Code compression architecture for cache energy minimisation in embedded systems

L. Benini, A. Macii and A. Nannarelli

Abstract: Energy consumption of the processor-to-memory path normally accounts for a large fraction of the total energy budget of modern embedded systems. A novel approach for reducing energy consumption in core processors used in systems with cache-based architectures is present. In this scheme, instructions are fetched and stored in the I-cache in compressed form. The beneficial effect is an increase of the cache hit ratio; therefore, the number of accesses to the main memory is reduced, and so is the energy required to fetch the instructions. Static code size reduction is achieved as a by-product. A hardware decompression unit performs fast low-energy on-the-fly instruction decompression at each cache look-up. 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. The viability and effectiveness of this solution is assessed through extensive benchmarking performed on a number of typical embedded programs. Beside code size, energy and performance optimisation results, the authors also report data regarding the synthesis and implementation of the decompression unit. The energy penalty it introduces is taken into account in the evaluation of the achieved energy savings.

1 Introduction

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, fuelled by the new generation of embedded computing tasks (e.g. signal processing, high-bandwidth data transfer etc.) have changed the picture. Currently, many embedded processors are based on high-performance RISC architectures, with an onchip cache [1] and full support for complex memory systems and peripheral controllers [2]. Such processors, and their software development environments, are usually purchased by system integrators from third-party companies that specialise 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 into devising techniques for improving code density and reducing instruction-related costs, in terms of speed, area and energy [3].

IEE Proceedings online no. 20020467

DOI: 10.1049/ip-cdt:20020467

Paper first received 31st October 2001 and in revised form 29th April 2002

L. Benini is with the Università di Bologna, Bologna 40136, Italy
E-mail: lbenini@deis.unibo.it
A. Macii is with the Politecnico di Torino, Torino 10129, Italy
A. Nannarelli is with the Università di Roma Tor Vergata, Roma 00133, Italy

Numerous code compression techniques have been proposed for reducing instruction memory size in low-cost embedded applications [4]. The basic idea was to store programs in compressed form and decompress them on-the-fly at execution time. Later, researchers realised that code compression could also be beneficial for energy, because it reduces the energy consumed in reading instructions from memory and communicating them to the processor core [5–7].

Code compression leverages well known lossless data compression techniques [8], but is characterised by two distinctive 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, traditional lossless techniques that decompress a stream starting from a single initial point are not applicable without changes. Secondly, the decompressor should be small, fast and energy-efficient, because its hardware cost must be amortised by the corresponding savings in memory size and energy, without compromising performance.

For simple processors with no instruction cache, the hardware decompression block is either merged with the processor core itself, or placed between the program memory and processor. The first solution has been implemented in several commercial core processors, in the form of a 'dense' instruction set, with short instructions (e.g. ARM Thumb [9] and MIPS16 [10] instruction sets). The second solution has been investigated in several papers [5, 6, 11, 12]. 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 (DCR) architecture), or it can be placed between the processor and the I-cache (decompress on fetch (DF) architecture). Both alternatives have been investigated [7, 13]. In [7] it was shown that from an energy and performance viewpoint, the DF architecture is superior to the DCR architecture, when the 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 [14, 15], indicating that reducing decoding overhead is a nontrivial 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 vital for making DF applicable in practice.

This paper proposes a 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. Secondly, the compression algorithm has been specifically designed for fast low-energy decompression during cache look-up. Compressed instructions are always aligned to cache line boundaries, branch destinations are word-aligned and instruction decompression is based on a single look-up into a small (and fast) memory buffer. Thirdly, 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 type of adopted program memory (i.e. onchip against offchip). For example, for a 4 kbyte cache and an onchip 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 offchip program memory. Cache performance is also very sensitive to cache and code size. For a 4 kbyte 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.* [13], and by Lekatsas *et al.* [7]. The Larin *et al.* 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. Larin *et al.* [13] does not report any data on energy, speed or area of the decompressor, 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 quite a large hardware block). Furthermore, the branch target addresses in compressed code are stored in a dedicated address remapping memory that must be accessed on every taken branch.

The Lekatsas *et al.* [7] 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 immediates are compressed using arithmetic coding. For branches, only destination displacement is variable-length coded. Fast dictionary instructions, with no immediates are compressed to a one-byte code and decompressed using a 256-entry look-up memory. Uncompressed instructions are left intact, and extended with a 3-bit preamble. Even though Lekatsas *et al.* report on decoding energy consumption, no information is provided in [7] on the decompressor area, speed and its interaction with cache (for instance, cache line unpacking, and branch destination remapping in the case of nonword-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 four-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 [16] as the target core processor, because it has a simple 32-bit RISC architecture with several open-source synthesisable 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 S_N of the N most frequently executed instructions is obtained. Without loss of generality, we assume in the following that $N = 256$. Similarly to [6, 7] (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 to compress an instruction i, even if it belongs to set S_N, depends on the neighbouring instructions that would fit into 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 into a single cache line when it is cached. The detailed structure of a compressed line is shown in Fig. 1.

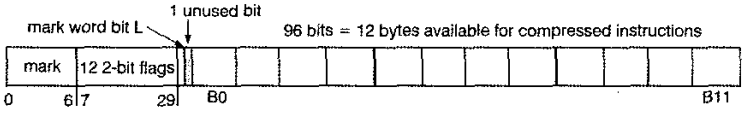

Fig. 1 *Compressed line structure*

The first word of the line contains a mark and a set of flag bits. The mark is an unused instruction opcode (DLX, and most processors, have a few unused opcodes. Unused opcodes are intentionally made available to give the system developers the possibility of building monitors and debuggers) (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).

A compressed line is constructed to store between a minimum of five instructions, and a maximum of 12 instructions. Thus, the best compression achievable is a factor of four. The compression algorithm analyses the code sequentially, starting from the first instruction (assuming that it is aligned to the cache line boundary), and tries to pack adjacent instructions in compressed lines. If a group of instructions can be advantageously packed (i.e. if the compressed line contains more than four 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, the instructions are left uncompressed even if they belong to $\mathcal{S}_{\mathcal{N}}$.

The compression procedure guarantees that the 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 the number of bits transferred during uncompressed code execution. It should be noted that these properties seem to contradict a basic rule of lossless data compression, which states that every conceivable compression algorithm is ineffective for some data sequences, and it increases their size [8]. This rule holds if we assume that every possible sequence has nonnull probability. Fortunately, this is not the case for instruction streams. More specifically, the existence of unused opcodes implies that not every bit stream can be a valid instruction trace. Our compression algorithm effectively exploits this property to ensure that code size increase is not possible. It is important to note that these desirable properties do not hold jointly for any previously published code compression algorithm.

To facilitate an efficient hardware implementation of the decompressor, 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 [13].
- Branch destination instructions are always word-aligned. This rule allows us to eliminate the address translation table, since branch addresses in the 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 the compressed lines may be left empty. However, this effect is taken into account when deciding whether or not a group of instructions can be stored in a compressed line.

Example 1: Fig. 2 shows an example of code compression. In the original memory map, instructions are marked with `CI` if they belong to $\mathcal{S}_{\mathcal{M}}$ (i.e. candidates for compression), with `UI` if they do not. The 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 nine `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 `UI`s, compression would be convenient only if we had at least three consecutive `CI`s, but we have only two. Hence, the `UI–D` and the following three instructions are stored uncompressed in memory. It should be noted 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 seven

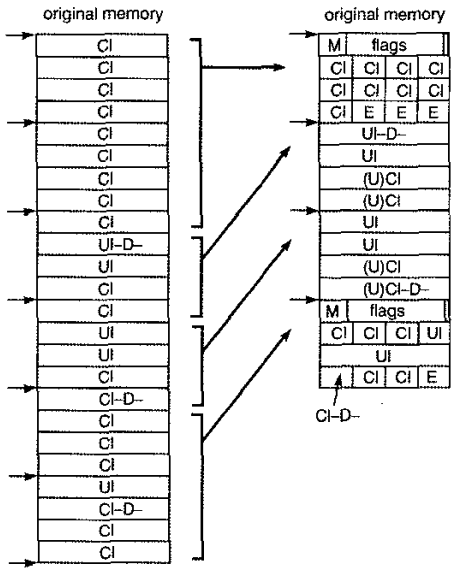

Fig. 2 *Code compression example*

instructions are stored in a compressed line. It should be noted that: (i) the compressed line contains an uncompressed instruction; (ii) the first empty byte is due to the branch alignment constraint (a branch destination must be word-aligned). The flags for the second compressed line are: 00.00.00.01.01.01.01.11.00.00.10.11.1. For this example, the compression reduced code size from 24 words to 16.

During program execution, at every cache miss, a new cache line is fetched from the 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 Section 3. 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 unit.

- A master controller, called the 'main controller' in the sequel, which handles the communication with the CPU.
- A slave controller, called the 'miss handler' in what follows, which supervises the transfers from main memory to the cache when a read miss occurs.
- A program counter update unit (PC-update).

The decompression-cache system, sketched in Fig. 3, is completed by the following units:

- The cache array of 256×4 words, plus tags of 20 bits.
- A multiplexer placed at the output of the cache array.
- A memory holding the compressed instructions (instruction decompression table (IDT)).
- A multiplexer 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).

When a memory read access is issued from the CPU (signal fetch request the address (index) is passed to the cache and the main controller performs the tag checking.

If the memory access is a cache hit, the main controller executes the following tasks:

(i) It reads the first word of the block (or cache line) to check if the block in question contains at least a compressed instruction. This is done by matching the first six bits (opcode) with the pattern indicating a compressed line.
(ii) According to the result of the matching, it selects the appropriate instruction by setting the control lines of the cache mux and the data bus mux.

If there is a tag mismatch (i.e. a miss) then

(i) The miss handler is woken up by the main controller.
(ii) It starts to transfer the requested block (four words in our case) from the memory to the cache, co-ordinating the hand-shaking between the two units.
(iii) On transfer completion:
- The miss handler signals the end of the transfer to the main controller.
- It puts its command and address buses toward the cache in high-impedance.
- It sets itself in the idle state.

(iv) The main controller executes the same tasks as in the case of a 'hit'.

If the cache line contains at least one compressed instruction, the main controller performs some additional operations: (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 instruction requested. If the instruction is compressed, the main controller issues the address of IDT to be accessed, otherwise it simply selects the

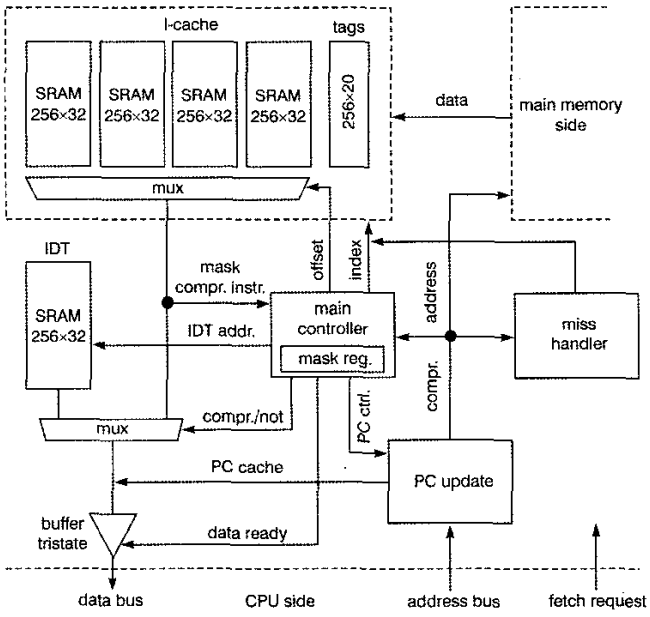

Fig. 3 *Implementation of the decompression system*

appropriate word by operating the cache mux. Finally, it executes task 2 above.

To keep track of the instruction to fetch, in the case of compression, the main controller updates a four-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 with the following policy:

- Increment by one, if current instruction is compressed.
- Increment by four, if the current instruction is not compressed.

The CPU does not know whether or not the instruction under execution is compressed. Therefore, the program counter (PC) is incremented by four for each instruction, assuming sequential code. However, compressed instructions in the cache occupy one byte instead of four, 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 PC-relative jumps, we can opt for one of two solutions:

- If the CPU core is modifiable, we can stop the PC increment when reading word multiple compressed instructions inside the same cache. This can be done with minimal 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).

We adopted the second solution, which is applicable to any embedded core (even those provided as black boxes by third-party developers). This approach implies an increase in the complexity of the hardware decompressor: 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). These registers allow jump destination address reconstruction in the case of PC-relative jumps.

However, an additional problem arises in the case of subroutine calls. The DLX instruction 'jal address' performs two tasks upon execution:

(i) The PC (return address) is stored in register 31.
(ii) The CPU jumps to 'address'.

To have the correct return address from the subroutine we have to store, in register 31, the cache PC instead of the CPU PC. This can be done by changing the instruction 'jal address' (during compression) with two instructions (which are eventually compressed)

'lw r 31, (don't care)' and 'j address'

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
$t_{fetch\text{-}wo}$	4.7	
$t_{fetch\text{-}dec}$		6.2

Table 2: Energy per cycle of units in Fig. 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 update		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

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 r31'), not only the correct address of the compressed program is restored, but the CPU PC is also realigned.

The decompressor also puts an overhead on the timing. In normal cache operation the time required to fetch the instruction, in the case of a 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 necessary for tag-matching, and t_{busEN} is the time to enable the tristate buffer if a hit has 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 and t_{busIN} is the transfer time through the tristate buffer. In this case, the delay introduced by the decompressor is larger than the time needed to enable the tristate buffer (t_{busEN}).

We used a 0.25 μm, 2.5 V 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 the Synopsys Design Compiler, and energy estimation using the synopsys power compiler. In Table 2 the sparse logic row refers to the control logic that handles the communication between the units and that manages the compression/decompression mechanism when the decompression unit is adopted. For this reason, the energy value in the case 'with decompressor' is around six times greater than the case 'without decompressor'.

4 Experimental results

We now report experimental data concerning the use of the proposed compression scheme. The software programs we considered for the experiments are some of the C benchmarks distributed along with Ptolemy [17]; they implement software functions that are widely exploited in embedded systems for DSP. Data collection has been carried out using the SuperDLX [18] compilation and simulation environment, and two different design scenarios have been explored. In the first, the program memory is built as an onchip RAM. In the second, it is built as an offchip FLASH memory.

For the first design option, we have used, as program memory, a 512 kbyte SRAM from ST, organised in eight banks of $32k \times 16$ and built in 0.25 μm technology at 2.5 V. The energy access cost, in read mode, is 17.2 nJ

Table 3: Results for a 4 kbyte cache and onchip 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

per memory block of four, 32-bit words. The line capacitance we used to determine the address and data bus energy is 0.6 pF.

Regarding the second design scenario, where accesses to the background memory go offchip, the memory is a 4 Mbit FLASH at 2.5 V from ST and the energy cost for accessing a four-word location is 21.4 nJ. The bus load we have assumed, in this case, is 8 pF per line.

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

Fig. 4 summarises the results. The bar-chart reports results for the two design options mentioned above.

For the first solution, the average code size reduction is around 28%, with a peak value around 61% for the benchmark program Chaos. The cache performance improves, on average, by approximately 19%, with a maximum increase in the hit ratio of 34% for the Integrator benchmark program. Finally, energy decreases, on average, by 30%, with peak reductions for the Chaos and Interp benchmark programs of around 48% and 53%, respectively.

Concerning the second scenario, although compression ratios and hit ratios remained unchanged, larger energy savings are observed (i.e. the average is around 50%) due to the greater penalty effect introduced by the offchip bus and main memory accesses.

The energy savings are further detailed in Fig. 5, 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 with respect to 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 compensates well for the overhead of the decompressor.

Fig. 6 compares cache performance and energy consumption variations as the cache size changes (we consider 8 kbyte, 4 kbyte and 2 kbyte 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 also accessed for the case of uncompressed programs. This causes a substantial decrease in bus and memory energy for both uncompressed and compressed execution. In addition, the cache access cost is higher and the decompression overhead is no longer compensated due to the lower absolute contribution of the bus and main memory to the total energy. As a consequence, the usefulness of compressing the code decreases: the average energy savings are 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, the cache access cost decreases. As a result, the energy reductions are greater. From the above observations we can conclude that the relationship between energy, code compression ratio and cache performance (i.e. hit ratio) is very complex, and does not allow us to come up with intuitive 'rules of thumb' for designing optimal cached-code compression systems. Intensive

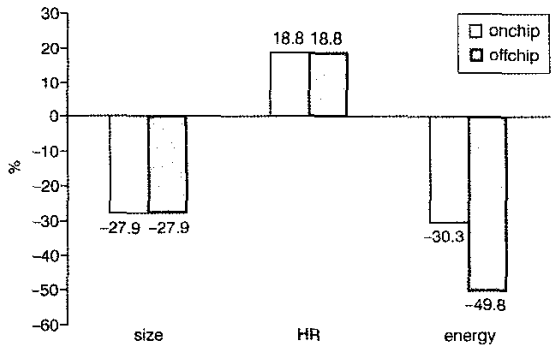

Fig. 4 *Size, hit ratio and energy variations for 4 kbyte cache*

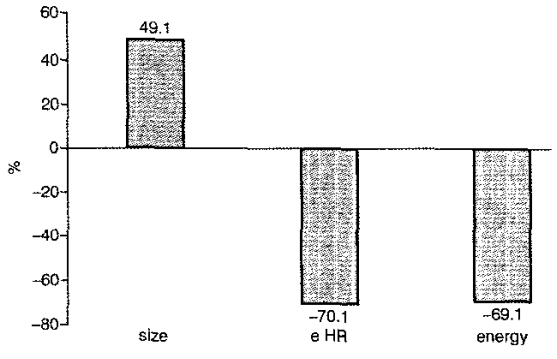

Fig. 5 *Break down of energy savings (4 kbyte cache and onchip SRAM)*

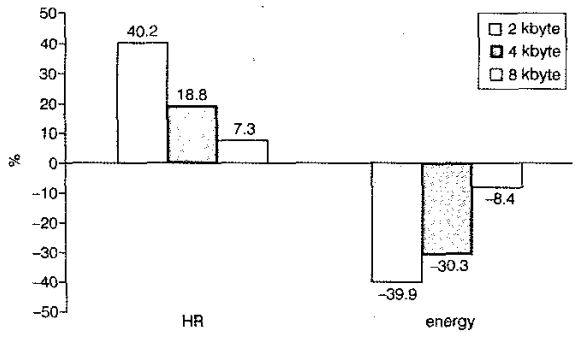

Fig. 6 *Hit ratio and energy variations for caches of different sizes and onchip SRAM*

code profiling and dynamic simulation are thus the key to achieving a satisfactory trade-off.

5 Conclusions

We have proposed an approach for reducing static code size and instruction fetch energy for cache-based processors used in embedded systems. In the proposed scheme, instructions are fetched and stored in the I-cache in compressed form. The benefit we obtain is an increased hit ratio and, as a consequence, a reduction of the amount of access to the main memory. Therefore, the energy required to fetch the instructions is minimized. Static code size reduction is achieved as a by-product. The method couples efficiency in compression with a low-overhead implementation of the decompression unit. The decompressor is placed outside the core boundary; therefore, processor architecture does not need any modification, making the proposed compression approach suitable for IP-based designs.

Code size, energy and performance improvements provided by our code compression scheme have been measured, taking into account the impact of the decompression unit (which has been designed and synthesized in hardware). The results of extensive benchmarking have demonstrated the effectiveness of the compression algorithm, which produces an average code compression of around 28% and an average energy saving of about 30%.

6 References

1 KO, U., BALSARA, P.T., and NANDA, A.K.: 'Energy optimization of multilevel cache architectures for RISC and CISC processors', *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, 1998, **6**, (2), pp. 299–308
2 SANTHANAM, S., BAUM, A.J., BESTUCCI, D. et al.: 'A low-cost, 300- MHz, RISC CPU with attached media processor', *IEEE J. Solid-State Circuits*, 1998, **33**, (11), pp. 1829–1839
3 BAJWA, R., HIRAKI, M., KOJINA, H. et al.: 'Instruction buffering to reduce power in processors for signal processing', *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, 1997, **5**, (4), pp. 417–424
4 LEFURGY, C.: 'Efficient execution of compressed programs'. 2000, Doctoral Dissertation, University of Michigan
5 YOSHIDA, Y., SONG, B.-Y., OKUHATA, H., ONOYE, T., and SHIRAKAWA, I.: 'An object code compression approach to embedded processors'. Proceedings of the International Symposium on Low power electronics and design, 1997, pp. 265–268
6 BENINI, L., MACII, A., MACII, E., and PONCINO, M.: 'Selective instruction compression for memory energy reduction in embedded systems'. Proceeding of the International Symposium on Low power electronics and design, 1999, pp. 206–211
7 LEKATSAS, H., HENKEL, J., and WOLF, W.: 'Code compression for low power embedded systems design'. Design automation conference, 2000, pp. 294–299
8 BELL, T., CLEARY, J., and WITTEN, I.: 'Text compression' (Prentice-Hall, Englewood Cliffs, 1990)
9 SEGARS, S., CLARKE, K., and GOUDGE, L.: 'Embedded control problems thumb and the ARM7TDMI', *IEEE Micro*, 1995, **15**, (5), pp. 22–30
10 KISSEL, K.Silicon Graphics MIPS Group: 'MIPS16: High-density MIPS for the embedded market. Technical Report, 1997
11 LIAO, S.Y., DEVADAS, S., and KEUTZER, K.: 'Code density optimization for embedded DSP processors using data compression techniques', *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, 1998, **17**, (7), pp. 601–608
12 LEKATSAS, H., and WOLF, W.: 'Code compression for embedded systems'. Design automation conference, 1998, pp. 516–521
13 LARIN, S., and CONTE, T.: 'Compiler-driven cached code compression schemes for embedded ILP processors'. MICRO-32, 1999
14 IBM, 'CodePack power PC code compression utility', User's manual version 3.0, 1998.
15 LEFURGY, C., PICCININI, E., and MUDGE, T.: 'Evaluation of a high performance code compression method'. MICRO-32, 1999, pp. 93–102
16 HENNESSY, J.L., and PATTERSON, D.A.: 'Computer architecture – a quantitative approach' (Morgan Kaufmann, San Francisco, CA, 1996, 2nd edn.)
17 DAVIS II, J., GOEL, M., HYLANDS, C. et al.: Overview of the Ptolemy project, ERL technical report UCB/ERL No. M99/37' (UC Berkeley, 1999)
18 MOURA, C. McGill University: 'Super DLX: a generic superscalar simulator'. 64, ACAPS technical memo, 1993