Design and evaluation of compact ISA extensions

Bruno Lopes[*], Leonardo Ecco[†], Eduardo C. Xavier[*], Rodolfo Azevedo[*]

University of Campinas, IC, Brazil[*]

{blopes,ecx,rodolfo}@ic.unicamp.br

Technische Universität Braunschweig, Germany[†]

ecco@ida.ing.tu-bs.de

Abstract

The modern embedded market massively relies on RISC processors. The code density of such processors directly affects memory usage, an expensive resource. Solutions to mitigate this issue include code compression techniques and ISAs extensions with reduced instructions bit-width, such as Thumb2 and MicroMIPS. This paper proposes a 16-bit extension to the SPARC processor, the SPARC16. Additionally, we provide the first methodology for generating 16-bit ISAs and evaluate compression among different 16-bit extensions. SPARC16 programs can achieve better compression ratios than other extensions, attaining results as low as 67%. Moreover, SPARC16 reduces cache miss rates up to 9%, requiring smaller caches than SPARC processors to achieve the same performance; a cache size reduction that can reach a factor of 16.

1 Introduction

Code Compression is the alternative representation to program instructions that reduces memory usage and can be implemented both in software and hardware [7]. In a solely software approach, the goal is to reduce program size to the fullest, with an associated performance degradation cost during decompression. The hardware approach focuses on decompression speed, usually at the cost of code size reduction. There are two different approaches for hardware decompression, one that compresses instructions or blocks of the program in an ad-hoc manner and other that tries to create an alternative encoding for the instructions in a smaller size: reducing a 32-bit ISA to a well-defined 16-bit format. This paper focuses on the latter approach to find a new encoding to the SPARCv8 ISA [48].

16-bit ISA extensions exists for RISC architectures such as MIPS, PowerPC and ARM, more details in Section 8. However, existing implementations are commercial products [3, 11, 16, 23, 29, 32, 43] and no published research presents how such extensions are designed. This paper shows how 16-bit extensions can be designed based on a specific case study: the creation of a 16-bit extension to the SPARC ISA, the SPARC16 [18]. The method includes an extensive static and dynamic analysis of several benchmarks and binaries to find intrinsic ISA compression inefficiencies and opportunities. Also, an *Integer Linear Programming (ILP)* model optimally assists in the creation of formats and fields encodings for the new 16-bit extension. A general method can be abstracted from our case study and further applied to other architectures.

Our compiler toolchain is based on LLVM [37] and Binutils [24] Open Source projects and provides SPARC16 support in the Frontend, Backend, Assembler and Linker. The linker is capable of linking object files from SPARCv8 and SPARC16 altogether, allowing usage of existing SPARCv8 libraries. Target specific optimizations enhance final SPARC16 code compressibility by approaching code generation specially - intrinsic 16-bit

instructions properties are considered.

The SPARC16 instructions are translated to their 32-bit counterparts during execution time by an additional hardware decompressor placed between the processor and the instruction cache - we implemented the mechanism in the SPARCv8 compliant processor Leon3. A SPARC16 emulator is capable of collecting SPARC16 dynamic execution traces, the input of a cache simulator which evaluates instruction cache usage.

Finally, we evaluate compression and performance results for SPARC16 and compare the results with other related work. We also present compression ratio results achieved for SPARC16 programs, which can be lower than most popular 16-bit extensions such as MicroMIPS and Thumb2.

The main contributions of this paper are:

- A method for designing 16-bit ISA extensions from 32-bit ISAs.
- A 16-bit extension for the SPARC architecture.
- Compression ratio evaluation across distinct 16-bit extensions.

This paper is organized as follows: Section 2 focus on the opportunities for code compression among several architectures, explaining why SPARC is our selected ISA. Section 3 evaluates the SPARCv8 ISA according to several code analysis for distinct programs, both static and dynamic. The Section 4 describes the Integer Linear Programming (ILP) model, which assists in the definition of SPARC16 instruction formats; the resulting encoding is then specified in Section 5. Details of the hardware implementation, emulators, compiler toolchain support and optimizations are covered in Section 6. The SPARC16 compression ratios and performance results are presented in Section 7. Finally, Section 8 describes the related work on the area and we finish the paper in Section 9, drawing our conclusions.

2 The SPARC Compression Potential

In order to apply and construct a method for generating a 16-bit instructions set, we need to choose an architecture for evaluation and case study. Hence, we need to find an architecture with good compression opportunities, otherwise there is no need for a 16-bit extension.

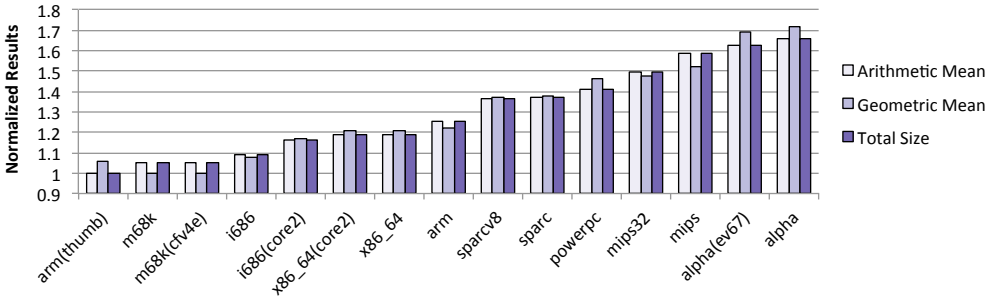


Figure 1: Overall SPEC CINT2006 program sizes for several architectures

We evaluated code size behavior of 15 ISAs using the SPEC CINT2006 benchmark [27]. A GCC [49] cross compiler toolchain is used[1] for each architecture and the same code size optimization flag *-Os* applied across all configurations. We also used *-mcpu* option to generate code for specific ISA extensions when needed. Figure 1 shows the code size evaluation for all tested architectures. Results are presented by total size, arithmetic and geometric means for each architecture. The values are normalized against the smallest one within each category.

[1]Since some architectures are very old, they need older GCC versions and hence the same version is not used across all compilers

SPEC CINT2006 results for *X86*, *X86_64*, *M68K* and *Thumb* have high code density and are very compressed. *Mips*, *ARM* and *PowerPC* already have 16-bit extensions and are not eligible. *Alpha* EV6 and EV7 have the lowest measured code density but disqualified because the architecture is discontinued and available toolchains and libraries are obsolete. Therefore, we choose the SPARC architecture, which has several advantages:

- Low code density, 40% bigger than higher density architectures.
- The SPARC ISA is a IEEE 1754 standard and still widely used – 50 registered members in SPARC International nowadays.
- No previous 16-bit compression mechanism is available for SPARC.
- Used in many academic works as a testbed for compression algorithms.
- Recent and stable software support: operating system kernel, libraries and compilers.

3 Instruction Set Evaluations

To create a new 16-bit extension to the SPARCv8 ISA, we need an extensive analysis of how instructions are used in existing compiled SPARCv8 applications.

ISA Usage. The first task in the design of a 16-bit extension is to define which instructions or functionalities from the regular 32-bit ISA must be present in the extension. The main concern is to be representative enough to effectively achieve compression ratios similar to the ones in the literature. To find potential instructions for the 16-bit extension, we organize our analysis as follows:

- Instructions never used by any analyzed program are removed from our selection pool unless a constraint exists. For example, short functions may contain specific instructions which are executed very often but have a low static count. Performance degradation may arise from issuing mode exchange to execute such instructions when there is no 16-bit counterpart available. Besides, the absence of an instruction not necessarily means it is not used, it may only mean that the search base is incomplete.
- Organize instruction in groups of similar functionality and collect statistics about the presence of each group in the analyzed programs. Most used instructions within the same group tend to share similar formats and are candidates to share a common 16-bit format. Less used instructions are likely to be discarded from having a 16-bit counterpart, unless proven worthy by other analysis.
- Usage count of each instruction in the ISA. Top used instructions are directly responsible for final 16-bit compressibility and need to be the first candidates for inclusion; receiving fewer encoding restrictions for their 16-bit form than least used ones.

Based on these criteria, we perform experiments using *MiBench* [25], *mediabench* [38] and the *Linux Kernel*, all compiled with GCC without floating point support and we disconsidered some special privileged instructions such as traps.

The total usage of SPARCv8 ISA instructions by MiBench, mediabench and Linux Kernel is 42.79%, 42.29% and 62.69%. The later uses more instructions since it performs specials tasks, such as privileged instructions and hand crafted assembly code. Roughly, 40% of the ISA instructions are never used, an initial set of instructions to dis-consider.

We divide instructions by groups of similar functionality and obtain static usage by each group. In Figure 2, top used instructions in mediabench, MiBench and Linux Kernel are from the *alu_arith* group; 22.42%, 22.63%

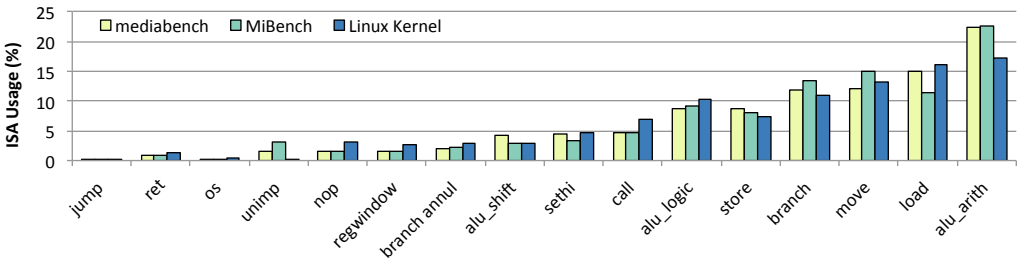


Figure 2: Percentage of SPARCv8 instructions usage by groups

and 17.21% respectively. Table 1 shows *alu_arith* in mediabench, some of the top used instructions are *add_imm*, *subcc_imm*, *add_reg* and *subcc_reg*. Thus, like mentioned before, instructions like these in each group are the main candidates.

Table 1: Instruction usage in the top 4 most used groups – mediabench

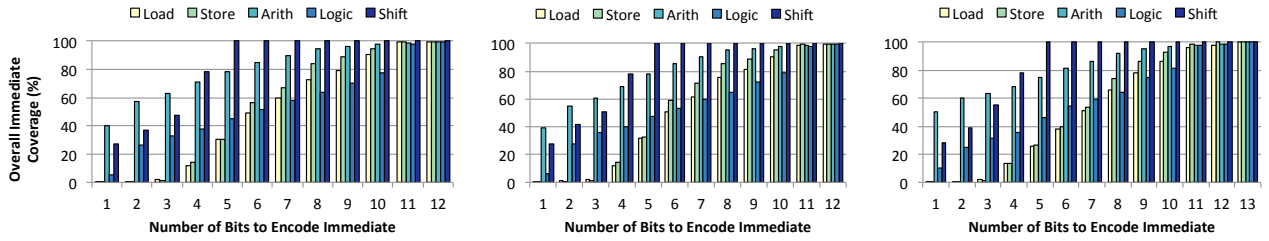
Usage	Group	Instructions			
22.42%	alu_arith	(29.59%)add_imm (28.49%)subcc_imm (15.93%)add_reg (12.28%)subcc_reg (05.68%)sub_reg (02.37%)smul_reg	(01.45%)addcc_imm (01.28%)subx_imm (01.22%)addx_imm (00.33%)umul_reg (00.32%)smul_imm (00.28%)udiv_reg	(00.20%)subx_reg (00.17%)addcc_reg (00.16%)sdiv_reg (00.14%)addx_reg (00.04%)udiv_imm (00.04%)sub_imm	(00.02%)smulcc_reg (00.01%)udivcc_reg (00.00%)sdivcc_reg
15.00%	load	(61.16%)ld_imm (16.69%)ld_reg (05.59%)ldub_reg	(04.46%)lduh_imm (03.54%)ldd_imm (01.85%)ldub_imm	(01.71%)ldsb_reg (01.70%)lduh_reg (01.41%)ldsh_imm	(00.71%)ldsb_imm (00.65%)ldsh_reg (00.54%)ldd_reg
12.14%	move	(71.93%)or_reg	(28.07%)or_imm		
11.76%	branch	(32.66%)ba (22.71%)be (13.58%)bne (07.86%)ble	(06.06%)bl (04.79%)bleu (04.27%)bg (02.91%)bgu	(02.76%)bge (00.80%)bcs (00.67%)bcc (00.58%)bpos	(00.35%)bneg

Immediate and Register Encoding. The basic fields needed when encoding instructions are opcode, immediate and registers. The opcode bit-width is determined by the number of instructions in the ISA and by considering a good compromise between available functionality and opcode space occupation: we must avoid opcode space waste by refusing to include infrequent functionalities. Additionally, we must consider that a big opcode field restricts the size of immediate and register fields.

Immediates usually occupy most part of an instruction and are the main target for compression. Thus, as mentioned in Section 8, 16-bit extensions – Thumb, MIPS16 and MicroMIPS – are designed with instructions containing restrained immediate fields. SPARCv8 has the following immediate field sizes across three formats: 30, 22 and 13 bits. For instance, the top used instruction groups (Figure 2) *alu_arith*, *load*, *move*, *store* and *alu_logic* are composed of instructions using the SPARCv8 format with 13-bit immediate field.

Figure 3 shows the accumulated number of bits needed to encode the immediate in all arithmetic, logic, shift, load and store instructions for the mediabench, MiBench and Linux Kernel. Notice that, although the immediate field is 13-bits wide, for the three analyzed cases 3a, 3b and 3c, more than 80% of the arithmetic instructions

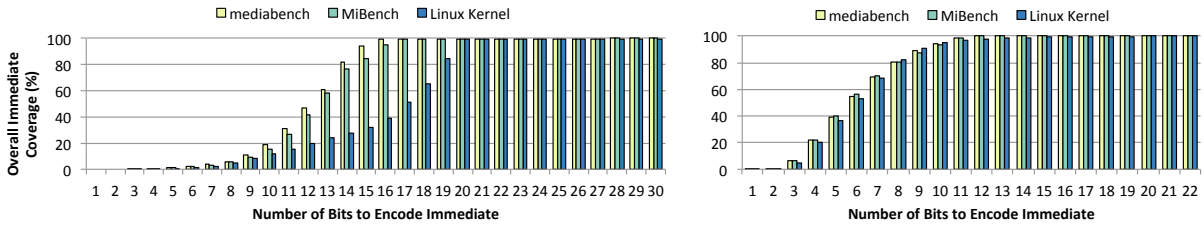
require 6 bits or less. Also, near 80% of the load and store instructions can be represented with 9 or less immediate bits.



(a) mediabench (b) MiBench (c) Linux Kernel

Figure 3: Immediate size usage for SPARCv8 Format 3 Instructions – load, store, arithmetic, logical and shift

Figure 4 represents the immediate usage for *calls* and *branches*. 80% of call instructions (30-bit immediate field size) can be encoded using 14-bits (Figure 4a) in mediabench and MiBench whereas the Linux kernel needs 19-bits. 80% of branch instructions (Figure 4b) need only 8 bits out of 22 in the three cases.



(a) Calls (b) Branches

Figure 4: Immediate size usage for SPARCv8 Format 1 and 2 Instructions - calls and branches

Register fields are smaller than immediate ones but also important, since they are an intrinsic part of an instruction. SPARCv8 uses a register window architecture [48], containing 32 general purpose registers, encoded in 5-bit register fields, divided into 4 categories: *input (%i0-%i7)*, *output (%o0-%o7)*, *local (%l0-%l7)* and *global (%g0-%g7)*. Instruction formats with two and three register fields are available, but two address[2] instructions do not exist. The %g0 register always yields a zero value, %g2-%g7 are never used by ABI [30] restrictions and %o6 and %i6 are aliases to the stack (%sp) and frame pointer (%fp)[3].

Immediate field sizes may vary among instructions, but register field sizes are fixed, allowing all instructions to have the same set of register visibility. In SPARCv8, %fp and %sp are always visible and accessible by all instructions accessing registers. Furthermore, those registers are only referenced during load and store to local variables (by *ld_imm* and *st_imm* instructions), function frame allocation (*save* and *restore*) and stack frame addresses copies (*add_imm*). Hence, both registers are reserved and never used by register allocation.

Table 2a shows that %fp is the fifth most used register and is used by 30.7% and 26.53% of all *ld_imm* and *st_imm* instructions, as shown in Table 2b. The situation suggests that we have specific load, store and save 16-bit instructions to implicitly access such registers, hiding them from regular instructions, which leaves extra room for encoding regular registers, improving register allocation.

[2]Situation where source and destination registers share the same register field.

[3]Usually %fp is used for accessing the stack frame, but %sp can be used instead if frame pointer is omitted by compilers.

We also analyzed immediates within *ld_imm* and *st_imm* instructions using %fp and no immediate can be encoded with less than 5-bits[4] as illustrated for *ld_imm* in Table 2c; such instructions highly demand large immediate fields. However, stack access for *ld_imm* and *st_imm* is always aligned and encoding the two least significant bits is needless. Then, besides implicitly encoding %fp and %sp, we can mitigate a large immediate field size problem by not encoding these two bits. A similar approach should be done to the *add_imm*, *save_imm* and *restore* instructions using %sp or %fp.

Registers[5]	Usage
%g1	16.77%
%i0	9.15%
%o0	8.97%
%o5	6.46%
%i6/%fp	6.31%

(a) Overall top 5 used registers

Instructions	%fp or %sp Usage
ld_imm	30.75%
st_imm	26.53%
save_imm	13.54%
add_imm	13.18%
std_imm	3.64%

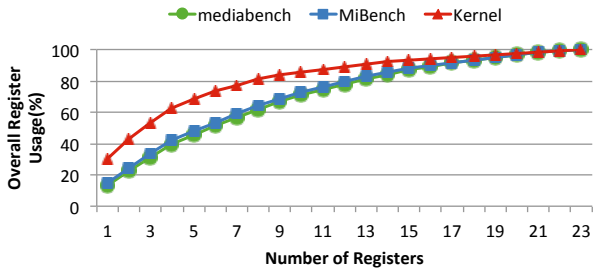
(b) Top %fp or %sp Usage In Instructions

Bits	Coverage
1-4	0%
5	12.34%
6	36.89%
7	52.40%
8	79.43%

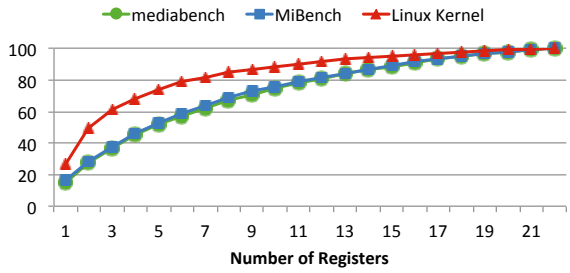
(c) *ld_imm* with %fp: Necessary bits to encode immediates

Table 2: Used registers and other %sp and %fp relative measurements for all benchmarks

The analysis in Figure 5a dis-consider %fp and %sp usage and shows that 8 distinct registers are responsible for 62%, 64% and 81% of the total register usage in instructions from mediabench, MiBench and Linux Kernel respectively. In Figure 5b, 67%, 69% and 85% of the total registers defined in all instructions are represented by only 8 different registers. The results show that using 3-bits for registers fields in the 16-bit extension is a good tradeoff: although 4-bits could be used, we reduce by a factor of four the number of registers and cover roughly 65% of all register usage, leaving one more bit for opcode or immediate.



(a) Registers Used in a Instruction



(b) Registers Defined in a Instruction

Figure 5: Overall register usage cover by number of registers

27% and 37% of three address instructions in *alu_arith* and *alu_shift* groups behave like two-address instructions - one of the source registers is the same as destination. In fact, even for instructions addressing only two registers in the *alu_arith* group, 26% of them uses the same source and destination register. Hence, two-address 16-bit instructions can be created based on instructions following the mentioned profiles.

Dynamic analysis. We analyzed execution of SPARCv8 binaries to dynamically collect information about the overall presence of instructions. Those with very low static usage count can have a high execution count. The dynamic analysis gives weight to highly executed instructions so they can have a 16-bit counterpart, mitigating a

[4]SPARCv8 ABI reserves a 16 byte space closer to %fp which had no use in any analyzed binary

[5]Registers %g0 and %g2-%g7 are disconsidered from the analysis because of aforementioned explanation about such registers.

potential performance degradation. For example, the instruction *xor_reg* takes 0.10% of all instruction usage in MiBench, while it represents 10% of all executed instruction in the same benchmark.

4 Integer Linear Programming Model

Based on the analysis described in Section 3, we developed an *ILP* to represent the problem of finding the best field sizes for every instruction format in the 16-bit extension. The ILP solution assisted in the creation and definition of SPARC16 instruction formats.

An input instance to the ILP model consists of a tuple (S, F, I, c). Each element $s \in S$ is a set of fields. Table 3a shows the fields for each of the sets $s \in S$. 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 $s \in S$ has three bits.

	Fields			
$s \in S$	1^{st} Opcode	2^{nd} Opcode	Register(s)	Imm
I	opc_1			imm_1
RI	opc_1		reg_1	imm_2
RRI	opc_1		reg_1, reg_2	imm_3
RR	opc_1		reg_1, reg_2	
RRR	opc_1		reg_1, reg_2, reg_3	
I2	opc_1	opc_2		imm_4
RI2	opc_1	opc_3	reg_1	imm_5
RRI2	opc_1	opc_4	reg_1, reg_2	imm_6
RR2	opc_1	opc_5	reg_1, reg_2	
RRR2	opc_1	opc_6	reg_1, reg_2, reg_3	

(a) Fields $s \in S$

	Field sizes (bits)			
Formats	opc_1	reg_1	reg_2	imm_3
F1	1	3	3	9
F2	2	3	3	8
F3	3	3	3	7
F4	4	3	3	6
F5	5	3	3	5
F6	6	3	3	4
F7	7	3	3	3

(b) Possible RRI formats

Table 3: Description of ILP fields and inputs

An attribution of size to each of the fields of an element $s \in S$ corresponds to a format. Formats follow 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 following the aforementioned rules and placed them into the set F. For example, Table 3b presents generated formats for the RRI set - containing one primary opcode, two registers and a immediate. The maximum opcode size is 7 bits which can encode 128 different instructions; usage of more bits to opcode field would reduce available bits to encode registers and immediates.

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, not only does this approach mitigates 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 mentioned in Section 3. 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 three-addresses instructions as two-addresses instructions (i.e., forcing a register to work simultaneously as source and destination of an operation).

In order to allow the ILP 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.

Therefore, 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

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

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

The ILP solution yields a initial set of formats, each holding one or more instructions. The formats are post processed and modified in order to build a more regular ISA. We also manually include other special instructions to perform specific tasks.

Instruction Formats. SPARC16 instructions formats, illustrated in Table 4, have different shapes and sizes. Every format is identified by a 5-bit major opcode and some formats also hold a secondary opcode - expanding the number of total opcodes.

15 14 13 12 11	10 9	8 7	6 5 4 3	2 1 0
I-type op	immediate			
IB-type op	a	immediate		
RI-type op	immediate			reg
RRI-type op	immediate		reg	reg
LW/ST-type op	op2	immediate	reg	reg
I2-type op	op2	a	immediate	
I2B-type op	op2	immediate		
RI2-type op	op2	immediate		reg
RRI2-type op	op2	imm	reg	reg
RR-type op	op2		reg	reg
RRR-type op	op2	reg	reg	reg
MOV-type op	op2	T	reg32	reg8

Table 4: SPARC16 formats

For instance, the *I* format is used to accommodate *call* instructions, which requires a large immediate field, but no registers. *IB* format hold some branches; *always*, *if equal* and *if not equal*, with the *annul* [48] field in bit 10. On the other hand, *RRR* format has no immediate field and is used to encode instructions that operate on three registers, such as *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 two-address instructions: one of the registers is used as source and destination of the operation.

The instructions *load word* and *store word* are allocated in the *RRI* format whereas other types of load and store belong to the *LW/ST* format. In SPARCv8, load and store instructions should always access aligned memory addresses regarding their intrinsic alignment; *byte*, *half*, *word* and *double*. Therefore, SPARC16 load and store instructions, as suggested in Section 3, discard the bits necessary for alignment in the immediate field encoding, the immediate is then shifted back accordingly prior to execution. For example, the immediate value 16 in a load word instructions, is encoded as the value 4, saving the need for two extra bits in the immediate field. Call and branch instructions are encoded using the same restrictions: both have 2-byte aligned immediates.

The format *RRI2*, very similar to *RRI*, represents instructions with a low usage rate, such as division and multiplication.

Mode exchange. The alternation between SPARCv8 and SPARC16 modes is accomplished by special instructions. Call with exchange (*callx*) or branch with exchange (*bx16*) instructions are used to switch from SPARC16 to SPARCv8. Conversely, jump and link with exchange (*jmplx*) and branch with exchange (*brx*) instructions are used to switch back. Both were added to the SPARCv8 ISA by using opcodes in reserved space. The least significant bit in the 32-bit target address determines the target routine's mode – 0 means SPARCv8 and 1 means SPARC16.

Register access. 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 one, the latter performs the opposite operation. The *MOV* format is used to represent these instructions using a three bit field *reg8*, used to index a SPARC16 visible register, and a five bit field *reg32*, used to index one of the 32 registers from the SPARCv8 register bank. These instructions are used to move function arguments into specific registers and to compute more elaborate arithmetic involving %sp or %fp.

Some instructions in SPARC16 include implicit access to registers %sp, %fp, %g0 and %ra, mitigating the drawback of having a smaller set of visible registers. Also, a implicit register reference means three free bits to encode a larger immediate. For example, *ldfp* and *ldsp* load instructions implicitly uses %fp and %sp as source registers, both encoded in the *RI* format. The *addfp* and *addsp* instructions work the same way but providing direct stack offsets computations through addition. The *savesp* implicitly uses and defines %sp, optimizing the very common SPARCv8 stack frame allocation pattern *save %sp,imm,%sp*, leaving room for more immediate encoding.

The EXTEND mechanism. Loading large constants using only SPARC16 is an expensive operation in terms of number of instructions. The reduced immediate field forces that several instructions might be used, yielding sometimes, a code size equal or worse than performing the same operation within SPARCv8. At the same time, the penalty of a SPARCv8 mode exchange to perform a simple constant load is prohibitive.

Large immediates are handled using an auxiliary instruction similar to the MIPS16 *EXTEND*, Thumb2 and MicroMIPS 32-bit instruction intermix. The 16-bit EXTEND instruction is used for increasing the available bits for encoding immediates. It has a reserved opcode, must precede a regular 16-bit instruction and contain an extra immediate field for use by the following instruction.

31 30 29 28 27	26 25 24 23 22 21 20 19 18 17 16	15 14 13 12 11	10 9 8 7 6 5 4 3 2 1 0
EXTENDED	immediate[21:11]	opc_1	immediate[10:0]
EXTENDED	unused | rs2	opc_1	opc_2 | rs1 | rd

Table 5: Extended formats *I* and *RR* - 32 bits

Instructions needing larger immediate can be extended using the available bits from the EXTEND instruction. The mechanism is also applied to provide additional registers for some instructions, a functionality not present in MIPS16. We also provide a 32-bit *SETHI* instruction which can be used in 16-bit mode, has a reserved opcode and does not use the EXTEND mechanism. The instruction can directly address a 22-bit immediate and a 3-bit destination register - the EXTEND instruction provides the maximum of 19 bits to any other instruction. SETHI or instructions which depend on a preceding EXTEND can be used interchangeably with all other 16-bit instructions without the need of any alignment requirement.

6 Implementation

This section describes the hardware and software base used to evaluate the SPARC16 extension. Additionally, we present compiler optimizations to assist in 16-bit code generation.

6.1 Hardware

SPARC16 is an extension to the SPARCv8 instruction set and it relies on a regular SPARCv8 pipeline. The SPARC16 instructions are translated to their 32-bit counterparts during execution time by placing a PDC decompressor between the instruction cache and the SPARCv8 pipeline, as shown in Figure 6. A PDC design yields more performance than other mechanisms (see Section 8) and is adopted by Thumb [3] and MIPS16 [32].

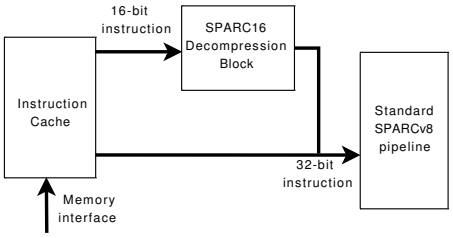


Figure 6: SPARC16 decompression diagram

The decompressor is integrated into a SPARCv8 processor Leon3 [22]. The implementation [17] focused on three key factors:

- Integrate the decompressor with the minimum hardware overhead in the Leon3 processor.
- No processor cycle time degradation after integration.
- Guarantee that SPARC16 code is reachable through jumps and calls, even if the target is not 4-byte aligned.

We avoid overhead and cycle time degradation by carefully choosing the bit encoding for each 16-bit instruction, simplifing the conversion between SPARC16 and SPARCv8. The processor clock is not reduced by two reasons: (1) The critical path in the chosen SPARCv8 processor is not in the Instruction Decode stage, and (2) The bigger decode lookup table only has 32 lines (5-bit address).

6.2 Software

Emulators. In order to validate SPARC16 instructions by running real programs and to collect extra information about execution, we used the QEMU emulator [5]. QEMU is a flexible process and system virtual machine, translating instructions from a target architecture into instructions of the host. We integrated a SPARC16 front-end into the existing SPARCv8 port, making it possible to run any SPARC16 compiled application.

Toolchain. A toolchain consists of a compiler front-end, back-end, assembler, linker and libraries to compile applications for a given architecture. We developed a SPARC16 toolchain based on LLVM 3.2 [37] from front-end to assembler; GNU Binutils 2.22 [24] for the linker and the uClibc [1] C library compiled for SPARCv8.

The SPARC16 linker can merge together object files from SPARC16 and SPARCv8. The approach is limited and, during linking time, all calls to external modules are considered to target SPARCv8 code. Thumb2 and MicroMIPS toolchains provide different mechanisms to allow a similar linking behavior between different modes. Using this method, we reuse a SPARCv8 compiled uClibc in our toolchain.

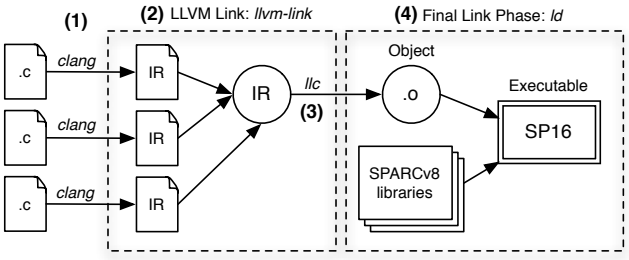


Figure 7: SPARC16 Toolchain - Compilation flow

One issue with the mentioned approach is the compilation of multi-source applications. In a common compilation flow, each source code is translated to a separated object file prior to linking - yielding several modules or object files with SPARC16 code, breaking the linker restriction. The LLVM infrastructure mitigates this problem by providing a linking phase between its intermediate representation modules prior to code generation.

The complete compiler flow for SPARC16 applications, illustrated in Figure 7, follows: (1) LLVM's intermediate representation *IR* is generated for each source code (2) Different IR modules are linked together inside one final IR module (3) The final IR module is translated into an object file (4) The object file is linked with other SPARCv8 objects and libraries.

ABI. The Application Binary Interface *ABI* defines a set of conventions to be followed by compilers and operating systems to make sure applications and libraries from different sources are able to interact with each other. Function calling sequence, ELF headers, object files relocations and dynamic linking are example of conventions mentioned in a ABI.

The SPARC16 ABI is an extension of the SPARCv8 ABI [30] and inherits most part of its conventions. The %o0-%o5 and %i0-%i5 group of regitsers are used to pass and receive the first six arguments within a function call, while the remainder are passed on the stack. *MOV8to32* and *MOV32to8* instructions move the arguments to and from the hidden registers - %o3-%o5 and %i3-%i5. The stack layout remains the same, including the reserved space on the stack for variable arguments.

Regarding the calling convention, no modifications need to be implemented to support interoperability between SPARC16 and SPARCv8. The only necessary ABI addition relies on the definition of relocations. SPARC16 preserves all relocations definitions from SPARCv8 and additionally define its own group. Hence, the linker is able to link between modes since it can understand relocations from both of them together.

6.3 Compiler Optimizations.

During code generation, the compiler should minimize the code compression ratio considering the 16 bits extension constraints; thus, we developed a set of specific target optimizations for SPARC16 ISA to enhance compressibility using the LLVM compiler pass interfaces.

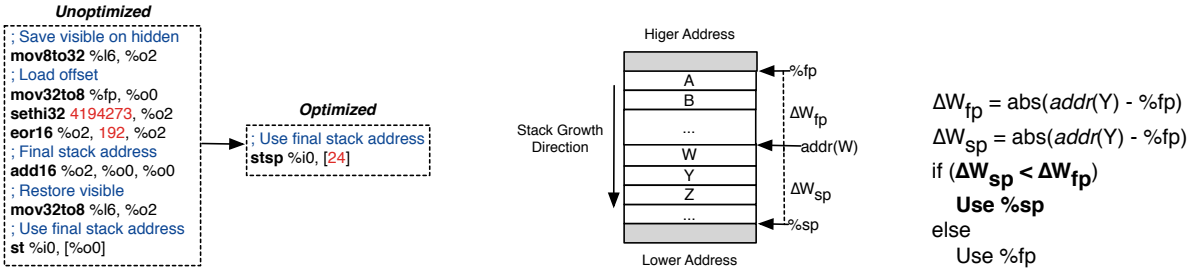
Delay slots. The SPARCv8 architecture defines that branches and call instructions requires a following delay slot instruction [48]. The delay slot can be fulfilled with any instruction without data or control hazards, or by a simple no-operation *NOP* instruction. The SPARC16 inherits the same restrictions in its 2-byte branches and calls and poses an additional restriction: instructions containing the EXTEND prefix must not be present in a delay slot because logically they represent a 4-byte instruction. A dedicate SPARC16 pass handles the delay slot fulfillment.

Instruction size reducer. All the instructions containing immediates – arithmetic, logic, load, stores, branches and calls – are conservatively emitted by the code generator with the EXTEND form. Then, by default, all such instructions prior to register allocation appear as 32-bit instructions.

In a late, post register allocation pass, the EXTEND instructions is removed whenever 16-bit instructions have enough immediate bits to represent the entire immediate. The pass is not able to reduce the sizes of external calls because computing targets and relocations is only possible at linking time.

Assembler relaxation. The LLVM SPARC16 assembler uses a pass [28] to relax all possible branch and call instructions in a module and reduce their instruction sizes to 16-bit whenever the target displacement fits.

Mixed Stack Access. Immediate field sizes are heterogeneous across SPARC16 instructions and the EXTEND is always used to increase constant coverage. When the immediate needs more bits than EXTEND may provide, a *sethi* and *extend+or* is used, an approach similar to SPARCv8. We name this group of instructions *SethiEOr*.



(a) Unoptimized vs Optimized (b) SPARC16 Stack Frame (c) Stack computation

Figure 8: Mixed %fp and %sp optimization

However, the use of instructions from *SethiEOr* increases code size and must only be used if no smaller approach is found. For instance, access to stack locations relying on large stack offsets is done via *SethiEOr* and additional instructions to copy the stack pointer, a situation illustrated in the unoptimized fragment from Figure 8a. We propose an optimization to this scenario, by accessing stack locations from closer locations when possible, avoiding the use of many instructions. The result is shown in the optimized fragment from Figure 8a.

The optimization is accomplished by using the closer available stack pointer register: %fp or %sp. Figure 8b shows that everything between the two registers is the current stack frame. We compute a delta of the stack element address against %fp and %sp (see Figure 8c); the shorter delta indicates which register to use when accessing the stack. The approach is limited to functions which preserve the frame pointer register %fp, make no use of stack alloca functions and have no dynamic stack reallocation.

7 Results

Program	Ratio
rawcaudio	71.8%
rawdaudio	72.1%
g271	68.7%
gsm	77.3%
jpeg	68.1%
mpeg2decode	75.1%
GeoMean	72.09%

(a) mediabench

Program	Ratio	Program	Ratio
basicmath	81.8%	bitcount	68.7%
susan	81.9%	jpeg	71.0%
lame	82.3%	dijkstra	75.9%
patricia	67.1%	stringsearch	69.2%
blowfish	71.8%	rijndael	77.6%
sha	72.4%	CRC32	69.6%
fft	80.3%	adpcm	72.1%
gsm	77.3%	GeoMean	74.43%

(b) MiBench

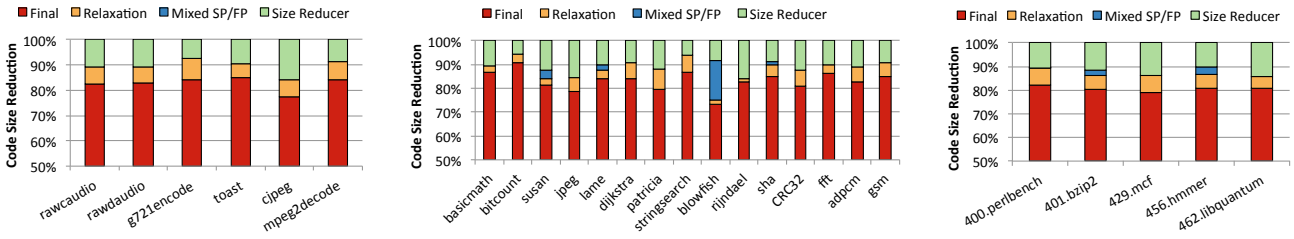
Program	Ratio
400.perlbench	71.7%
401.bzip2	72.9%
429.mcf	71%
456.hmmer	74.1%
462.libquantum	69.3%
GeoMean	71.78%

(c) SPEC CINT2006

Table 6: mediabench, MiBench and SPEC CINT2006 - Compression Ratios

We evaluate SPARC16 code compression by computing the compression ratio for programs in the mediabench, MiBench and SPEC CINT2006 benchmarks. The compilation of each program relies on the LLVM based toolchain for SPARC16 and SPARCv8. The compiler flag *-Os*, a set of code size optimizations, is used among all programs. In mediabench (Table 6a), the best compression ratio is 68.1% for *jpeg* and the worst case 77.3% for *gsm*. Table 6b shows that MiBench ratios are heterogeneous, ranging from 67.1% in *patricia* to 82.3% in *lame*. Programs from SPEC CINT2006 have a very homogeneous compression ratio, Table 6c shows a geometric mean of 71.78%.

The optimizations mentioned in Section 6.3 have an important role in the achieved compression ratios: optimizations are responsible, in average, for reducing 20% in code size from SPARC16 programs. In order to maintain a fair comparison against SPARCv8, the optimizations explore SPARC16 instructions set compression opportunities, avoiding compiler misusage of such new instructions.

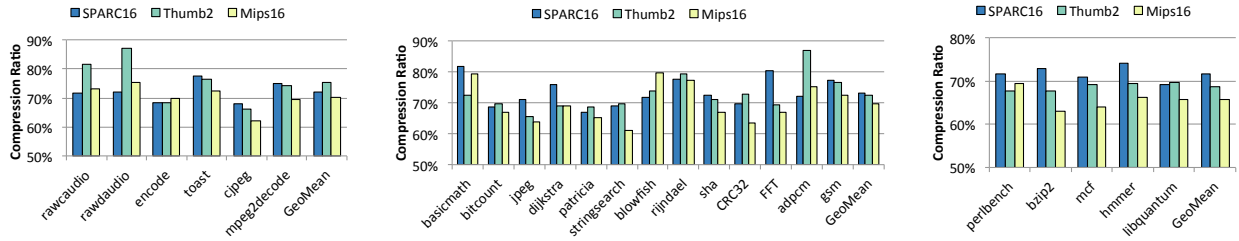


(a) mediabench (b) MiBench (c) SPEC CINT2006

Figure 9: Effect of optimizations in code size reduction of SPARC16 programs

Figure 9 shows the overall code size reduction achieved in unoptimized SPARC16 programs by applying optimizations. The size reducer and relaxation cause a greater reductions in most analyzed programs, whereas the mixed %sp/%fp approach is effective in programs with functions containing big stack frames and many stack accesses.

In mediabench, Figure 9a, relaxation reduces code size by an average of 7% and the size reducer by 10%. This is roughly the same reduction caused by both optimizations in SPEC CINT2006 (Figure 9c) and MiBench (Figure 9b). The mixed %sp/%fp optimization causes a 2.1% and 2.8% code size reduction in *401.bzip2* and *456.hmmer* programs from SPEC CINT2006 while *blowfish*, *susan* and *lame* from MiBench are reduced by 16%, 3.5% and 2.2% respectively.



(a) mediabench (b) MiBench (c) SPEC CINT2006

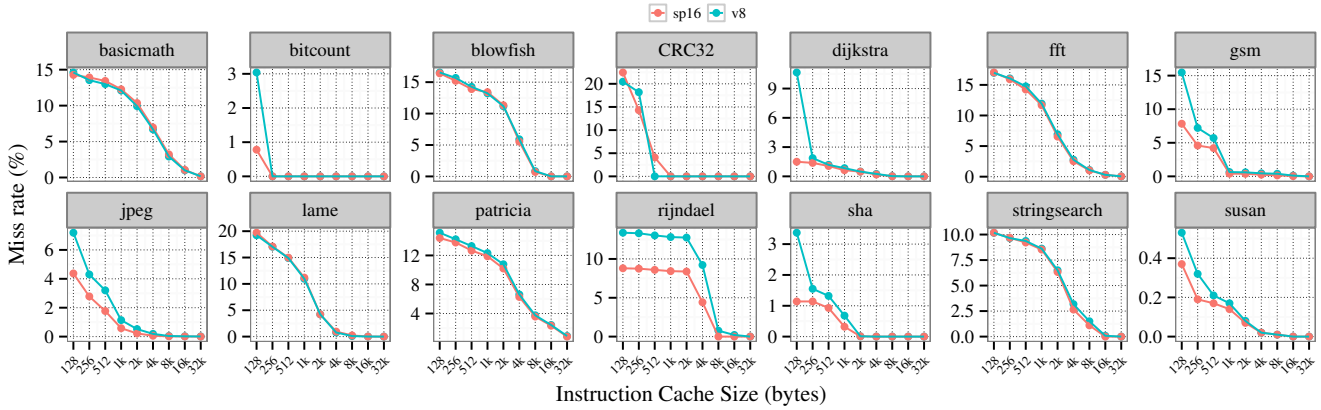
Figure 10: Compression ratio comparison between SPARC16, Thumb2 and Mips16

We also evaluate SPARC16 compression ratios against Cortex-A9 ARM/Thumb2 and Mips32/Mips16 by compiling programs for both architectures using LLVM 3.3. Figure 10 compares mediabench, MiBench and SPEC CINT2006 benchmarks; the geometric mean between all programs for each shows that SPARC16 achieves very similar compression ratios than Thumb2 and Mips16. Actually, SPARC16 yields better compression ratios than MIPS16 for 50% of MediaBench programs, and better than Thumb2 for 50% of MiBench programs. For instance, in the *adpcm* program, SPARC16 compression ratio is 15% smaller than Thumb2. Note that, although Mips16 and Thumb2 code generation are production quality and take advantage of other optimizations, code compression ratios are very similar from our implementation.

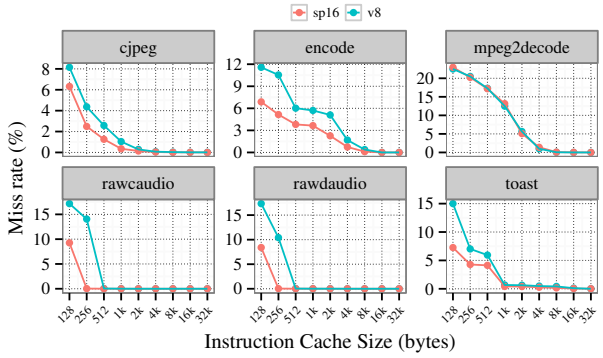
Performance. We evaluate SPARC16 performance by measuring and comparing SPARC16 versus SPARCv8 code impact on the instruction cache. The reduction of instruction cache misses leverages overall performance and at the same time leads to power consumption reduction.

Using SPARC16 and SPARCv8 support in QEMU, we collected execution traces of several programs and

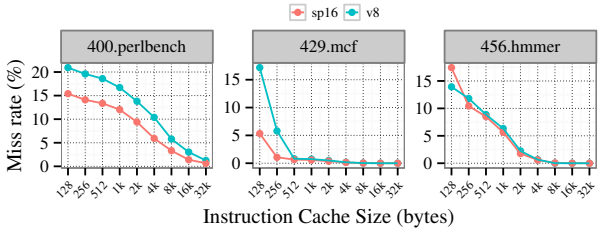
analyzed the traces using Dinero IV [19] cache simulator. We simulated power of two cache sizes from 128 bytes to 32 kbytes, in a 2-way, 32 bytes per line cache.



(a) MiBench



(b) mediabench



(c) SPEC CINT2006

Figure 11: SPARCv8 vs SPARC16 Cache Miss Rates

The dynamic instruction count for SPARC16 is usually higher than SPARCv8. The SPARCv8 version of *cjpeg*, for instance, dispatches 18M cache demand fetches against 23M in SPARC16 - a 22% increase in the number of executed instructions. As noted, comparing the absolute miss count values would be misleading; thus, we used the cache miss rates.

Figure 11 compares instruction cache miss rates between SPARC16 and SPARCv8 for the MiBench, mediabench and SPEC CINT2006 benchmarks. In the *rijndael* program, a reduction of 5% in the cache miss rates happens in any cache size between 128 and 4k bytes. Additionaly, in a 128 byte cache size configuration, the cache miss rates for dijkstra and gsm programs are reduced by 9% and 8% respectively.

Two SPARC16 programs have slightly worse miss rates than SPARCv8: *basicmath* and *CRC32*. The absence of alignment restriction between regular 2 byte and extended instructions in SPARC16 may generate more demand misses; whenever a extended instruction is present across cache lines, an extra cache miss is generated.

$$\begin{aligned} Cycles &= (Fetches - Misses) + Misses * Penalty \\ Speedup_{sp16} &= Cycles_{v8}/Cycles_{sp16} \end{aligned} \tag{1}$$

Using Equation 1 we evaluate SPARC16 performance against SPARCv8. Figure 12 shows the performance speedup estimation obtained for different penalty values and instruction cache sizes for the *rijndael* program. Line (1) and (2) shows a 13% and 48% performance speedup for a 150 cycle miss penalty. The cache sizes in which

only SPARC16 has no cache misses are represented in Lines (3) and (4) - a linear relation between speedup and penalty. Moreover, in Line (5) SPARC16 has worse performance than SPARCv8 for all estimated penalties; both architectures have no cache misses and the number of instruction fetches in SPARC16 is always higher.

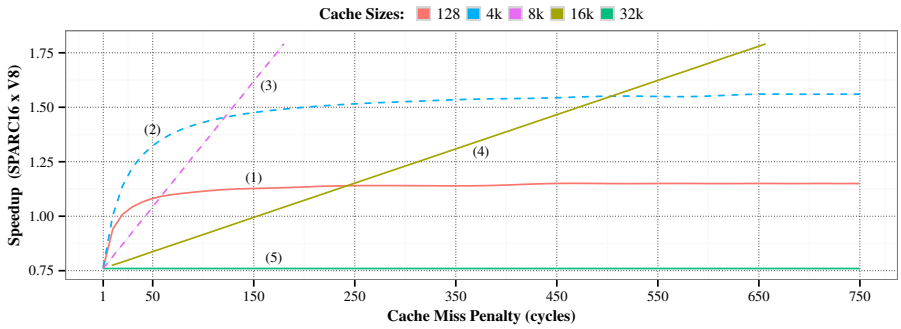


Figure 12: SPARC16 performance speedup against SPARCv8

Figure 13 compares instruction cache sizes from SPARCv8 and SPARC16 and shows that for several programs, the later can use smaller caches to run the same programs, without any performance degradation. For instance, considering a 50 cycle penalty miss, 4 programs can run in a 128 byte cache SPARC16 processor with the same performance the same 4 programs yield in a 256 byte SPARCv8 processor; a reduction in half the size. The reduction can achieve a factor of 16: one program in 350 cycle penalty configuration can use a 128 byte SPARC16 processor against a 2k byte in SPARCv8. Therefore, SPARC16 is one alternative for shipping devices with less memory while avoiding performance degradation.

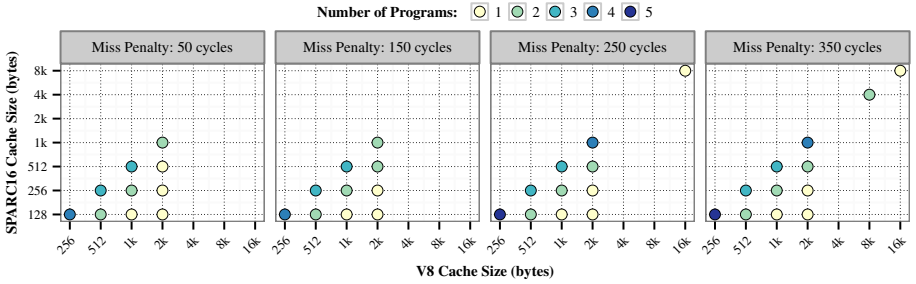


Figure 13: Equivalence of cache sizes without performance degradation

SPARC16 is faster, more power efficient, requires smaller instruction cache size and proved to be an excellent alternative to SPARCv8 and other 16 bit architectures.

8 Related Work

We use the term *Compression Ratio* to represent code size reduction. Equation 2 shows how the Compression Ratio should be calculated. Notice that, in this case, lower means better. All related overhead to implement compression must be considered within the ratio computation, a rule not strictly followed by all related work in the area. Example: a Compression Ratio of 56% means that the program is reduced to 56% of its original size.

$$Compression\ Ratio = \frac{Compressed\ Size + Overhead}{Original\ Size} \quad (2)$$

16-bit Instruction Sets. The *DLX* architecture, created by Hennessy and Patterson [45], 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 [11].

ARM introduced *Thumb* [3] as the first 16-bit extension in ARM7. Later on, *Thumb2* was released and superseded initial Thumb, introducing additional features. Thumb2 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. Mode exchange is achieved during runtime through *BX* and *BLX* instructions: branch and call instructions that flip the current mode bit in a special processor register. A group of only 8 registers including the stack pointer and link registers are visible, but the remaining registers can also be accessed implicitly or through other special instructions. Results presented by ARM for Thumb, 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.

The MIPS16 [32] ISA is the first 16-bit extension released for MIPS. It contains capabilities to exchange between modes (using the *JALX* instruction), share address space, has only 8 visible registers and reduced immediate size. Special move instructions move data between visible and hidden registers, while special instructions may access hidden registers implicitly. New features introduced by MIPS16 include the *EXTEND* instruction: 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 [32].

The MicroMIPS [43], released in 2009, is a new 16-bit ISA for MIPS and is not compatible with MIPS16. It introduces 54 instructions and is supported as a distinct mode from MIPS32 and MIPS64. While in MicroMIPS mode, each instruction has a 16-bit and 32-bit version and no *EXTEND* instruction is required, a design very similar to Thumb2. Additionally, 16-bit and 32-bit instruction can mixed altogether without any alignment restrictions. Results provided by MIPS technologies [43] reports an average of 65% compression rate for the CSiBE [54] benchmark and a 2% speed-up against MIPS32 for the Dhrystone benchmark.

Compression and Decompression in Software. Fraser [21] changed the compiler to issue a compact representation and a custom interpreter to it instead of executable code. Liao [41, 42] abstracts most executed instruction blocks into procedures and the original code is replaced by a dictionary entry for the procedure. Raeder [12] studies compression of Java bytecodes with a 70% compression ratio and 30% performance loss. Every compression approach entirely in software suffer from performance degradation during execution.

Mixed Software and Hardware. Kirovski et al. [31] compress whole procedures and use a directory service called *Procedure Cache* to solve references: decompression is realized by demand in dedicated RAM regions resulting in a 60% compression ratio and 10% performance loss.

Xu et al. [53] propose a memory compression architecture for the first Thumb ISA, achieving a Thumb code size reduction from 15% to 20%. A high-speed hardware decompressor also improves timing performance of the architecture, resulting in performance overheads limited within 5% of the original application. Also, to circumvent the inefficiencies of only using 16-bit instructions in the first Thumb release, Krishnaswamy [34] creates the ThumbAX extension, prefixing 16-bit instructions with an extra halfword to achieve equivalent functionality of a regular 32-bit ARM instructions, without the need of mode exchange. The mechanism is similar to Mips16 EXTEND and a very similar mechanism is present in Thumb2.

Two hardware models can be used for code decompression: *Processor Decompressor Memory (CDM)* and *Cache Decompressor Cache (PDC)*. In CDM the hardware decompression engine is present between the processor and memory, whereas in PDC it is between processor and cache. Compression methods using PDC yields better performance [6, 8, 39, 40, 44] than CDM [2, 51]. In PDC, because the cache holds compressed code, the number

of cache hits increases, diminishing the number of accesses to the main memory which leverages the bandwidth between the processor and cache. Lekatsas [40] and Benini [6] propose PDCs respectively resulting in 65% and 72% compression ratio and 25% and 28% performance gain.

Billo [8] and Wanderley [44] also proposes a compression mechanism for the SPARC architecture and combine a PDC with dictionaries built upon static and dynamic usage of instructions, achieving compression ratios from 72% to 88%, up to 45% performance gain and 35% on power consumption reduction.

The use of a dictionary [4,13,26,35,47] to compress code tends to reduce energy consumption and improves the performance with a reasonable static compression ratio. Using bitmask and prefix based Huffman encoding [26], the compression ratio improves by 9–20%. Kumar [35] analyzes compression for variable-length ISA RISC processors with two approaches: using a bit-vector and using a reserved instruction to identify code words, results demonstrate an speed-up of up to 15%, reduction in code size (up to 30%), bus-switching activity (up to 20%) leading to power consumption decline.

Other studies [9,10,14,46] use hardware support approaches to optimize the code compression/decompression. Bonny [9,10] achieved compressions ratios as low as 56% using a set of applications applied to ARM, MIPS and PowerPC architectures. Qin [46] implements a fast parallel decompression and improved the decode bandwidth up to four times, with minor impact (less than 1%) on compression efficiency. Corliss [14] shows reductions in code size up to 35%, 10% reduction in power consumption and 20% performance improvements.

Compiler Optimizations. The use of 16-bit extensions do not always guarantee a smaller code size. Code for loading large constants, for instance, is smaller and faster when using 32-bit instructions, since there is more space to encode immediates and extra registers. Hence, compiler code generators can apply optimizations to take advantage of mixed 16 and 32 bits ISAs extensions like MicroMIPS and Thumb2.

Krishnaswamy [33] proposes a coarse grained approach where only one bit-width type of instructions, 16 or 32-bit, could be emitted in each function. Profile information is used to select hot functions and several heuristics to decide the bit-width of each function in a program are evaluated: results range from 69,5% e 77,2% compression ratio against pure 16-bit code.

Edler [20] uses a compiler for *ARCOMPAT*, a mixed 16 and 32-bit ISA, to propose a special instruction selection heuristic for mixed ISAs. The heuristic avoids the emission of 16-bit instructions when it is not profitable. First, only 16-bit instructions are selected during instruction selection. Second, a special register allocator annotates all places where the usage of 16-bit instructions are responsible for generating spills. Finally, the instruction selection is called again, but using annotated data, avoids using 16-bit instructions in the potential spill sites. The feedback-guided instruction selection improves improves performance by 17% with 85% compression ratio.

The 32-bit ISA *UniCore32* and its 16-bit ISA counterpart *UniCore16* are analyzed by Xianhua [52]. UniCore16 has regular instructions, such as *add* and *mov*, capable of mode exchange. The compiler emits 32-bit instructions by default and heuristics may replace them for 16-bit instruction during link-time when profitable. The compression ratio is 73% against pure UniCore16 without any performance drawback.

Sutter [15,50] applies link time optimizations to ARM binaries: reconstruction from final executables is done through the creation of an augmented whole-program control-flow graph (*AWPCF*), where all text and data sections are considered and optimizations such as whole-program analyses, duplicate code and data elimination techniques are applied. The mechanism reduces code size from 16% to 18%, provide 8% to 17.4% performance gain and power consumption reduced by 7.9% to 16.2%.

Kumar [36] evaluates link time optimizations using a peephole technique with finite state machines instead of string matching, yielding code size reductions from 0.9 to 1.1%.

9 Conclusions

We present a method to analyze and detect compression opportunities in RISC ISAs through the specific SPARCv8 case study, followed by an extensive evaluation of SPARCv8 programs from different benchmarks and the Linux Kernel. A Integer Linear Programming uses the extracted information and assists in the selection of the best 16-bit instruction formats, minimizing the code size. We design and present a 16-bit instruction set extension for the SPARCv8 architecture: the SPARC16.

A FPGA implementation, software emulation and compiler code generator support are used to evaluate the SPARC16 extension. We achieve an average compression ratio of 72% for mediabench programs, 74% for MiBench and 72% for the SPEC2006. We show that some target specific compiler optimizations are crucial to allow code size reduction benefits from using 16-bit instructions; overall optimizations reduce unoptimized generated SPARC16 code by 20%. Therefore, we achieve SPARC16 compression ratios similar to the ones obtained by production quality MIPS and ARM 16-bit extensions, achieving better compression ratios than both in several programs.

Furthermore, we evaluated SPARC16 performance by analyzing effects of code size reduction in the instruction cache. Although the execution of SPARC16 programs yields more instructions than SPARCv8, lower cache miss ratios are achieved by SPARC16 compiled programs; the best ratio achieves 9% reduction in *dijkstra* from MiBench. Finally, we demonstrate how reduction in cache misses affects performance and that SPARC16 is an alternative for shipping devices with reduced cache sizes, without any performance degradation.

References

[1] E. Andersen. uclibc, http://www.uclibc.org.

[2] G. Araújo, P. Centoducatte, R. Azevedo, and R. Pannain. Expression tree based algorithms for code compression on embedded risc architectures. *IEEE Transactions on VLSI Systems*, 8(5):530–533, 2000.

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

[4] 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.

[5] F. Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[6] L. Benini, A. Macii, and A. Nannarelli. Code compression for cache energy minimization in embedded systems. In *IEE Proceedings on Computers and Digital Techniques, 149(4)*, pages 157–163, 2002.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] L. R. Clausen, U. P. Schultz, C. Consel, and G. Muller. Java bytecode compression for embedded systems. Technical Report RR-3578, Inria, Institut National de Recherche en Informatique et en Automatique, 1998.

[13] 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.

[14] 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.

[15] B. De Sutter, B. De Bus, and K. De Bosschere. Link-time binary rewriting techniques for program compaction. *ACM Trans. Program. Lang. Syst.*, 27(5):882–945, 2005.
[16] I. M. Division. *The PowerPC 405 Core*. International Business Machines (IBM) Corporation, 1998.
[17] L. Ecco. Sparc16: Uma nova visão de compressão para processadores sparc. Master's thesis, University of Campinas, 2010.
[18] L. Ecco, B. Lopes, E. Xavier, R. Pannain, P. Centoducatte, and R. Azevedo. Sparc16: A new compression approach for the sparc architecture. In *Computer Architecture and High Performance Computing, 2009. SBAC-PAD '09. 21st International Symposium on*, pages 169–176, 2009.
[19] J. Edler and M. Hill. Dinero iv trace-driven uniprocessor cache simulator, online at http://www.cs.wisc.edu/ markhill/dineroiv/, 2003.
[20] T. J. Edler von Koch, I. Böhm, and B. Franke. Integrated instruction selection and register allocation for compact code generation exploiting freeform mixing of 16- and 32-bit instructions. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, pages 180–189, New York, NY, USA, 2010. ACM.
[21] C. W. Fraser and T. A. Proebsting. Custom instruction sets for code compression. *http://research.microsoft.com/ toddpro/papers/pldi2.ps*, 1995.
[22] J. Gaisler. The leon3 processor. online, 2008. http://www.gaisler.com.
[23] M. Game and A. Booker. *CodePack: Code Compression for PowerPC Processors*. International Business Machines (IBM) Corporation, 1998.
[24] GNU. Gnu binutils, http://www.gnu.org/software/binutils.
[25] 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.
[26] 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.
[27] J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, Sept. 2006.
[28] R. Hundt, E. Raman, M. Thuresson, and N. Vachharajani. Mao – an extensible micro-architectural optimizer. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 1–10, Washington, DC, USA, 2011. IEEE Computer Society.
[29] IBM. *CodePack: PowerPC Code Compression Utility User's Manual. Version 3.0*. International Business Machines (IBM) Corporation, 1998.
[30] S. International. *SYSTEM V APPLICATION BINARY INTERFACE, SPARC Processor Supplement*. 1996.
[31] D. Kirovski, J. Kin, and W. H. Mangione-Smith. Procedure based program compression. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 204–213, Washington, DC, USA, 1997. IEEE Computer Society.
[32] K. Kissell. *MIPS16: High-density MIPS for the Embedded Market*. Silicon Graphics MIPS Group, 1997.
[33] A. Krishnaswamy and R. Gupta. Profile guided selection of arm and thumb instructions, 2002.
[34] A. Krishnaswamy and R. Gupta. Dynamic coalescing for 16-bit instructions. *ACM Trans. Embed. Comput. Syst.*, 4(1):3–37, 2005.
[35] 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.
[36] R. Kumar, A. Gupta, B. S. Pankaj, M. Ghosh, and P. P. Chakrabarti. Post-compilation optimization for multiple gains with pattern matching. *SIGPLAN Not.*, 40(12):14–23, 2005.
[37] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 75, Washington, DC, USA, 2004. IEEE Computer Society.
[38] 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.
[39] H. Lekatsas, J. Henkel, and W. Wolf. Code compression for low power embedded system design. In *DAC '00: Proceedings of the 37th conference on Design automation*, pages 294–299, New York, NY, USA, 2000. ACM.
[40] H. Lekatsas, J. Henkel, and W. Wolf. Design and simulation of a pipelined decompression architecture for embedded systems. In *ISSS '01: Proceedings of the 14th international symposium on Systems synthesis*, pages 63–68, New York, NY, USA, 2001. ACM.

[41] S. Y. Liao, S. Devadas, and K. Keutzer. Code density optimization for embedded dsp processors using data compression techniques. In *ARVLSI '95: Proceedings of the 16th Conference on Advanced Research in VLSI (ARVLSI'95)*, page 272, Washington, DC, USA, 1995. IEEE Computer Society.

[42] S. Y.-H. Liao. *Code generation and optimization for embedded digital signal processors*. PhD thesis, 1996. Supervisor-Srinivas Devadas.

[43] I. MIPS Technologies. micromips instruction set architecture, uncompromised performance, minimum system cost. Technical report, MIPS Technologies, Inc., MIPS Technologies, Inc. 955 East Arques Avenue Sunnyvale, CA 94085 (408) 530-5000, 2009.

[44] 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.

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

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

[47] 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.

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

[49] R. Stallman. *Using GCC: the GNU compiler collection reference manual*. Free Software Foundation, 2003.

[50] B. D. Sutter, L. V. Put, D. Chanet, B. D. Bus, and K. D. Bosschere. Link-time compaction and optimization of arm executables. *ACM Trans. Embed. Comput. Syst.*, 6(1):5, 2007.

[51] A. Wolfe and A. Chanin. Executing compressed programs on an embedded risc architecture. *SIGMICRO Newsl.*, 23(1-2):81–91, 1992.

[52] L. Xianhua, Z. Jiyu, and C. Xu. Efficient code size reduction without performance loss. In *Proceedings of the 2007 ACM symposium on Applied computing*, SAC '07, pages 666–672, New York, NY, USA, 2007. ACM.

[53] X. H. Xu, C. T. Clarke, and S. R. Jones. High performance code compression architecture for the embedded arm/thumb processor. In *CF '04: Proceedings of the 1st conference on Computing frontiers*, pages 451–456, New York, NY, USA, 2004. ACM.

[54] Árpád Beszédes, R. Ferenc, T. Gergely, T. Gyimóthy, G. Lóki, and L. Vidács. Csibe benchmark: One year perspective and plans. Technical report, University of Szeged, Hungary, 2004.