# Selective Instruction Compression for Memory Energy Reduction in Embedded Systems

Luca Benini [#] Alberto Macii [‡] Enrico Macii [‡] Massimo Poncino [‡]

[#] Università di Bologna
Bologna, ITALY 40136

[‡] Politecnico di Torino
Torino, ITALY 10129

**Abstract**

*We propose a technique for reducing the energy required by firmware code to execute on embedded systems. The method is based on the idea of compressing the most commonly executed instructions so as to reduce the energy dissipated in memory accesses. Instruction decompression is performed on the fly by a hardware module located between processor and memory: No changes to the processor architecture are required. Hence, our technique is well-suited for systems employing IP cores whose internal architecture cannot be modified.*

*We describe a number of decompression schemes and architectures that effectively trade off hardware complexity for memory energy and bandwidth reduction, as proved by experimental data collected by executing several sample programs.*

## 1 Introduction

Power optimization for embedded systems is an active area of research that has received considerable attention in recent times. On one hand, hardware/software partitioning for low power and software power optimization are key steps to achieve a global control of power dissipation. On the other hand, hardware power minimization is still essential, especially if it is targeted at a very high level of abstraction [1].

A major contributor to the system power budget is the memory-processor interface [2]. For this reason, several techniques that allow a reduction of this component of the dynamic power have been proposed in the literature. They can be categorized in two broad classes: *Bus encoding* techniques [3, 4, 5, 6, 7] and *memory organization* techniques [8, 9, 10, 11].

Bus encoding schemes reduce interface power by changing the format of the information transmitted on the processor-memory bus. In this way, the switching activity on the bus gets minimized, and so does the power. Memory organization methods change the way information is stored in memory so that the address streams generated by the processor have already low transition activity. Also in this case power savings come solely from a reduced switching activity on the bus.

Although bus power is relevant, additional improvements can be achieved by minimizing the dissipation due to memory accesses. Power minimization through instruction memory bandwidth optimization has been first exploited in the ARM7TDMI core [12]. Here, a 16-bit instruction set (called Thumb) consisting of 36 instructions is supported besides the regular 32-bit instruction set. In order to exploit Thumb instructions, the architecture of the basic processor core has been modified. Moreover, software tools for generating Thumb machine code are obviously required and are supplied by ARM to the users of this processor.

An alternative approach to memory bandwidth reduction has been presented in [13]. The basic assumption of this method is that the firmware running on a given embedded processor normally uses only a small subset of the instructions supported by the processor. By replacing such instructions with binary patterns of limited width (i.e., $\lceil \log_2 N \rceil$, where $N$ is the number of distinct instructions appearing in the code), memory bandwidth usage can be reduced, thus decreasing the total energy.

The solution of [13] does not require the availability of ad-hoc source-code compilers; in fact, the original machine instructions (by instruction we intend the complete $k$-bit pattern stored in memory, i.e., op-code and operand(s), if any) can be automatically replaced by $\lceil \log_2 N \rceil$-bit instructions by means of a script after the subset of instructions used by the program is identified through execution profiling or instruction-level simulation, and the number $\lceil \log_2 N \rceil$ is determined. The original machine code can thus be *compressed* to reduce the memory bandwidth that is needed to run the program. The so-called *instruction decompression table* and the related control circuitry can be designed and placed between the processor and the memory. Hence, the architecture of the core processor is left unchanged. This is a big plus for system designers employing third-party, off-the-shelf cores and microcontrollers that are either not disclosed (IP hard or soft macros) or not easily modifiable.

Data compression techniques have been used to reduce the size of executable programs; new ideas on the subject have thus been explored, especially for what concerns the domain of embedded systems (see, for example, [14] and [15]). However, memory occupation, rather than memory energy consumption has always been considered as the objective of the optimization. To the best of our knowledge, the approach of [13] is thus the first one that explicitly targets memory energy optimization.

In this work we present a technique that builds upon the method of [13] by overcoming its major limitation: If the number of instructions used by the embedded software gets large, so does the number of bits of the compressed instructions. Besides increasing the size of the instruction decompression table, this may excessively complicate the implementation of the control logic that handles instruction fetching and decoding, especially when the bit-width of the compressed instructions is not compatible with the available memory addressing scheme (e.g., bit-width different from a multiple of 8 on a byte-addressable memory).

We move from the observation that the number of machine instructions used by most software programs, although limited with respect to the total number of instructions supported by the processor, has a highly non-uniform statistical distribution. In other words, some instructions are usually much more used than others. This claim is confirmed by some experiments we have run on the MIPS R4000 RISC processor. We have profiled the execution of several software applications, we have determined how many times the various instructions are executed, and in Figure 1 we plot the results regarding the 256 most used instructions. The value reported on the $y$ axis is the percentage with respect to the total number of executed instructions.

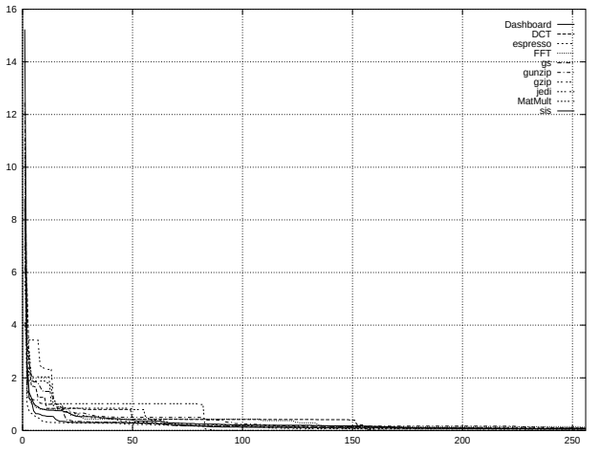


Figure 1: Profiling Results (Top 256 Instructions).

Table 1 provides a more complete summary of the profiling experiments; in particular, it gives the total number of executed instructions and the total number of distinct instructions. The total number of times (percentage with respect to the total number of executed instructions) the 256 most frequent instructions are executed is also reported (column *Percentage 256*).

| *Program* | *# of Executed Instructions* | *# of Distinct Instructions* | *Percentage 256* |
|---|---|---|---|
| Dashboard | 3.06590e04 | 1972 | 75.077% |
| DCT | 8.52347e08 | 366 | 99.999% |
| espresso | 1.75454e06 | 5754 | 54.360% |
| FFT | 1.32774e05 | 856 | 93.186% |
| gs | 4.04896e05 | 3546 | 79.142% |
| gunzip | 6.40040e04 | 1469 | 92.422% |
| gzip | 1.19116e05 | 1619 | 89.238% |
| jedi | 1.46909e07 | 3352 | 88.456% |
| MatMult | 4.81175e07 | 246 | 100.000% |
| sis | 7.47147e07 | 13609 | 61.111% |

Table 1: Complete Profiling Results.

In view of these results, we propose to consider for compression only the instructions used by the embedded code with the highest execution probability. This solution allows us to fix *a priori* the bit-width of the compressed instructions (i.e., 8 bits, in our particular case); the two-fold advantage we get from this choice is that the size of the instruction decompression table is fixed and limited, and the instruction fetching/decompression logic has reduced complexity. We discuss four different architectural options for implementing such logic, and we compare their relative merit, in terms of achievable energy savings and execution time improvement/degradation, from both the theoretical and the practical points of view.

## 2 Memory Energy Reduction by Compression

We consider the processor-memory architecture of Figure 2(a). For energy minimization purposes, in [13] such architecture has been modified as depicted in Figure 2(b).

The program is stored in memory in compressed format, i.e., each instruction is replaced with a $\lceil \log_2 N \rceil$-bit binary pattern which is in one-to-one correspondence with the original instruction. Every time an instruction is fetched from the memory, it is first decompressed (i.e., the original format is restored) by means of the *instruction decompression table (IDT)* and then passed to the processor's decoding logic.

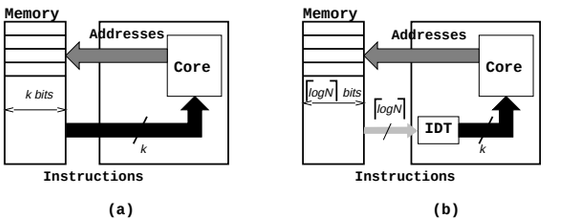


Figure 2: Original Architecture (a) and Modification of [13] (b).

This architecture is motivated by the fact that software programs normally use only a subset, of cardinality $N$, of all possible instructions offered by the processor's instruction set. Since the width of the uncompressed instructions, $k$, is wider than $\lceil \log_2 N \rceil$, accessing the memory to retrieve the compressed instructions has the beneficial effect of reducing both memory energy and bus power with respect to the reference case.

Although in principle the solution illustrated above offers good opportunities for energy optimization, as discussed in [13], there is a number of issues that need to be addressed to make this proposal applicable in practice.

It happens quite often that the number of distinct instructions, $N$, used by a program is not so small. This statement is supported by the data of Table 1; out of the 10 programs that we have profiled, only one (`MatMult`) has less than 256 distinct instructions. It is then quite evident that the idea of compressing *all* the instructions appearing in the code, as proposed in [13], has three major disadvantages. First, the size of the IDT can become very large, therefore area and power demanding. Second, the bit-width of the compressed instructions ($\lceil \log_2 N \rceil$) may become comparable to the bit-width of the original instructions ($k$), thus making negligible the reduction in memory bandwidth. Third, in case the memory is not bit-addressable, values of $\lceil \log_2 N \rceil$ which are not multiples of 8 can not be handled very efficiently. In other words, storing the compressed code in memory may result in a waste of space (for example, for $\lceil \log_2 N \rceil = 9$, two bytes are required to store each compressed instruction). This problem can be solved by making the memory bit-addressable (if possible); however, the cost of the address decoding circuitry may become sizable.

One way of overcoming the problems listed above is suggested by the profiling data of Table 1. For all the programs, out of the total number of distinct instructions, only a few are executed very often. In particular, the 256 most used instructions are always executed for at least 50% (program espresso) and up to 99.99% (program `DCT`) of the time.

To take advantage of this result, we propose to compress only a subset of fixed cardinality (256 elements, in our specific case) of the instructions used by a program, namely, those that are executed more often; less probable instructions are left unchanged and stored as they are in memory. This choice guarantees a fixed and limited size for the IDT, as well as a fixed compression ratio for the 256 most used instructions (i.e., $k/8$). On the other hand, it requires the introduction of a controller that properly handles instruction fetching. This is because the program stored in memory is a mix of compressed (many) and uncompressed (few) instructions that must be manipulated differently. The solution illustrated above can be implemented as in Figure 3. Several options are possible to realize it, and they depend on a number of factors, such as memory organization, desired energy savings, allowed execution time penalty (if any), complexity of the controller, type of program the system will execute. In the next section we present four architectural schemes and we discuss their relative characteristics, advantages and limitations.

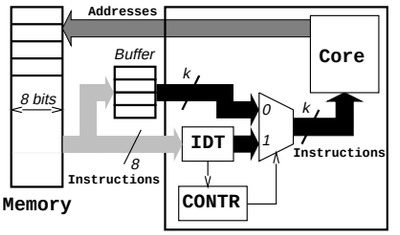


Figure 3: Proposed Architecture.

# 3 Architectures for Memory Energy Minimization

## 3.1 Assumptions

We consider only RISC processors, such as those belonging to the MIPS and the ARM families. This implies that all the supported instructions, including the op-code and the operand(s), if any, have the same length ($k = 32$ bits, in our specific case) and that they can be fetched from the memory in one bus cycle. We assume instruction and data memory to be separated. This is quite common for embedded systems, since the machine code of the program is fixed and resides in either a ROM or a FLASH memory, while data are normally stored in a RAM.

We consider byte-addressable memory and we assume that each memory bank (which is made of 8-bit locations) can be disabled by the processor on a bus cycle-by-cycle basis, for instance by selectively de-activating the chip select input of the bank.

Our compression approach requires the storage in memory of both compressed and uncompressed instructions; the control logic that regulates instruction fetching must then be able to distinguish between the two cases. We reserve one 8-bit code, out of the 256 available for the compressed instructions, and place it in memory right before every uncompressed instruction. We refer to this reserved 8-bit pattern as *the mark*. This technique is in sharp contrast with the one implemented in the ARM7TDMI processor, where processing of Thumb (16-bit) instructions is activated/de-activated by specific instructions included in the processor's assembly language. Although our choice may not be optimal regarding the size of the compressed machine code, it allows us to leave the architecture of the processor unchanged; thus, it avoids the need of designing a new core or microcontroller and its related software development environment.

## 3.2 Evaluation Metrics

The architectural solutions that we present in the sequel will be evaluated with respect to the following two metrics:

- Total *dynamic memory utilization*, $E$, required by a program, measured as the total number of accesses to 8-bit memory locations during instruction fetching. For a fixed instruction memory architecture, $E$ is roughly proportional to the total energy spent in fetching instructions from memory.
- Total *bus utilization*, $T$, required by a program to complete, measured as the total number of bus cycles (that can correspond to 8-bit or 32-bit read operations, depending on memory organization). Usually, bus cycle time is much slower than processor clock period, hence decreasing $T$ improves performance (i.e., program execution time).

In our analysis, $E$ and $T$ are evaluated for different values of the compression ratio, $R$ (defined as the fraction of compressed instructions over the total number of executed instructions). More precisely, $R = 0$ indicates no compression, while $R = 1$ corresponds to the case where all the instructions have been compressed. For clarity, we consider normalized values of $E$ and $T$ (w.r.t. the value of the reference architecture of Figure 2(a)).

## 3.3 Code Compression Schemes

We have devised four architectural schemes of increasing complexity that differ in the way memory is organized and accessed.

### 3.3.1 Architecture 1

The program memory consists of one, 8-bit bank. The accesses to such memory are thus always 8-bit wide. In the worst case ($R = 0$), this architecture requires five, 8-bit memory accesses for each uncompressed instruction (the mark plus the 32-bit instruction). The bus utilization $T$ is then 5 times the original, while the dynamic memory utilization $E$ is 1.25 times the original. In the best case ($R = 1$), the bus utilization equals that of the original architecture (there is one, 8-bit memory read per instruction instead of four), while the total $E$ is reduced to 0.25 of the reference value. $T$ and $E$ decrease linearly with respect to $R$:

$$T(R) = 5 - 4R \qquad E(R) = 1.25 - R$$

### 3.3.2 Architecture 2

The program memory consists of four, 8-bit banks. Compressed instructions are fetched with 8-bit reads (during this phase, the three unused memory banks are disabled to avoid useless power dissipation). On the contrary, uncompressed instructions require one, 8-bit access (for the mark) plus one 32-bit access for the instruction. As a consequence, in the worst case, the program requires exactly twice as many bus cycles than in the reference architecture (i.e., $T = 2$), while the total dynamic memory utilization is again 1.25 of the original (five, 8-bit memory accesses are required to fetch one uncompressed instruction). In the best case, this scheme performs like the first one. Although there are four banks, only one of them is accessed, while the remaining three are disabled. Therefore, $T = 1$ and $E = 0.25$. Again, $T$ and $E$ decrease linearly with respect to $R$:

$$T(R) = 2 - R \qquad E(R) = 1.25 - R$$

### 3.3.3 Architecture 3

The program memory consists of four, 8-bit banks. Compressed instructions that are located consecutively are packed into 32-bit words (i.e., four compressed instructions per word). However, any time an uncompressed instruction occurs, a mark is placed back-to-back with the last compressed instruction and the uncompressed instruction is stored in the following 32-bit memory location. This implies that uncompressed instructions are always word-aligned, and therefore there is a chance that some 8-bit memory locations are left unused.

Unlike the previous architectures, the formula that gives the value of $T$ as a function of $R$ does not represent a single curve but, rather, a whole family. The parameter that distinguishes the various members of the family is the relative placement (i.e., the factor of interleaving) between compressed and uncompressed instructions. All the curves in the family share the extreme points. In particular, for $R = 0$, eight 8-bit memory accesses are required to fetch each instruction (four bytes for the mark and four bytes for the instruction); therefore, $T = 2$. For $R = 1$, four compressed instructions are always packed into a memory word, resulting in a four-fold reduction; therefore, $T = 0.25$. On the other hand, it can be demonstrated that the line obtained by interpolating the two extremes represents the upper bound (corresponding to non-interleaved placement of compressed and uncompressed instructions) of the family of curves. Clearly, since memory bank disabling is not allowed in this architecture, the curves of $E$ are similar. The equations representing the upper-bound lines of $T$ and $E$, called $T^U$ and $E^U$, are then:

$$T^U(R) = 2 - 1.75R \qquad E^U(R) = 2 - 1.75R$$

### 3.3.4 Architecture 4

This architecture is a variant of architecture 3, where the packing of the instructions is extended to the uncompressed ones. This implies that uncompressed instructions are in general not aligned to 32-bit boundaries, and that there are no unused 8-bit locations in the memory.

This complete filling of the memory makes execution time independent of the placement of the instructions, unlike architecture 3. The worst-case value of $T$ can be derived by observing that five bytes (the mark plus the four bytes of a uncompressed instruction) need to be read instead of four (the uncompressed instruction alone); thus, $T = 1.25$. Obviously, $E$ behaves similarly. The equations for $T$ and $E$ are then:

$$T(R) = 1.25 - R \qquad E(R) = 1.25 - R$$

## 3.4 Comparison and Discussion

To better clarify how the four architectures exploit the compression scheme, in Figure 4 we show how a sample code is stored in the memory in the four cases.

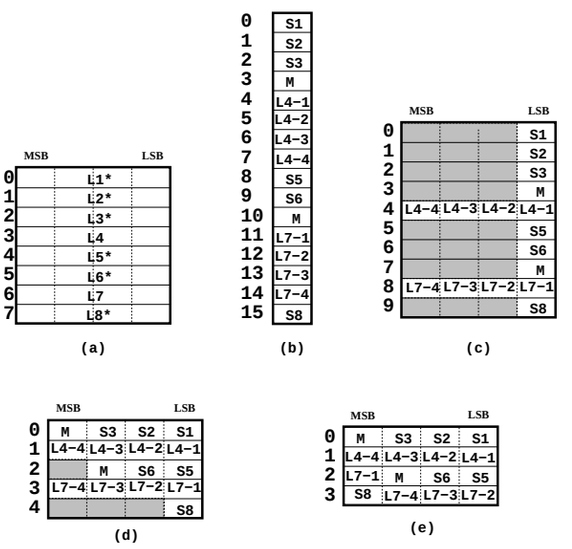


Figure 4: Memory Maps of a Sample Code Fragment.

Figure 4(a) depicts the code placement in the reference architecture. Here, all instructions are 32-bit long and are denoted with $L_i$. Those marked with a "$\star$" are the instructions that will be compressed.

The memory maps of the sample code fragment after compression are shown in Figures 4(b)-(e). Here, compressed instructions (8-bit long) are denoted as $S_i$, while the mark used to signal the presence of an uncompressed instruction is represented by symbol $M$. Also, the individual bytes of the generic uncompressed instruction $L_i$ are denoted with $L_i - k, k = 1, \ldots, 4$. In the pictures, the shaded areas indicate memory locations that do not contain useful information.

Figure 5 plots the normalized values of $E$ and $T$ as a function of the compression ratio $R$. The values are actually obtained by generating various streams with the desired value of $R$, and by evaluating the corresponding values of $E$ and $T$. For architecture 3, only the worst-case curves (i.e., $E^U(R)$ and $T^U(R)$) are plotted.

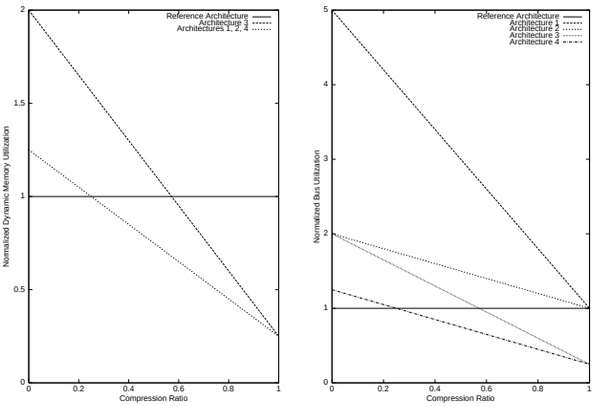


Figure 5: Plots of $E(R)$ and $T(R)$ for the Four Architectures.

## 3.5 Assignment of the Compressed Codes

In [13], the codes to the compressed instructions were assigned randomly. Obviously, this choice has no impact on memory access energy; however, if we consider architectures 1 and 2, we notice immediately that all the compressed instructions are stored in the same memory bank, say the right-most one. Therefore, the higher the value of the compression ratio, the higher the chance that a compressed instruction crosses the memory output pins of the right-most bank immediately after another compressed instruction.

We propose to assign minimum Hamming distance binary codes to the compressed instructions that have high probability of appearing in sequence on the memory output pins of the right-most bank. This will have the beneficial effect of reducing the switching power dissipated by charging and discharging the capacitances associated to the memory-bus interface. To perform this task we have resorted to the tool for low-power op-code generation described in [16]; the algorithms implemented in such tool are similar to those developed for state assignment in the logic synthesis domain, and rely on profiling information about the adjacency of instruction pairs.

As mentioned earlier in this section, non-random code assignment is only effective for architectures 1 and 2, while it does not provide any relevant benefit in the case of architectures 3 and 4. This is because instruction packing is likely to destroy the sequentiality of the compressed words at the output pins of the memory banks.

# 4 Architectural Issues

In this section we focus on the hardware architecture of the instruction decompression block that extracts standard, full-length instructions from a compressed instruction stream. The basic requirements for the decompressor implementation are:

- Decompression should be performed on the fly, in response to instruction fetches;
- The decompressor must be able to interface with the processor with the same protocol used by instruction memory;
- Decompression should be fast.
- The complexity of the decompressor should be kept as low as possible, because the power dissipated in decompression must be substantially lower than the power saved by reducing memory traffic.

We implemented the decompression block for a DLX processor [17] core. This choice is motivated by several factors. First, many synthesizable HDL models of DLX cores are available in the public domain (we used that of [18]), giving us the possibility of thoroughly testing the processor-decompressor interface with cycle-accurate HDL simulation. Second, DLX instruction set architecture is a pure 32-bit RISC. Several high-performance processor cores for embedded systems implement similar instruction set architectures (for instance, ARM and MIPS instruction sets are 32-bit RISC). Third, the basic DLX core implemented in [18] is simpler than most commercial core processors, thereby simplifying the development of our prototype decompressor.

Among the compression schemes presented in the previous section, we analyze the implementation of the third one, namely the architecture that reads four compressed instructions at a time from memory. Uncompressed instructions are flagged by the reserved mark byte. The presence of the mark in any word indicates that the next word in memory should be fetched, because it contains the uncompressed instruction.

The block diagram of the decompressor interface with processor and instruction memory is shown in Figure 6(a). The DLX core communicates with the decompressor using the standard instruction memory interface based on a four-phase protocol. At the beginning of every instruction fetch cycle, the processor outputs the instruction address on `Iaddrbus` and lowers the `IStart_L` signal. When the decompressor is ready to return a decompressed instruction, it outputs the instruction on `Instrbus` and lowers signal `IWait`. Upon receiving the new instruction, the processor raises `IStart_L`, and the compressor raises `Iwait`. Fetch cycle sequencing is shown in Figure 6(b).

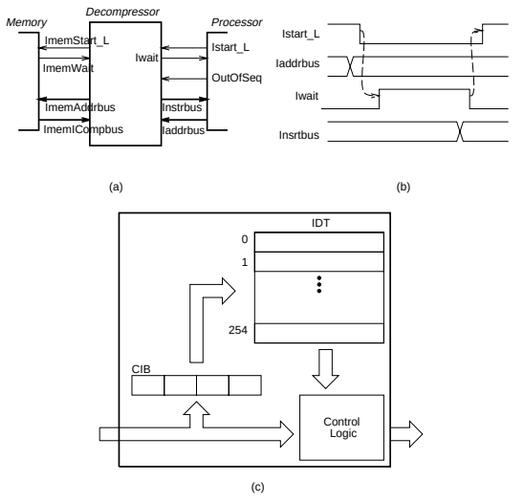


Figure 6: (a) Decompressor Interface; (b) Handshake Protocol Between Processor and Compressor; (c) Decompressor Block Diagram.

An additional signal is specified in the decompressor-processor interface, called `OutOfSeq`. Whenever `OutOfSeq` is high, the processor is issuing a fetch instruction which is not in sequence with the previous one (i.e., the target of a taken branch or jump). Notice that this signal can be autonomously generated by the decompressor, by simply observing when two consecutive addresses differs by a number larger or smaller than four. The purpose of `OutOfSeq` is to indicate that the decoder should read from memory directly at the address provided by the processor, by-passing its internal memory address generation logic (described later).

The decompressor-memory interface is based on the same protocol used in the decompressor-processor interface. The handshake signals are `ImemStart_L` and `ImemWait`. The address bus is `ImemAddrbus` and the compressed instruction bus from memory is `ImemICompbus`. One 32-bit word at a time is read from memory, which is equivalent to four compressed instructions.

The main functional blocks of the decompressor are shown in Figure 6(c). The instruction decompression table (IDT) contains 255 32-bits words. The address of each word is the compressed code of the instruction stored in the word. Decompression is performed simply by reading the content of the IDT at the address specified by the byte of the compressed instruction. The compressed instruction buffer (CIB) is a 32-bit register that can be accessed byte-by-byte. Words coming from memory are stored in the CIB and are decompressed one byte at a time. Finally, the control logic block coordinates interface signals, IDT look-up, CIB read/write and manages the direct transfer from memory to processor of uncompressed instructions. A key function of the control block is address generation. If the processor reads addresses in sequence, the controller generates one new memory address every four processor fetch cycles. In the remaining three cycles, compressed instructions are extracted from the CIB. On the contrary, if either the processor is fetching the destination address of a branch/jump or the mark is found, a new read cycle to instruction memory is initiated.

The hardware cost of the decompressor is dominated by the IDT (a $255 \times 32 = 8K$ memory). Similarly, decompression performance is set by IDT read time. When the compressor is processing compressed instructions, it performs one memory bus cycle every four fetch cycles. When it does not involve CIB refill, fetch time for compressed instructions reduces to IDT read time. This is the most common case. In the remaining cases fetch latency is longer. Memory access time plus IDT read time is the time required for fetching a compressed instruction immediately after a CIB refill. The worst case fetch time is experienced when the first instruction after a CIB refill is not compressed; here, two instruction memory reads are needed to fetch an instruction.

## 5 Experimental Results

In this section, we report experimental data concerning the use of the proposed compression schemes and the corresponding architectures. The results include both dynamic memory utilization ($E$) and bus utilization ($T$) figures, and the software programs used for the experiments are the same listed in Table 1. Program execution profiling has been performed using the `pixie` utility available in the MIPS development kit.

Table 2 shows the results. For each example, the total dynamic memory utilization (Column $E$), the total bus utilization (Column $T$), and the switching activity at the memory output pins (Column $SW$) are reported. Data are normalized to the corresponding values of the reference architecture.

As expected, dynamic memory utilization is substantially improved for all the compression schemes, with some advantage for architectures 1, 2 and 4. On the other hand, bus utilization is much more influenced by the chosen compression scheme. Clearly, there is a trade-off here. The more complex the decompression logic, the faster the execution time. Finally, the switching activity at the memory output pins decreases in most of the cases, although for this experiment compressed codes have been assigned randomly.

It is interesting to observe that program `MatMult` uses less than 255 distinct instructions. Consequently, the compression ratio is ideal (i.e., $R = 1$) and so are the the energy savings and the program execution times.

| *Application* | *Architecture 1* | | | *Architecture 2* | | | *Architecture 3* | | | *Architecture 4* | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | *E* | *T* | *SW* | *E* | *T* | *SW* | *E* | *T* | *SW* | *E* | *T* | *SW* |
| Dashboard | 0.50 | 1.99 | 0.68 | 0.50 | 1.24 | 0.81 | 0.65 | 0.65 | 0.76 | 0.50 | 0.50 | 0.66 |
| DCT | 0.25 | 1.00 | 0.31 | 0.25 | 1.00 | 0.30 | 0.25 | 0.25 | 0.30 | 0.25 | 0.25 | 0.20 |
| espresso | 0.70 | 2.82 | 1.05 | 0.70 | 1.45 | 1.32 | 1.00 | 1.00 | 1.28 | 0.70 | 0.70 | 1.07 |
| FFT | 0.32 | 1.27 | 0.44 | 0.32 | 1.06 | 0.48 | 0.36 | 0.36 | 0.43 | 0.32 | 0.32 | 0.38 |
| gs | 0.45 | 1.83 | 0.68 | 0.45 | 1.20 | 0.78 | 0.58 | 0.58 | 0.72 | 0.45 | 0.45 | 0.65 |
| gunzip | 0.32 | 1.30 | 0.44 | 0.32 | 1.07 | 0.47 | 0.37 | 0.37 | 0.43 | 0.32 | 0.32 | 0.40 |
| gzip | 0.35 | 1.43 | 0.47 | 0.35 | 1.10 | 0.51 | 0.42 | 0.42 | 0.49 | 0.35 | 0.35 | 0.47 |
| jedi | 0.36 | 1.46 | 0.44 | 0.36 | 1.11 | 0.52 | 0.44 | 0.44 | 0.51 | 0.36 | 0.36 | 0.45 |
| MatMult | 0.25 | 1.00 | 0.23 | 0.25 | 1.00 | 0.23 | 0.25 | 0.25 | 0.22 | 0.25 | 0.25 | 0.22 |
| sis | 0.64 | 2.55 | 0.89 | 0.64 | 1.38 | 1.15 | 0.92 | 0.92 | 0.95 | 0.64 | 0.64 | 1.22 |
| **Average** | **0.41** | **1.66** | **0.56** | **0.41** | **1.16** | **0.66** | **0.52** | **0.52** | **0.61** | **0.41** | **0.41** | **0.57** |

Table 2: Experimental Results.

In a second set of experiments, we have compared the switching activity results obtained after applying the low-energy code assignment algorithm to those determined with random code assignment. As mentioned in Section 3.5, the comparison is meaningful only for architectures 1 and 2. Table 3 summarizes the experimental data.

| *Application* | *Architecture 1* | | *Architecture 2* | |
|---|---|---|---|---|
| | *Rand Enc* | *Low-En Enc* | *Rand Enc* | *Low-En Enc* |
| Dashboard | 0.68 | 0.59 | 0.81 | 0.70 |
| DCT | 0.31 | 0.21 | 0.30 | 0.20 |
| espresso | 1.05 | 0.96 | 1.32 | 1.19 |
| FFT | 0.44 | 0.33 | 0.48 | 0.34 |
| gs | 0.68 | 0.54 | 0.78 | 0.63 |
| gunzip | 0.44 | 0.36 | 0.47 | 0.36 |
| gzip | 0.47 | 0.40 | 0.51 | 0.41 |
| jedi | 0.44 | 0.36 | 0.52 | 0.42 |
| MatMult | 0.23 | 0.14 | 0.23 | 0.14 |
| sis | 0.89 | 0.80 | 1.15 | 1.02 |
| **Average** | **0.56** | **0.47** | **0.66** | **0.54** |

Table 3: Impact of Low-Energy Compressed Code Assignment.

As expected, the switching activity at the memory-bus interface is further reduced by approximately 15% for both architecture 1 and architecture 2, without any penalty in memory access energy and execution time.

These results show that our instruction compression scheme has good potential for reducing the power consumption in instruction memory and memory-processor interface. The drastic reduction of $E$ implies that memory reads are reduced, thereby saving active memory power. The reduction of the number of bus cycles observed for architectures 3 and 4 is an indication that further power reductions could be achieved by trading off some performance slack for decreased power (for example, by choosing slower and less power-consuming memories). Needless to say, further investigation in required to assess the impact of the decompression block on the cycle time of the processor. Regarding the decompressor's power dissipation, the small size of the IDT and of the surrounding control logic does not rise major concerns.

## 6 Conclusions

We have presented a method for improving the energy efficiency of instruction fetch for embedded processors. Our approach is based on the selection of a "dense subset" of the instructions of a program, derived by evaluating their relative occurrence. The instructions in this subset are encoded with 8-bit patterns and stored in memory instead of the original (32-bit) instructions; in this way, memory bandwidth is reduced, and so is the energy required to execute the program. Program execution time may also get shorter as a side, yet positive effect.

The proposed scheme does not require any modification of the processor, since it always executes full-size instructions. This result is achieved by interposing an instruction decompressor between the processor and the instruction memory. We have introduced four hardware architectures for instruction decompression with different complexity and performance. To demonstrate the feasibility of our solution we have also implemented and simulated one of such architectures; for the experiment, a simplified version of the DLX processor has been used. Experimental results, obtained on a set of test programs indicate that good energy savings can be obtained. The cost in power and performance of the decompression block, as well as the tradeoffs involved in the selection of the size of the decompression table are currently under investigation.

## References


[1] E. Macii, M. Pedram, F. Somenzi, "High-Level Power Modeling, Estimation, and Optimization," IEEE TCAD, Vol. 17, No. 11, pp. 1061-1079, Nov. 1998.

[2] C. L. Su, C. Y. Tsui, A. M. Despain, "Saving Power in the Control Path of Embedded Processors," IEEE Design and Test, Vol. 11, No. 4, pp. 24-30, Winter 1994.

[3] M. R. Stan, W. P. Burleson, "Bus-Invert Coding for Low-Power I/O," IEEE TVLSI, Vol. 3, No. 1, pp. 49-58, Jan. 1995.

[4] L. Benini, G. De Micheli, E. Macii, D. Sciuto, C. Silvano, "Asymptotic Zero-Transition Activity Encoding for Address Busses in Low-Power Microprocessor-Based Systems," GLSVLSI-97, pp. 77-82, Mar. 1997.

[5] L. Benini, G. De Micheli, E. Macii, M. Poncino, S. Quer, "Reducing Power Consumption of Core-Based Systems By Address Bus Encoding," IEEE TVLSI, Vol. 6, No. 4, pp. 554-562, Dec. 1998.

[6] E. Musoll, T. Lang, J. Cortadella, "Working-Zone Encoding for Reducing the Energy in Microprocessor Address Buses," IEEE TVLSI, Vol. 6, No. 4, pp. 568-572, Dec. 1998.

[7] L. Benini, A. Macii, E. Macii, M. Poncino, R. Scarsi, "Synthesis of Low-Overhead Interfaces for Power-Efficient Communication over Wide Buses", DAC-36, Jun. 1999.

[8] S. Wuytack, F. Catthoor, L. Nachtergaele, H. De Man, "Global Communication and Memory Optimizing Transformations for Low Power Design," IWLPD-94 pp. 203-208, Apr. 1994.

[9] P. R. Panda, N. D. Dutt, "Reducing Address Bus Transitions for Low Power Memory Mapping," EDTC-96, pp. 63-67, Mar. 1996.

[10] P. R. Panda, N. D. Dutt, "Low Power Mapping of Behavioral Array to Multiple Memories," ISLPED-96, pp. 289-292, Aug. 1996.

[11] J. P. Diguet, S. Wuytack, F. Catthoor, H. De Man, "Formalized Methodology for Data Reuse Exploration in Hierarchical Memory Mappings," IEEE TVLSI, Vol. 6, No. 4, pp. 529-537, Dec. 1998.

[12] S. Segars, K. Clarke, L. Goudge, "Embedded Control Problems, Thumb and the ARM7TDMI," IEEE Micro, Vol. 15, No. 5, pp. 22-30, Oct. 1995.

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

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

[15] H. Lekatsas, W. Wolf, "Code Compression for Embedded Systems," DAC-35, pp. 516-521, Jun. 1998.

[16] L. Benini, G. De Micheli, A. Macii, E. Macii, M. Poncino, "Reducing Power Consumption of Dedicated Processors Through Instruction Set Encoding," GLSVLSI-98, pp. 8-12, Feb. 1998.

[17] J. L. Hennessy, D. A. Patterson, Computer Architecture - A Quantitative Approach, II Edition, Morgan Kaufmann Publ., 1996

[18] Mississippi State University, Micro-Systems Prototyping Lab., "The DLX Processor Core," http://WWW.ERC.MsState.Edu/mpl, 1998.