SPARC16: A new compression approach for the SPARC architecture

Leonardo Ecco, Bruno Lopes, Eduardo C. Xavier, Ricardo Pannain,
Paulo Centoducatte, Rodolfo Azevedo
Institute of Computing
University of Campinas - UNICAMP
Av. Albert Einstein, 1251, Campinas, Brazil
leonardo.ecco@students.ic.unicamp.br, {bruno.lopes,ecx,pannain,ducatte,rodolfo}@ic.unicamp.br

Abstract

RISC processors can be used to face the ever increasing demand for performance required by embedded systems. Nevertheless, this solution comes with the cost of poor code density. Alternative encodings for instruction sets, such as MIPS16 and Thumb, represent an effective approach to deal with this drawback. This article proposes to apply a new encoding to the SPARCv8 architecture. Through extensive analysis of a program mix from the Mibench and Mediabench benchmark suites, we suggest a new 16-bit instruction set, easily translated to its 32-bit counterpart during execution time. Using the aforementioned program mix to infer how code could be represented in the proposed 16-bit ISA, compression ratios as low as 56% can be obtained. We also evaluated the cache behavior and showed reductions of 42% on cache misses that can increase performance up to 28% (for patricia program with 2KB cache).

1 Introduction

Recent works in the Code Compression area have proved that code size reduction is not only a matter of reducing the program footprint on memory but also a very effective way to improve the program performance while reducing the total power on a system [4, 8, 18]. Going further, it is not only necessary to have good algorithms to encode the program instructions but also a very good decompression system to allow all these gains together.

The Code Compression problem can be stated as the search for an alternative representation to the program instructions that reduces the memory usage and can be done both in software and hardware [3], usually with different goals. While the software approach focuses on the maximum reduction on the program size, at the possible cost of program performance since the decompression will be done in software, the hardware approach focuses on decompression speed, usually at the cost of code size reduction. There are, basically, two different approaches for hardware decompression, one that compresses instructions or blocks of the program in an ad-hoc manner, and another that tries to create an alternative encoding for the instructions in a smaller size, like representing the 32-bit instructions in a well-defined 16-bit format. This paper focuses on the latter approach to find a new encoding to the SPARC ISA.

The SPARC ISA was selected after an analysis of 15 variations of 7 different ISAs and represent a good trade-off between opportunity to compress (big code size) and impact of the result (the SPARC ISA still has a good user base nowadays).

In this work, we will use the term **Compression Ratio** to represent the code size reduction. Equation 1 shows how the Compression Ratio should be calculated. It is important to notice that, in this case, lower means better. A Compression Ratio of 56% means that the program is reduced to 56%. In this number, all the overhead should be included. Unfortunately, not all related work in the area consider the overhead in this number.

$$Compression\ Ratio = \frac{Compressed\ Size + Overhead}{Original\ Size} \quad (1)$$

We used the MediaBench and MiBench benchmarks to analyze the instruction occurrences and design the SPARC16 encoding. The results were evaluated using the same programs from MediaBench (average 58.1%, better 56.4%), MiBench (average 57.6%, better 56.4%), and with the Linux Kernel (64.4%) and binaries (64.2%). We also evaluated the instruction cache behavior of these programs and found that it is possible to reduce the miss rate in 42% which can improve the performance (28% on the instruction cache in the patricia program) and reduce the power consumption.

This paper is organized as follows: Section 2 describes

the previous works on this area. Section 3 shows an evaluation of current ISAs focusing on opportunities for code compression and justifies the selected ISA. Section 4 shows an Integer Linear Programming (ILP) model used to create a 16-bit encoding for the SPARCv8 ISA followed by Section 5 that describes the resulting encoding. Section 6 shows the results of our architectural evaluation and Section 7 draws some conclusions.

2 Related Work

Although Code Compression is not a new research area [24], some of the goals of recent projects are. In the beginning, researchers focused only on code size reduction because memory cost and size were the main system restriction. Recent works on the field showed that, if well designed, the decompression hardware can hide memory hierarchy latencies and improve the system performance while reducing power consumption [9, 10, 16].

The *DLX* architecture, created by Hennessy and Patterson [19], was the first 32 bits architecture to have a 16 bit extension. The extension, called D16, had instructions with 2 registers as operands (the original had 3) and also smaller immediate field. This configuration allowed a 62% compression ratio and 5% performance loss [7].

First presented by ARM on its ARM7 model, the next 16 bits processor extension in the market was *Thumb* [1]. Thumb enabled ARM processors are capable of running code in both 32 and 16 bits modes and allow subroutines of both types to share the same address space, while the mode exchange is achieved during runtime through **BX** and **BLX** instructions, which are branch and call instructions that flip the current mode bit in a special processor register. To fit functionality in 16 bits, a group of only 8 registers together with a stack pointer and link registers are visible, the remaining registers can only be accessed implicitly or through special instructions. Results presented by ARM show a compression ratio ranging from 55% to 70%, with an overall performance gain of 30% for 16 bit buses and 10% loss for 32 bit ones. *Thumb2* is the recent version of the original *Thumb* incremented with new features like the addition of specific instructions for operating system usage.

The MIPS16 [15] instruction set is the MIPS processor 16-bit extension, it contains the same capability of exchanging between modes (using the **JALX** instruction) and sharing address space. There is a reduction from 16 to 5 bits on the immediate size, and only 8 registers, out of 32 on the traditional MIPS, are visible. Special move instructions can be used to move data between the visible and hidden registers, while some other instructions can also use hidden registers implicitly. New features introduced by MIPS16 include the *EXTEND* instruction, which has only an opcode and an immediate field that is used to extend the immediate of the following instruction. MIPS16 also has PC and SP relative addressing that can be used to efficiently load constants and load/store instructions whose computed effective addresses are shifted to match type alignment. This reduced encoding achieved a 60% compression ratio according to MIPS technologies [15].

[2, 9, 14, 16, 22] focus their work in Code Compression using dictionaries. The use of a dictionary to compress code tends to reduce energy consumption and to improve the performance with a reasonable static compression ratio. Using bitmask and prefix based Huffman encoding, [14] have improved their compression ratio in 9–20%. [16] have used the variable-length ISA RISC processor CR16C (National Semiconductor Inc.) with two approaches: using a bit-vector and using a reserved instruction to identify code words. An additional logic to decompress the code on-the-fly was designed for each of these approaches. The goal of this work is to use the code compression to improve performance. The results have demonstrated an speed-up of up to 15%, achieving also a reduction in code size (up to 30%) and bus-switching activity (up to 20%). The reduction of the bus-switching activity also reduced the energy consumption.

[5, 6, 10, 20] have used a hardware support approach to optimize the code compression/decompression. [5, 6] achieved compressions ratios as low as 56%. In both works they have used a set of applications applied to ARM and MIPS architectures, they also used PowerPC in the first work. [20] implemented a fast parallel decompression and improved the decode bandwidth up to four times, with minor impact (less than 1%) on compression efficiency. The experimental results of [10] showed reductions in code size (up to 35%) and energy consumption (up to 10%) and improvements in performance (up to 20%).

3 Instruction Set Evaluations

In order to select the base architecture for our experiments, we evaluated 15 variations of 7 different ISAs regarding to code size for the same set of programs, the SPEC 2006 benchmark. To evaluate them, we used gcc built specifically for each architecture variation and the same global gcc options to allow a fair comparison. By global options we mean all the options that are not architecture specific. From the architecture specific options, we used the ones that generate the smaller code size.

Figure 1 shows the total program size for each architecture variation supported by gcc sorted in ascending order. We represented the program sizes in three different ways and normalized the results in regard to the smaller one value for each category. In the graph, **AM** means Arithmetic Mean of all programs for each architecture, **GM** means Geometric Mean, and **TT** means Total Size. Both **AM** and **TT**

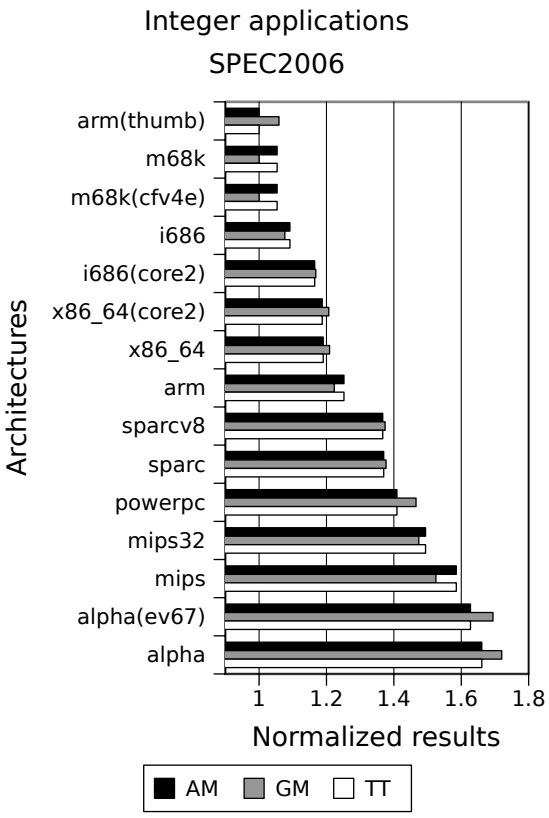


Figure 1. SPEC 2006 program sizes compiled with the same gcc global options

were normalized to ARM (Thumb) and **GM** was normalized to M68K.

The best candidate for our compression experiment is an architecture that is still being used nowadays and that has a low code density (large code size). Going from the architecture with lower to the higher density, we have:

1. **Alpha:** Both EV6 and EV7 were evaluated and have the lowest code density. Based on that, they would be the best choice but this architecture has been discontinued.

2. **MIPS:** Both MIPS and MIPS32 are the next architecture with the lower code density. But the MIPS architecture has already a compact ISA called MIPS16. Unfortunately, the gcc provided with this architecture is not capable of compiling the SPEC2006 benchmark (issues in the standard C library newlib which is focused on embedded systems). From the MIPS16 documentation [15], the compression ratio goes around 60% which places MIPS16 in the group of higher code density.

3. **PowerPC:** The PowerPC architecture also has an extension to compression, codenamed CodePack [12], so it goes to the same set as MIPS16 and Thumb.

4. **SPARC:** The SPARC and SPARCv8 [23] architecture are the next in the lower code density rank, with code near than 40% bigger than the higher density architectures. The SPARC instruction set is the IEEE 1754 standard, which made a market with several different makers around the world (more than 50 registered members in SPARC International nowadays). The SPARC architecture does not have a compression mechanism and it has been used for a long time in several academic works as a testbed for compression algorithms.

5. **ARM:** The ARM architecture has the Thumb and Thumb2 extensions which makes it the higher code density ISA we evaluated.

6. **i686 and x86_64:** Intel architecture 32-bit and 64-bit versions. Although it is not the focus of this paper, notice that the code size keeps growing with the architecture evolution. This is because newer instructions (ISA extensions) need more bits to be represented in the old ISA. There is, on average, a 15% increase in the code size for the same program from i686 to x86_64. It is important to mention that we are not comparing performance or any other features here, it is only a code size analysis for now.

7. **m68k:** The Motorola 68000 family is a highly encoded instruction set, the smallest considering the GM values.

Based on the the previous comments, we selected the SPARCv8 architecture since it represents a very good compromise between potential gain in code size reduction and current ISA usage. To reduce the SPARCv8 ISA to 16-bits, we need to analyze how the ISA fields are used. The most important fields to reduce are the immediates since they can occupy more than half instruction when used.

Figure 2 shows the accumulated number of bits needed to encode the immediate in all arithmetic, logic, shift, load and store instructions for the MiBench Benchmark. Notice that more than 80% of the arithmetic instructions require 6 or less bits. Also, near 80% of the load and store instructions can be represented with 9 or less immediate bits. We got similar results from the MediaBench Benchmark.

Based on the analysis of the ISA, we developed an Integer Linear Programming model (ILP) to represent the problem and solved it to find the best field sizes for every instruction, defining a new set of formats. The model will be described in the next section.

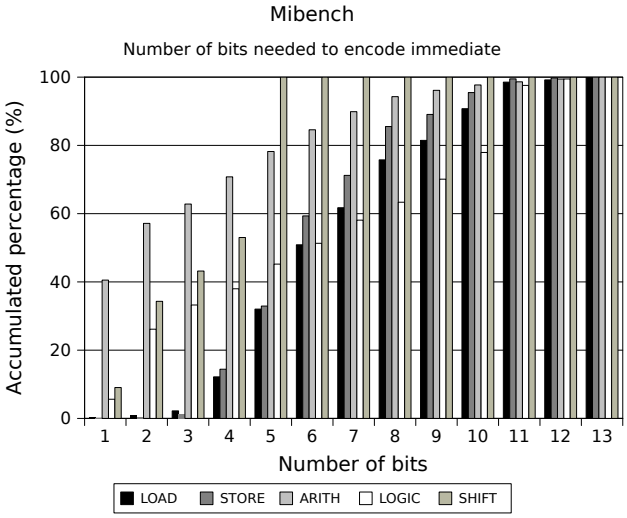


Figure 2. Immediate sizes for load, store, arithmetic, logical and shift instructions

4 An Integer Linear Programming Model to Solve the Problem

An input instance to the ILP model mentioned in section 3 consists of a tuple (S, F, I, c). Each element $s \in S$ is a set of fields. A field can be a primary opcode, a secondary opcode, a register or an immediate. The primary opcode is mandatory for every $s \in S$. Secondary opcodes and immediates are optional, but limited to one per $s \in S$. Registers are also optional, but each $s \in S$ can have up to three.

An attribution of size to each of the fields of an element $s \in S$ corresponds to a format. Formats obey two rules: the sum of sizes of each of its fields equals sixteen and the size of a register field is always three. For each $s \in S$, we generated every possible format obeying the aforementioned rules and placed them into the set F.

The set I contains SPARC16 candidate instructions. We took as candidates SPARCv8 instructions and pseudo-instructions. Pseudo-instructions were included because we wanted to hide the existence of certain registers, such as %g0, %sp, %fp and %ra, thus enabling us to not only mitigate the impact of having only three bits to index the register bank, but also to increase the size of immediate fields, since the pseudo-instructions reference registers implicitly.

The cost function $c : I \times F \rightarrow \mathbb{I}$ specifies the cost of mapping $i \in I$ to format $f \in F$. Certain mappings are invalid, for instance, associating an instruction that needs an immediate to a format that does not have an immediate field. The c function disregards those. The costs were calculated using the programs from Mediabench [17] and Mibench [13]. For every instruction in those programs, we identified an equivalent in I, and attributed the cost of associating it to every valid format taking into account factors such as the immediate field size being big enough to accommodate constants and attempts to represent 3-addresses instructions as 2-addresses instructions (i.e., forcing a register to work simultaneously as source and destination of an operation).

In order to allow the resolution of the ILP model to discard candidate instructions, the special format 0 was created. The cost of associating $i \in I$ to 0 represents the cost of not supporting i in SPARC16. We calculated that by estimating the number of supported SPARC16 instructions that would have to be executed to achieve the same effect as i. Obviously, this is an speculative value, since the SPARC16 ISA is yet to be determined.

That being said, the problem is to map every instruction to a single format minimizing the total cost incurred in this mapping. The mapping is subject to several constraints that guarantee that the obtained mapping makes sense — meaning that the primary opcode has the same number of bits in every chosen format, and that the number of bits attributed to the opcode fields affects the number of instructions that can be placed in a format. Below we describe the ILP model.

We have binary variables x_{if} that indicate if instruction i is going to be mapped to format f or not. We also have binary variables y_f that specify if the format f is going to be used. A format can have at most two opcode fields (a primary and a secondary). There are at most K opcode fields identified by $OP_1, OP2, \ldots, OP_K$. The primary opcode (OP_1) is shared amongst every $s \in S$, while the secondary opcodes ($OP_2, \ldots, OP_K$) are exclusive. Each opcode has at most L bits. There are binary variables op_{kl} indicating that the k-th opcode uses l bits. There is a special integer variable T that specifies the number of primary opcode available slots. For each $k \in \{2, \ldots, K\}$ and $l \in \{1, \ldots, L\}$ we create integer variables G_{kl} that state the number of primary opcode slots, among the total T, that will be occupied by instructions mapped to a format that has the k-th opcode with l bits as its secondary opcode. Notice that we can map at most $2^l G_{kl}$ instructions to the format in question. The integer variable $G1$ states the number of primary opcode slots occupied by instructions mapped to a format that has no secondary opcode. We use $f \in s$ to denote that a format $f \in F$ was generated from a set $s \in S$. We use $i \in f$ to denote that instruction $i \in I$ can be mapped to format $f \in F$ and we call the set of formats to which i can be mapped F_i.

$$\text{Min} \sum_{i \in I} \sum_{f \in F_i} c(i, f) x_{if}$$

Subject to

$$\sum_{f \in s} y_f \leq 1, \text{ for } s \in S \quad (1)$$
$$x_{if} - y_f \leq 0, \text{ for } i \in I \text{ and } f \in F_i \quad (2)$$
$$\sum_{f \in F_i} x_{if} = 1, \text{ for } i \in I \quad (3)$$
$$y_f + y_{f'} \leq 1, \text{ for } f, f' \in F \text{ inconsistent} \quad (4)$$
$$\sum_{l \leq L} op_{kl} = 1, \text{ for } k \leq K \quad (5)$$
$$y_f - op_{kl} \leq 0, \text{ for each } k \text{ and } l, f \text{ has } op_{kl} \quad (6)$$
$$T - \sum_{l \leq L} 2^l op_{1l} \leq 0 \quad (7)$$
$$G1 + \sum_{k=2}^{K} \sum_{l \leq L} G_{kl} - T \leq 0 \quad (8)$$
$$G_{kl} - 2^L op_{kl} \leq 0, \text{ for each } k \text{ and } l \quad (9)$$
$$\sum_{(i,f) \in G1} x_{if} - G1 \leq 0 \quad (10)$$
$$\sum_{(i,f) \in G_{kl}} x_{if} - 2^l G_{kl} \leq 0, \text{ for each } k \text{ and } l \quad (11)$$

Constraint (1) assures that at most one format $f \in F$ of each set $s \in S$ is chosen. Constraint (2) establishes that an instruction is assigned to a format only if the format is chosen. Constraint (3) guarantees that each instruction is assigned to a format (remember that every instruction can be mapped to format 0). Constraint (4) assures that inconsistent formats, i.e. formats that attributed different sizes to a field in common, cannot be used at the same time. For example, this guarantees that all the chosen formats will have the same number of bits dedicated to the primary opcode (since OP_1 is shared amongst all formats). Constraint (5) establishes that each opcode field will have a fixed size of l of bits. Constraint (6) says that one format with the k-th opcode using l bits, can be used only if the solution uses that opcode with l bits. Constraint (7) calculates the total amount T of slots for the primary opcode. Constraint (8) guarantees that the number of slots of the primary opcode that are spread among the groups is at most T. Constraint (9) assures that group G_{kl} is going to be used only if in the solution, opcode k uses l bits. Constraints (10) and (11) limit the number of instructions that can be assigned to each format. For these constraints, $(i, f) \in G1$ refers to instructions $i \in I$ mapped to a format $f \in F$ with no secondary opcode, while $(i, f) \in G_{kl}$ refers to instructions $i \in I$ mapped to a format that has k with l bits as its secondary opcode.

5 The SPARC16 ISA

SPARC16 is an extension to the SPARCv8 instruction set and it relies on a regular SPARCv8 pipeline in order to work. The SPARC16 instructions are simple enough to be translated to their 32-bit counterparts during execution time. The translation, per se, is accomplished by placing a decompressor between the instruction cache and the SPARCv8 pipeline, as shown in figure 3. This is similar to the scheme adopted by Thumb [1] and MIPS16 [15].

Table 1 illustrates the formats in which SPARC16 instructions are encoded. Every format is identified by a 5-bit major opcode. Some formats also hold a secondary opcode. Formats come in different shapes and sizes in order to fulfill specific instruction needs. For instance, I format is used to

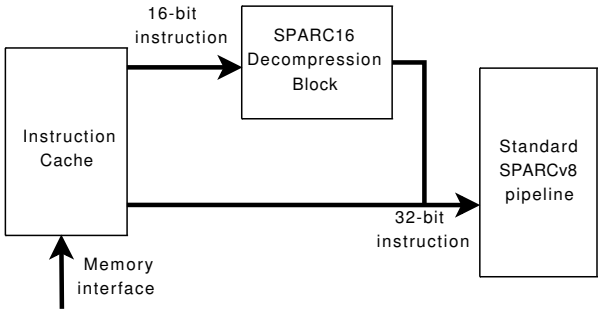


Figure 3. SPARC16 decompression diagram

accommodate a *call and link* instruction, which requires a large immediate field, but no registers.

On the other hand, RRR format has no immediate field and is used to encode instructions that operate on three registers, such as an *add*, which sums the values held by two source registers and stores the result in a third register. The RR format is also available and is used to encode SPARC16 instructions that operate on two registers only, forcing one of the registers to act as source and destination of the operation. Although this may look like a disadvantage, we can use the bits, otherwise used to encode the third register, in a more convenient way, such as encoding a larger set of instructions through the use of a secondary opcode.

SPARC16 uses a subset of the SPARCv8 registers. From the 32 SPARCv8 registers, 8 are visible and can be explicitly referenced by SPARC16. To access hidden registers, two special instructions are provided — *MOV8to32* and *MOV32to8*. The former moves data from a visible register to a hidden register, the latter performs the opposite operation. The I2 format is used to represent these instructions by breaking the eight bit immediate into a three bit field, used to index a SPARC16 visible register, and a five bit field, used to index one of the 32 registers from the SPARCv8 register bank. These instructions could be used to spill data into the hidden registers avoiding the emission of memory operations during register allocation.

Beyond being able to explicitly index registers, some instructions in SPARC16 include implicit access to registers %sp, %fp, %g0 and %ra. This presents two main advantages. Firstly, it allows us to mitigate the impact of having a smaller set of visible registers – only 8, since we use 3 bits to index the register bank in SPARC16. If the aforementioned registers were visible, we would only have 4 registers left to work with. Secondly, implicitly referencing a register means 3 free bits that can be used to encode a larger immediate. As an example, we can list *ldfp*, which is a *load* instruction, encoded in the RI format, where the *frame pointer* is used implicitly as a source register.

Large constants are handled using an auxiliary instruction similar to the MIPS16 *EXTEND* one. Any instruction

Table 1. SPARC16 formats

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0				
I-type op	immediate			
I2-type op	op2	immediate		
RI-type op	immediate			reg
RI2-type op	op2	immediate		reg
RRI-type op	immediate		reg	reg
RRI2-type op	op2	imm	reg	reg
RR-type op	op2		reg	reg
RRR-type op	op2	reg	reg	reg

Table 2. Mediabench compression rates

Program	Comp. rate	Program	Comp. rate
cjpeg	56.8%	rawcaudio	56.6%
djpeg	57.8%	rawdaudio	56.6%
mpeg2dec	59.9%	timing	56.4%
mpeg2enc	59.9%	toast	58.9%
pegwit	58.7%	untoast	58.9%

with the need of larger constants can be extended using some bits from the extend instruction. This mechanism is also applied for providing additional registers to some instructions, allowing some operations from the SPARCv8 not represented in SPARC16 to take place with only a 16 bit overhead, instead of a mode exchange. Neither MIPS16 nor Thumb used the extend instruction to encode the third register for an instruction.

Alternating between execution modes can be accomplished by the *jmpl* instruction, available in SPARCv8 and SPARC16. The least significant bit in the 32-bit target address determines the target routine's mode — 0 means SPARCv8 and 1 means SPARC16.

Although the instruction allocation came from an ILP model, SPARC16 was designed to leave some encoding space for future ISA extensions – as a matter of fact, the *MOV8to32*, *MOV32to8* and *EXTEND* instructions were added after we obtained the formats using the ILP algorithm described in section 4. The bit encoding for each new instruction will be made trying to simplify the conversion between SPARC16 and SPARCv8. The encoding is not expected to reduce the processor clock by two reasons: (1) The critical path in the SPARCv8 processor, that we are going to use, is not in the Instruction Decode stage, and (2) The bigger decode lookup table to be used will have 32 lines (5-bit address).

6 Experimental Results

In order to estimate the compression rate achieved by the proposed ISA, we used the SPARCv8 code as a starting point. Our approach consisted in checking each instruction of a SPARCv8 program and mapping them to the corresponding SPARC16 instruction. For instance, an SPARCv8 *add %reg,imm,%reg* instruction can be represented by its 16-bit counterpart, which is encoded in the RRI format. The difference between the two instructions is that the former has 13 bits to encode the immediate while the latter has only 5. Therefore, when we analyze an SPARCv8 *add %reg,imm,%reg* instruction, we first check if the immediate can be accommodated in a 5-bit field. If so, we account one SPARC16 instruction (2 bytes). If not, we know we have to use the *EXTEND* mechanism, mentioned in Section 5, which implies we will need 4 bytes (2 bytes from the *EXTEND* plus the 2 bytes of the *add %reg,imm,%reg* instruction). We have used a similar approach when converting instructions from 3 registers to 2 registers. In both cases, we did not consider the costs involved in the possible spill code that can be generated by having only 8 visible registers. We expect that this cost will not make a big impact our results because the compiler can try to avoid it and it is still possible to access the other registers using the MOV8to32 and MOV32to8 instructions.

Table 3. Mibench compression rates

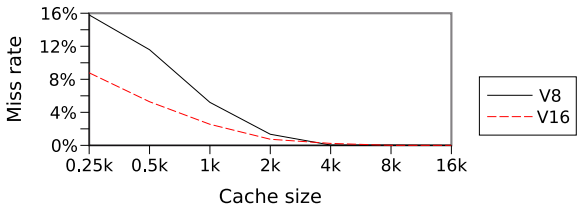
Program	Comp. rate	Program	Comp. rate
bmath_large	58.3%	qsort_small	56.7%
bmath_small	58.3%	rawcaudio	56.6%
bitcnts	56.8%	rawdaudio	56.6%
cjpeg	56.8%	rijndael	58.7%
crc_32	56.7%	search_large	56.7%
dijkstra_large	57.0%	search_small	56.7%
dijkstra_small	57.0%	sha	56.7%
djpeg	57.8%	susan	57.0%
fft	57.2%	timing	56.4%
lame	63.9%	toast	58.9%
patricia	57.2%	untoast	58.9%
qsort_large	57.1%		

Table 4. Linux compression rates

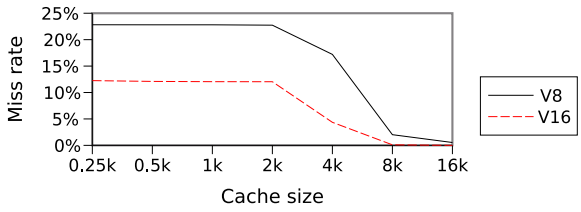
Program	Comp. rate
Kernel	64.4%
/bin programs	64.2%

Using the aforementioned method, we estimated the compression ratio for MediaBench, MiBench and Linux. Results are presented in Tables 2, 3 and 4, respectively. Since the MediaBench and MiBench files were used to create the model, we also used other set of files to check the compression ratio: the Linux kernel and its */bin* programs.

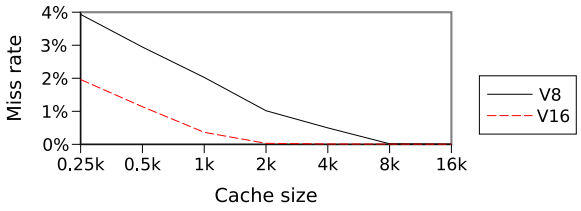
On average, the compression ratio for the MediaBench programs (table 2) is 58.1%, the MiBench (table 3 is 57.6%,



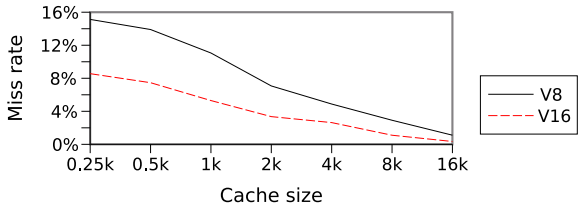
(a) qsort



(b) rijndael



(c) dijkstra



(d) patricia

Figure 4. SPARC16 vs SPARCv8 miss rate evaluation

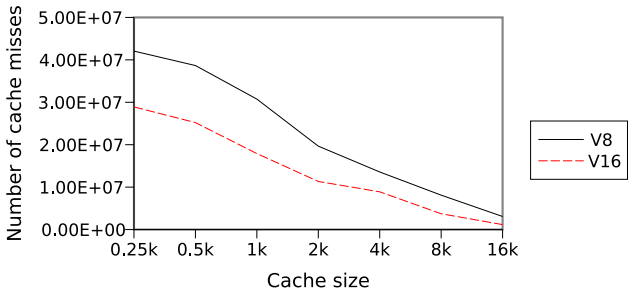


Figure 5. Absolute number of cache misses for the patricia program

and the Linux kernel and binaries average is 64.3%. The difference in the Linux programs comes from the usage of privileged instructions and immediates that are bigger on average. Even with this numbers, we still have a good code size reduction.

We also estimated the impact that SPARC16 should have on reducing cache misses. We collected an execution trace from the SPARCv8 ArchC [21] model for four different MiBench programs: qsort, rijndael, dijkstra and patricia. Using the previously explained approach to map the SPARCv8 code to SPARC16 instructions, we obtained SPARC16 traces from the SPARCv8 ones.

We analyzed the traces using Dinero IV [11]. We tested different sizes for a 2-way, 16 bytes per line cache. Results are presented in Figures 4(a), 4(b), 4(c) and 4(d).

It is not fair to compare miss rates, since the instruction count is higher for SPARC16 than SPARCv8. For instance, for the patricia program, SPARCv8 performed 278,059,318 instruction fetches while SPARC16 performed 337,833,652 – approx. 21.4% more. Nevertheless, due to a smaller instruction size, SPARC16 achieves a smaller absolute number of cache misses for every tested cache size, as shown by figure 5. Similar patterns were observed for the other programs.

Now let's have a closer look at how performance can be affected by a reduction in miss rates. Consider a 2-way, 16 bytes per line, 2KB cache. With this cache configuration and running the patricia program, the absolute number of cache misses for SPARCv8 and SPARC16 are 19,664,314 and 11,338,999, respectively. Table 5 shows how the cache miss penalty affects the number of cycles spent on instruction fetching. For example, considering a memory latency of 10 cycles, the SPARC16 uses 3% ($= 1 - 0.97$) less time in instruction fetch than SPARCv8. This value is increased to 28% for a memory latency of 50 cycles.

Table 5. Cycles spent on instruction fetches

Miss penalty (cycles)	10	20	30	40	50
SPARC16/SPARCv8	0.97	0.85	0.79	0.75	0.72

7 Conclusions

We have presented the SPARC16 instruction set extension for the SPARCv8 processor. We first evaluated 7 ISAs to find a good candidate for code compression. Then we

analyzed all the instructions and formats usage. To find the best possible encoding, we used an Integer Linear Programming method and selected the best formats to minimize the code size.

In order to not have a biased model, we designed it based on the MiBench and MediaBench benchmarks and evaluated it with MiBench, MediaBench, Linux kernel and some Linux programs (`/bin` files). As result, we got an average compression ratio of 58.1% for MediaBench programs, 57.6% for MiBench and 64.3% for the Linux programs. Besides that, we also analyzed the instruction-cache behavior after using our encoding. As expected, a SPARC16 program needs more instructions to execute than the SPARCv8 counterpart, but SPARC16 instructions are smaller, and when using the same cache configuration it takes less time to fetch all the program instructions to SPARC16 than SPARCv8. For the patricia program, fetching all instructions to the processor is up to 28% faster in SPARC16 as the result of 42% less cache misses for a 2KB cache. As shown in previous work, this will certainly reflect in the energy consumption.

8 Acknowledgments

Our thanks to FAPESP, CAPES and CNPq for supporting this work.

References

[1] ARM. *An Introduction to Thumb*. Advanced RISC Machines Ltd., Mar. 1995.

[2] N. Aslam, M. Milward, I. Nousias, T. Arslan, and A. Erdogan. Code compression and decompression for instruction cell based reconfigurable systems. pages 1–7, March 2007.

[3] A. Beszédes, R. Ferenc, T. Gyimóthy, A. Dolenc, and K. Karsisto. Survey of code-size reduction methods. *ACM Comput. Surv.*, 35(3):223–267, 2003.

[4] E. Billo, R. Azevedo, G. Araujo, P. Centoducatte, and E. W. Netto. Design of a decompressor engine on a sparc processor. In *SBCCI '05: Proceedings of the 18th annual symposium on Integrated circuits and system design*, pages 110–114, New York, NY, USA, 2005. ACM.

[5] T. Bonny and J. Henkel. Efficient code density through look-up table compression. *Design, Automation and Test in Europe Conference and Exhibition*, 0:151, 2007.

[6] T. Bonny and J. Henkel. Instruction re-encoding facilitating dense embedded code. *Design, Automation and Test in Europe Conference and Exhibition*, 0:770–775, 2008.

[7] J. Bunda, D. Fussell, W. C. Athas, and R. Jenevein. 16-bit vs. 32-bit instructions for pipelined microprocessors. *SIGARCH Comput. Archit. News*, 21(2):237–246, 1993.

[8] X. Chen, L. Yang, H. Lekatsas, R. P. Dick, and L. Shang. Design and implementation of a high-performance microprocessor cache compression algorithm. *Data Compression Conference*, 0:43–52, 2008.

[9] M. Collin and M. Brorsson. Two-level dictionary code compression: A new scheme to improve instruction code density of embedded applications. *Code Generation and Optimization, IEEE/ACM International Symposium on*, 0:231–242, 2009.

[10] M. L. Corliss, E. C. Lewis, and A. Roth. The implementation and evaluation of dynamic code decompression using dise. *ACM Trans. Embed. Comput. Syst.*, 4(1):38–72, 2005.

[11] J. Edler and M. Hill. Dinero iv trace-driven uniprocessor cache simulator, online at http://www.cs.wisc.edu/ markhill/dineroiv/, 2003.

[12] M. Game and A. Booker. *CodePack: Code Compression for PowerPC Processors*. International Business Machines (IBM) Corporation, 1998.

[13] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. Mibench: A free, commercially representative embedded benchmark suite. *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3–14, Dec. 2001.

[14] S. I. Haider and L. Nazhandali. A hybrid code compression technique using bitmask and prefix encoding with enhanced dictionary selection. In *CASES '07: Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 58–62, New York, NY, USA, 2007. ACM.

[15] K. Kissell. *MIPS16: High-density MIPS for the Embedded Market*. Silicon Graphics MIPS Group, 1997.

[16] R. Kumar and D. Das. Code compression for performance enhancement of variable-length embedded processors. *ACM Trans. Embed. Comput. Syst.*, 7(3):1–36, 2008.

[17] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 330–335, Washington, DC, USA, 1997. IEEE Computer Society.

[18] E. W. Netto, R. Azevedo, P. Centoducatte, and G. Araujo. Multi-profile based code compression. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 244–249, New York, NY, USA, 2004. ACM.

[19] D. A. Patterson and J. L. Hennessy. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[20] X. Qin and P. Mishra. Efficient placement of compressed code for parallel decompression. *VLSI Design, International Conference on*, 0:335–340, 2009.

[21] S. Rigo, G. Araujo, M. Bartholomeu, and R. Azevedo. Archc: a systemc-based architecture description language. pages 66–73, Oct. 2004.

[22] S.-W. Seong and null Prabhat Mishra. An efficient code compression technique using application-aware bitmask and dictionary selection methods. *Design, Automation and Test in Europe Conference and Exhibition*, 0:112, 2007.

[23] C. SPARC International, Inc. *The SPARC architecture manual: version 8*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[24] W. T. Wilner. Burroughs b1700 memory utilization. In *AFIPS '72 (Fall, part I): Proceedings of the December 5-7, 1972, fall joint computer conference, part I*, pages 579–586, New York, NY, USA, 1972. ACM.