5	5.49e-6	60.96	-68.02	-66.91
integrator	1.15e-5	5.43e-5	1.26e-5	1.60e-5	1.60e-5	3.85e-6	38.80	-70.49	-69.47
interp	2.21e-5	2.74e-4	6.35e-5	3.77e-5	2.51e-5	6.02e-6	70.18	-90.83	-90.52
scramble	1.08e-4	9.78e-4	2.27e-4	1.53e-4	5.43e-4	1.30e-4	41.56	-44.47	-42.53
Average							49.08	-70.16	-69.11