Compilation Framework for Code Size Reduction Using Reduced Bit-Width ISAs (rISAs)

AVIRAL SHRIVASTAVA, PARTHA BISWAS, ASHOK HALAMBI, NIKIL DUTT, and ALEX NICOLAU
University of California, Irvine

For many embedded applications, program code size is a critical design factor. One promising approach for reducing code size is to employ a "dual instruction set", where processor architectures support a normal (usually 32-bit) Instruction Set, and a narrow, space-efficient (usually 16-bit) Instruction Set with a limited set of opcodes and access to a limited set of registers. This feature however, requires compilers that can reduce code size by compiling for both Instruction Sets. Existing compiler techniques operate at the routine-level granularity and are unable to make the trade-off between increased register pressure (resulting in more spills) and decreased code size. We present a compilation framework for such dual instruction sets, which uses a profitability based compiler heuristic that operates at the instruction-level granularity and is able to effectively take advantage of both Instruction Sets. We demonstrate consistent and improved code size reduction (on average 22%), for the MIPS 32/16 bit ISA. We also show that the code compression obtained by this "dual instruction set" technique is heavily dependent on the application characteristics and the narrow Instruction Set itself.

Categories and Subject Descriptors: D.2.7 [**Software**]: Programming Languages—*Processors*

General Terms: Algorithms, Design, Experimentation, Measurement, Performance

Additional Key Words and Phrases: Code generation, compilers, optimization, retargetable compilers, dual instruction set, codesize reduction, code compression, rISA, thumb, narrow bit-width instruction set, register pressure-based code generation

1. INTRODUCTION

Programmable RISC processors are increasingly being used to design modern embedded systems. Examples of such systems include cell-phones, printers,

This work was partially funded by grants from the National Science Foundation (NSF) (MIP-9708067) and the Office of Naval Research (ONR) (N00014-93-1-1384), DARPA (F33615-00-C-1632), and by the Hitachi and Motorola fellowship.

Authors' address: Center for Embedded Computer Systems, School of Information and Computer Science, University of California, Irvine, Irvine, CA 92697; email: {aviral,partha,halambi, dutt,nicolau}@ics.uci.edu.

modems, handhelds etc. Using RISC processors in such systems offers the advantage of increased design flexibility, high computing power and low on-chip power consumption. However, RISC processor systems suffer from the problem of poor code density which may require more ROM for storing program code. As a large part of the IC area is devoted to the ROM, this is a severe limitation for large volume, cost sensitive embedded systems. Consequently, there is a lot of interest in reducing program code size in systems using RISC processors.

Traditionally, ISAs have been fixed width (e.g., 32-bit SPARC,64-bit Alpha) or variable width (e.g., x86). Fixed width ISAs give good performance at the cost of code size and variable width ISAs give good performance at the cost of added decode complexity. Neither of the above are good choices for embedded processors where performance, code size, power, all are critical constraints. Dual width ISAs are a good tradeoff between code size flexibility and performance, making them a good choice for embedded processors. Processors with dual with ISAs are capable of executing two different Instruction-Sets. One is the "normal" set, which is the original instruction set, and the other is the "reduced bit-width" instruction set that encodes the most commonly used instructions using fewer bits. A very good example is the ARM [Advanced RISC Machines, Ltd. 2003] ISA with a 32-bit "normal" Instruction Set and a 16-bit Instruction Set called "Thumb."

Other processors with a similar feature include the MIPS 32/16 bit TinyRISC [LSI LOGIC 2000], ST100 [ST Microelectronics 2004] and the Tangent A5 [ARC Cores 2005]. We term this feature as the "**r**educed bit-width **I**nstruction **S**et **A**rchitecture" (**rISA**).

Processors with rISA feature dynamically translate (or decompress, or expand) the narrow rISA instructions into corresponding normal instructions. This translation usually occurs before or during the decode stage. Typically, each rISA instruction has an equivalent instruction in the normal instruction set. This makes translation simple and can usually be done with minimal performance penalty. As the translation engine converts rISA instructions into normal instructions, no other hardware is needed to execute rISA instructions. If the whole program can be expressed in terms of rISA instructions, then up to 50% code size reduction can be achieved.

The fetch-width of the processor being the same, the processor when operating in rISA mode fetches twice as many rISA instructions (as compared to normal instructions) in each fetch operation. Thus, while executing rISA instructions, the processor needs to make lesser fetch requests to the instruction memory. This results in a decrease in power and energy consumption by the instruction memory subsystem.

Typically, each rISA instruction has an equivalent instruction in the normal instruction set. This makes the translation from rISA instructions to normal instructions very simple. Research [Benini et al. 2001; Lekatsas et al. 2000] have shown that the translation unit for a rISA design can be very small and can be implemented in a fast, power efficient manner. Furthermore, we have shown that significant energy savings achieved by executing rISA code [Shrivastava and Dutt 2004].

Thus, the main advantage of rISA lies in achieving low code size and low energy consumption with minimal hardware alterations. However, since more rISA instructions are required to implement the same task, rISA code has slightly lower performance compared to the normal code.

Although the theoretical limit of code size reduction achievable by rISA is 50%, severe restrictions on the rISA IS severely limit the compiler capability to achieve so. The rISA IS, because of bit-width restrictions, can encode only a small subset of the normal instructions. Thus, not all normal instructions can be converted into rISA instructions, and sometimes more than one rISA instruction may be the equivalent of one normal instruction. As a result, indiscriminate conversion to rISA instructions may result in an increase in code size. Again, due to bit-width restrictions rISA instructions can access only a restricted set of registers. For example, the ARM-Thumb allows access to 8 registers out of 16 general purpose ARM registers. As a result indiscriminate conversion of normal instructions to rISA instructions may result in register spilling, and thus degradation in performance and increase in code size. Furthermore, several options available in normal IS like predication and speculation may not be available in the rISA mode. Such severe restrictions on the rISA IS make the code-size reduction obtainable by using rISA very sensitive to the compiler quality, the application features and the rISA Instruction Set itself.

The compiler technology to generate code for rISA architectures is still primitive and does not consider all the trade-offs. To simplify the problem of code generation for a rISA processor, compilers usually perform the conversion at a routine-level granularity. In other words, all the instructions in a routine have to be in one mode, normal or rISA. This simplification misses out several significant opportunities to convert only the profitable regions of a routine.

This article makes three main contributions:

—We propose conversion to rISA instructions at a instruction-level granularity
—We present a novel compilation framework to exploit the conversion of instruction at instruction-level granularity, and show that our approach results in on average 22% code compression, which is much more than the achievable code compression by previous techniques.
—Finally, we use our retargetable code generation technique to explore several interesting rISA designs. Our analysis reveals that the code compression achieved by rISA is heavily dependent on the application characteristics and the rISA design itself. It is therefore very important for application specific processors to carefully design and tune the rISA.

The rest of the article is organized as follows: Section 2 discusses the hardware and software aspects of implementing a **r**educed bit-width **I**nstruction **S**et **A**rchitecture. Section 3 discusses the impact of these architectural features on the achievable code compression. Section 4 describes previous compilation techniques to exploit rISA. Section 5 describes our compilation framework to performs rISAization at instruction-level granularity to achieve high compression. Section 5.2 describes the key contribution of this article, that is, a register-pressure-based profitability heuristic to decide whether or not

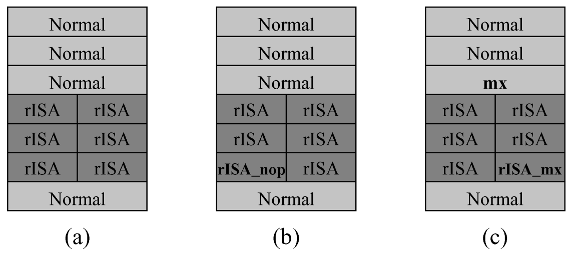

Fig. 1. Normal and rISA instructions co-exist in Memory.

to convert a sequence of normal instructions to rISA instructions. Converting at instruction-level granularity incurs a penalty in the form of extra instructions that are needed for explicit mode change. Section 5.4 first defines and then solves the problem of optimally inserting mode change instructions. In Section 6, we first show the efficacy of our compilation approach, and then perform exploration of several interesting rISA designs. We conclude with ongoing research directions in Section 7.

2. REDUCED BIT-WIDTH INSTRUCTION SET

A **rISA** processor is one which supports instructions from two different Instruction Sets. One is the "normal" 32-bit wide Instruction Set, and the other is the "narrow" 16-bit wide Instruction Set. The "narrow" instructions comprise the **r**educed bit-width **I**nstruction **S**et **rIS**. The code for a rISA processor contains both normal and rISA instructions, but the processor dynamically converts the rIS instructions to normal instructions, before or during the instruction decode stage.

2.1 rISA: Software Aspects

2.1.1 *Adherence to Word Boundary.* The code for rISA processors contains instructions from both the instruction sets, as shown in Figure 1(a). Many architectures impose the restriction that all code should adhere to the word boundary.

In order for the normal instructions to adhere to the word boundary, there can be only even number of contiguous rISA instructions. To achieve this, a rISA instruction that does not change the state of the processor is needed. We term such an operation as *rISA_nop*. The compiler can then pad odd-sized sequence of rISA instructions with *rISA_nop*, as shown in Figure 1(b). ARM-Thumb and MIPS32/16 impose such architectural restrictions, while the ARC processor can decode the instructions even if they cross the word boundary.

2.1.2 *Mode Change Instructions.* rISA processors operate in two modes, the normal mode and the rISA mode. In order to change the execution mode of a processor dynamically, there should be a mechanism in software to specify change in execution mode. For most rISA processors, this is accomplished using explicit mode change instructions. We term an instruction in the normal instruction set that changes mode from normal to rISA the *mx* instruction, and an instruction in the rISA instruction set that changes mode from rISA to

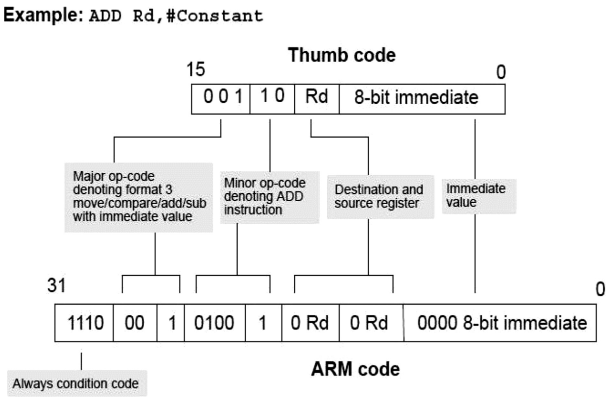
Example: ADD Rd,#Constant
Thumb code
15
0
0 0 1
1 0
Rd
8-bit immediate
Major op-code denoting format 3 move/compare/add/sub with immediate value
Minor op-code denoting ADD instruction
Destination and source register
Immediate value
31
0
1110
00
1
0100
1
0 Rd
0 Rd
0000 8-bit immediate
Always condition code
ARM code

Fig. 2. Simple translation of Thumb instruction to ARM instruction.

normal the *rISA_mx* instruction. The code including the mode change instructions is shown in Figure 1(c).

In ARM/Thumb, ARM instructions *BX* or *BLX* switch the processor to Thumb mode. The ARM *BX* instruction is a version of a ARM branch instruction, which changes the execution mode of the processor to rISA mode. Similarly, the ARM *BLX* instruction is a version of ARM *BL* instruction (Branch and Link), with the additional functionality of switching the processor to rISA mode. Similar earmarked instructions also exist in the Thumb instruction set to switch the processor back to the normal mode of operation. The MIPS16 ISA has an interesting mechanism for specifying mode changes. All routines encoded using MIPS16 instructions begin at the half word boundary. Thus, calls (and returns) to half word aligned addresses change the mode from normal to rISA. The ARC Tangent A5 processor on the other hand allows native execution of the ARCompact instructions. No special instruction is required to switch modes.

2.2 rISA: Hardware Aspects

Although the code for a rISA processor contains instructions from both the Instruction Sets, the instruction fetch mechanism of the processor is oblivious of the presence of rISA instructions. The fetched code is interpreted (decoded) as normal or rISA instruction depending on the operational mode of the processor. When the processor is in rISA mode, the fetched code is assumed to contain two rISA instructions. The first one is translated into normal instruction, while the second one is latched and kept for the next cycle of execution.

Figure 2 shows an example of translation of a Thumb-ADD instruction to a normal ARM-ADD instruction. The translation can be realized in terms of simple and small table look-ups. Since the conversion to normal instructions is done during or before the instruction-decode stage, the rest of the processor remains same. To provide support for rISA typically only decode logic of the processor needs to be modified. The rISA instructions can be decoded by either single-step decoding or two-step decoding.

Figure 3 shows the one-step decoding of rISA instructions. The single step decoding of rISA instructions is just like performed at the same time as the

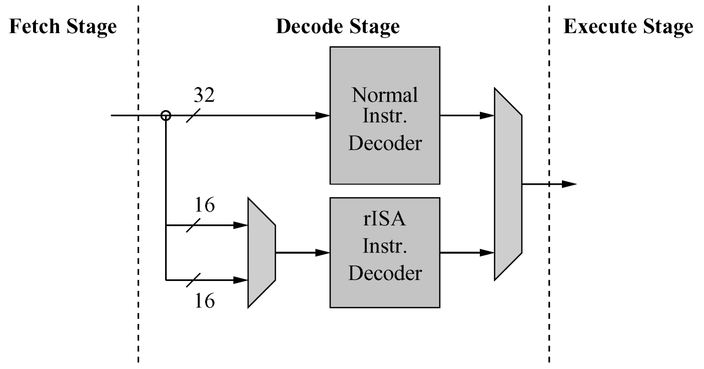

Fig. 3. Single-step decoding of rISA instructions.

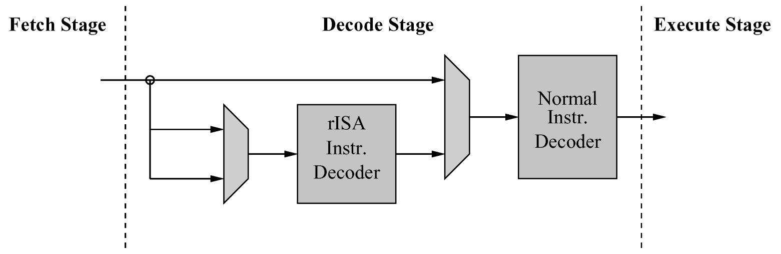

Fig. 4. Two-step decoding of Thumb instructions.

decoding of the normal instructions. Single-step decoding is typically simpler because of the rISA instructions are "narrow".

In the two-step decoding shows in Figure 4, the rISA instructions are first translated to normal instructions. The normal instructions can then be decoded as before. Although such implementation requires minimal logical changes to the existing architecture, it may lengthen the cycle time. ARM7TDMI implements two step decoding approach of Thumb instructions.

3. COMPILATION ISSUES FOR RISA

Although up to 50% code compression can be achieved using rISA, owing to severe restrictions on the number of operations, register accessibility and reduced functionality, it is in practice tough to consistently achieve more than 30% code size reduction. In order to alleviate such severe constraints, several solutions have been proposed. We discuss several such architectural features in the light of aiding code generation in rISA processors.

3.1 Granularity of rISAization

We term the compiler-process of converting normal instructions into rISA instructions as **rISAization**. Existing compilers like the ARM-Thumb compiler (as yet) supports the conversion at a routine level granularity. Thus, all the instructions in a routine can be in exactly one mode, the normal ARM mode, or the Thumb mode. A routine cannot have instructions from both the

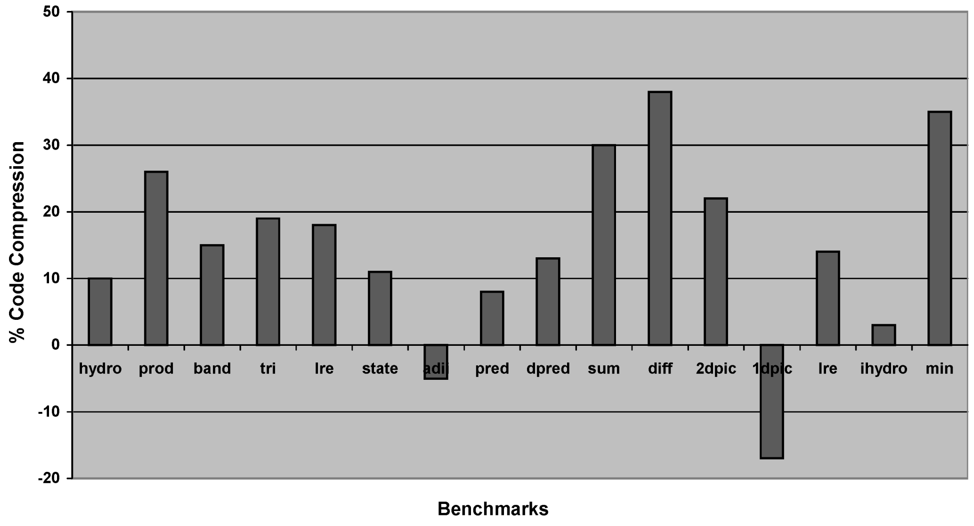

Fig. 5. Code compression achieved by routine-level rISAization.

ISAs. Furthermore, existing Compilers rely on human analysis to determine which routines to implement in ARM instructions, and which ones in Thumb instructions.

Figure 5 plots the code compression achieved by MIPS32/16 compiler performing indiscriminate rISAization. The compiler could achieve 38% code compression on *diff* benchmark, that has only one routine and low register pressure. All the instructions could be mapped to rISA instructions. However, on *1dpic*, the most important routine had high register pressure, that resulted in register spilling and thus leading to an increase in code size. The case with *adii* is similar. Some other benchmarks had routines in which the conversion resulted in an increase in the code size. Thus, although routine-level granularity of rISAization can achieve high degrees of code compression on small routines, it is unable to achieve decent code compression on application level. In fact, MIPS32/16 compiler could achieve only 15% code size reduction on our set of benchmarks. This is because not many routines can be found whose conversion results in substantial code compression. The inability to compress code is due to two main reasons: (i) rISAization may result in an increase in code size and (ii) rISAization of routines that have high register pressure results in register spills and thus increase in the code.

Figure 6 explains the two major drawbacks of rISAizing at routine-level granularity and shows that instruction-level granularity alleviates these drawbacks.

First, a routine-level granularity approach misses out on the opportunity to rISAize code sections inside a routine, which is deemed non profitable to rISAize. It is possible that it is not profitable to rISAize a routine as a whole, but some parts of it can be profitably rISAized. For example, the *Function* 1 *and Function* 3 are found to be nonprofitable to rISAize as a whole. Traditional routine-level granularity approaches will therefore not rISAize these routines,

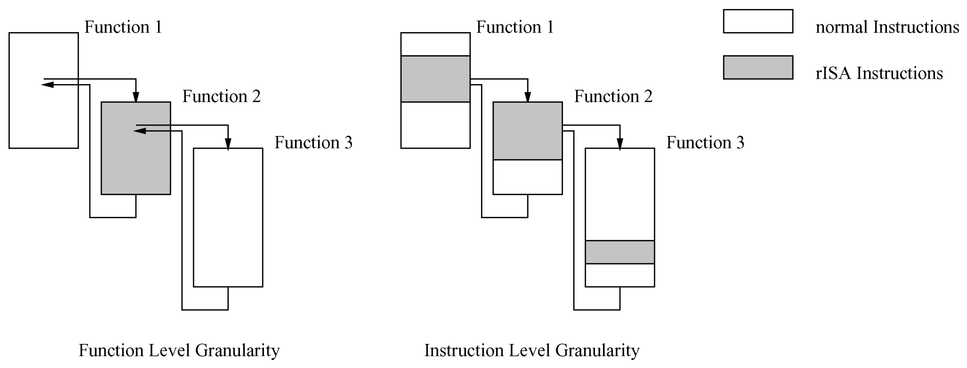

Fig. 6. Routine level granularity versus instruction level granularity.

while instruction-level granularity approaches will be able to achive some code size reduction by identifying and rISAizing only some profitable portions of the routine.

Second, the compiler is not able to leave out some regions of code inside a routine that may incur several register spills. It is possible that leaving out some peices of code inside a profitable routine may increase the code compression achieved. For example, in Figure 6, the instruction-level granularity approaches have the choice to leave out some regions of code inside a routine to achieve higher code compression.

Thus, consistently high degree of code compression can be achieved by rISAization at instruction level granularity. However, instruction-level granularity of rISAization has some overheads. Instruction-level granularity of rISAization needs explicit mode change instructions. We define a normal instruction *mx* changes the execution mode of the processor from normal to rISA, while a rISA instruction *rISA_mx* changes the execution mode of the processor from rISA mode to normal mode. The mode change instructions have to be inserted in the code at the boundaries of normal and rISA instructions. Unlike the conversion at routine level granularity, this causes an increase in code size. But our results show that, even with this code-size penalty, consistently higher degrees of code compression can be achieved by rISAization at instruction-level granularity.

3.2 rISA Instruction Set

The rISA instruction set is tightly constrained by the instruction width. Typical instruction sets have three operand instructions; two source operands and one destination operand. Only 16 bits are available to encode the opcode field and the three operand fields. The rISA design space is huge, and several instruction set idiosyncrasies makes it very tough to characterize. As shown in Figure 7, we informally define *rISA_wxyz* as a rISA instruction set with opcode width w-bits, and three operands widths as x-bit, y-bit, and z-bits respectively. Most interesting rISA instruction sets are bound by *rISA_7333* and *rISA_4444*.

The *rISA_7333* format describes an instruction set in which the opcode field is 7-bit wide, and each operand is 3-bit wide. Such an instruction set would contain 128 instructions, but each instruction can access only 8 registers. Although such

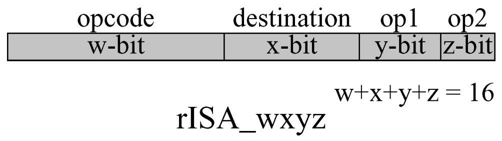

Fig. 7. rISA instruction format.

a rISA instruction set can rISAize large portions of code, but register pressure may become too high to achieve a profitable encoding. On the other extreme is the *rISA_4444* format, which has space for only 16 instructions, but each instruction can access up to 16 registers. For applications that do not use a wide variety of instructions, but have high register pressure, such a rISA instruction set is certainly a good choice.

The design space between the two extremes is huge. All realistic rISA instruction sets contain a mix of both type of instructions, and try to achieve the "best of both worlds". Designing a rISA instruction set is a essentially a trade-off between encoding more instructions in the rISA instruction set, and providing rISA instructions access to more registers.

The "implicit operand format" of rISA instructions is a very good example of the trade-off the designers have to make while designing rISA. In this feature, one (or more) of the operands in the rISA instruction is hard-coded (i.e., implied). The implied operand could be a register operand, or a constant. In case a frequently occurring format of add instruction is *add* R_i R_i R_j (where the first two operands are the same), a rISA instruction *rISA_add1* R_i R_j, can be used. In case an application that access arrays produces a lot of instructions like $addr = addr + 4$, then a rISA instruction *rISA_add4 addr* which has only one operand might be very useful. The translation unit, while expanding the instruction, can also fill in the missing operand fields. This is a very useful feature that can be used by the compiler to generate good quality code. ARC Tangent A5 processor uses this feature extensively to optimize ARCompact instruction set [ARC Cores 2005].

3.3 Limited Access to Registers

rISA instructions usually have access to only a limited set of processor registers. This results in increased register pressure in rISA code sections. A very useful technique to increase the number of useful registers in rISA mode is to implement a *rISA_move* instruction that can access all registers. This is possible because a move operation has only two operands and hence has more bits to address each operand.

3.4 Limited Width of Immediate Operands

A severe limitation of rISA instructions is the inability to incorporate large immediate values. For example, with only 3 bits are available for operands, the maximum unsigned value that can be expressed is 7. Thus, it might be useful to vary the size of the immediate field-depending on the application and the values that are (commonly) generated by the compiler. Increasing the size of the immediate fields, however, reduces the number of bits available for opcodes (and also the other operands). Several architectures implement a *rISA_extend* instruction, which extends the immediate field in the next rISA instruction. Such an instruction is very useful to be able to rISAize contiguous large portions of code.

4. RELATED WORK

Many leading 32-bit embedded processors also support the 16-bit (rISA) instruction set to address both memory and energy consumption concerns of the embedded domain [Advanced RISC Machines, Ltd. 2003; LSI LOGIC 2000; ST Microelectronics 2004; ARC Cores 2005].

There has been previous research effort to achieve further code compression with the help of architectural modifications to rISA. The ARM Thumb IS was redesigned by Kwon et al. [1999a] to compress more instructions and further improve the efficiency of code size reduction. This new Instruction Set is called Partitioned Register Extension (PARE), reduces the width of the destination field and uses the saved bit(s) for the immediate addressing field. The register file is split into (possibly overlapping) partitions, and each 16-bit instructions can only write to a particular partition. This reduces the number of bits required to specify the destination register. With a PARE-aware compiler, the authors claim to have achieved a compression ratio comparable to Thumb and MIPS16.

Another direction of research in rISA architectures has been to overcome the problem of decrease in performance of rISA code. Kwon et al. [1999b] proposed a parallel architecture for executing rISA instructions. TOE(Two Operation Execution) exploits Instruction Level Parallelism provided by the compiler. In the TOE architecture all rISA instruction occur in pairs. With 1-bit specifying the eligibility of the pair of rISA instructions to execute in parallel, the performance of rISA can be improved. Since the parallelization is done by the compiler, the hardware complexity remains low.

Krishnaswamy and Gupta [2003] observed that there exist Thumb instruction pairs that are equivalent to single ARM instructions throughout the 16-bit Thumb code. They enhanced the Thumb instruction set by an AX (Augmenting Extensions) instructions. Compiler finds the pairs of Thumb instructions that can be safely executed as single ARM instruction, and replace them by AX+Thumb instruction pairs. They coalesce the AX with the immediately following Thumb instruction at decode time and generate an ARM instruction to execute single instruction, thus increasing performance.

While there has been considerable research in the design of architectures/architectural features for rISA, the compiler techniques employed to generate code targeted for such architectures are rudimentary. Most existing compilers either rely on user guidance or perform a simple analysis to determine which routines of the application to code using rISA instructions. These approaches, which operate at the routine level granularity, are unable to recognize opportunities for code size optimization within routines. We propose instruction-level granularity of rISAization and present compiler frmaework to generate optimized code for such rISA architectures. Our technique, is able to aggressively reduce code size by discovering codesize reduction opportunities inside a routine, resulting in high degrees of code compression.

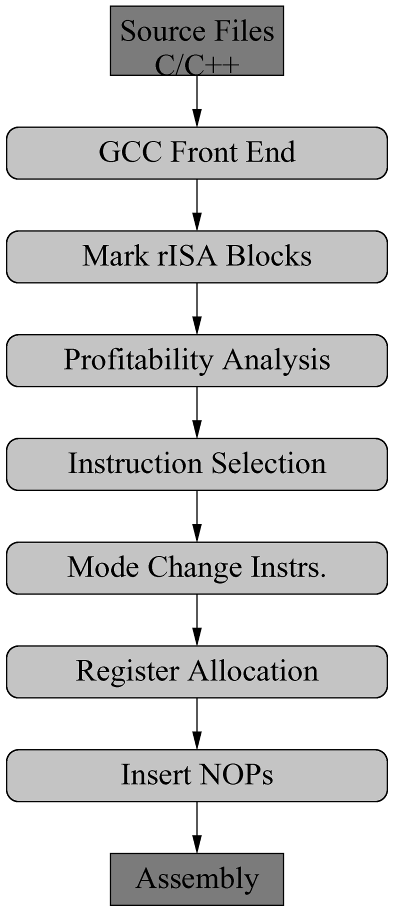

Fig. 8. Compilation steps for rISA.

5. COMPILATION FRAMEWORK FOR rISA

We implemented our rISA compiler technique in the EXPRESS retargetable compiler. EXPRESS [Halambi et al. 2001] is an optimizing, memory-aware, Instruction Level Parallelizing (ILP) compiler. EXPRESS uses the EXPRESSION ADL [Halambi et al. 1999] to retarget itself to a wide class of processor architectures and memory systems. The inputs to EXPRESS are the application specified in C, and the processor architecture specified in EXPRESSION. The front-end is GCC based and performs some of conventional optimizations. The core transformations in EXPRESS include **RDLP** [Novack and Nicolau 1997]—a loop pipelining technique, **TiPS** : Trailblazing Percolation Scheduling [Nicolau and Novack 1993]—a speculative code motion technique, Instruction Selection and Register Allocation. The back-end generates assembly code for the processor ISA.

A rISA compiler not only needs the ability to selectively convert portions of application into rISA instruction, but also heuristics to perform this conversion only where it is profitable. Figure 8 shows the phases of the EXPRESS compiler with our rISAization technique. The code generation for rISA processors in EXPRESS is therefore a multi-step process:

5.1 Mark rISA Blocks

Various restrictions on the rISA instruction set, means that several normal instructions may not be convertible into rISA instructions. For example, an instruction with a large immediate value may not be rISAizable. The first step in compilation for a rISA processor is thus marking all the instructions that can be converted into rISA instructions. However, converting all the marked instructions into rISA instructions may not be profitable, because of the overhead associated with rISAization. The next step is therefore, to decide which contiguous list of marked instructions are profitable to convert to rISA instructions. Note that a list of contiguous marked instructions can span across basic block boundaries. To ensure correct execution, mode change instructions need to be added, so that the execution mode of processor (normal or rISA) matches that of the instructions it is executing. The instruction selection, however is done within basic blocks. Contiguous list of marked instructions in a basic block is termed **rISABlock**.

5.2 Profitability Heuristic for rISAization

Even though all the instructions in a rISABlock, can be rISAized, it may not be profitable (in terms of code compression, or performance) to rISAize the rISABlock. For example, if a rISABlock is very small, then the mode change instruction overhead could outshine any code compression achievable by rISAization. Similarly if the rISABlock is very big, the increase register pressure (and register spilling therefore) could make rISAization the rISABlock a bad idea. Thus an accurate estimation of code size and performance trade-off is necessary before rISAizing a rISABlock. In our technique, the impact of rISAization on code size and performance is estimated using a profitability analysis (PA) function. The PA function estimates the difference in code size (CS) and performance (PF) if the block were to be implemented in rISA mode as compared to normal mode. The compiler (or user) can then use these estimates to trade-off between performance and code size benefits for the program. Next, we describe how the PA function measures the estimated impact on code size and performance.

5.2.1 *Code Size (CS).* Figure 9 shows the portion of the PA function that estimates the code size reduction due to rISA. Ideally, converting a block of code to rISA instructions reduces the size of the block by half. However, the conversion typically incurs an overhead that reduces the amount of compression. This overhead is composed of three factors:

Mode Change Instructions ($CS1$). Before every block of rISA instructions, a *mx* (Mode Change from normal to rISA) instruction is needed. This causes an increase in code size by one full length instruction. At the end of every rISA block, a *rISA_mx* (Mode Change from rISA to normal) instruction is needed, causing an increase in code size by the size of the rISA instruction. Thus, for an architecture with normal instruction length of 4 bytes and rISA instruction of 2 bytes, $CS1 = 4 + 2 = 6$ bytes.

Estimate_CodeSize_Reduction

```
01: **CS1** $= sizeof(mx) + sizeof(rISA\_mx)$
02: **CS2** $= sizeof(rISA\_nop)$
03: **CS3** $= Extra\_Spill\_Reload\_Estimate(bl)$
04: **return** $sizeBlock(bl, NORMAL) - sizeBlock(bl, rISA) - CS1 - CS2 - CS3$
```

Extra_Spill_Reload_Estimate(Block bl)

```
05:   // Estimate spill code if the block is rISAized
06:   $extra\_rISA\_reg\_press = Avg\_Reg\_Press(bl, rISA\_vars) - K1 \times rISA\_REGS$
07:   **if** $(extra\_rISA\_reg\_press > 0)$
08:     $avail\_non\_rISA\_regs = TOTAL\_REGS - rISA\_REGS$
09:     $rISA\_spills = \frac{extra\_rISA\_reg\_press \times bl.num\_instrs}{Avg\_Live\_Len(bl)}$
10:   **else**
11:     $avail\_non\_rISA\_regs = TOTAL\_REGS - rISA\_REGS - extra\_rISA\_reg\_press$
12:     $rISA\_spills = 0$
13:   **endif**

14:   $extra\_non\_rISA\_reg\_press = Avg\_Reg\_Press(bl, non\_rISA\_vars)$
14:                                   $-K1 \times avail\_non\_rISA\_regs$
15:   **if** $(extra\_non\_rISA\_reg\_press > 0)$
16:     $non\_rISA\_spills = extra\_non\_rISA\_reg\_press$
17:   **else**
18:     $non\_rISA\_spills = 0$
19:   **endif**

20:   **spill_code_if_rISA** $= rISA\_spills \times SIZE\_rISA\_INSTR$
20:                                   $+ non\_rISA\_spills \times SIZE\_NORMAL\_INSTR$

21:   // Estimate spill code if the block is NOT rISAized
22:   $extra\_normal\_reg\_press = Avg\_Reg\_Press(bl, all\_vars) - K1 \times TOTAL\_REGS$
23:   **if** $(extra\_normal\_reg\_press > 0)$
24:     $normal\_spills = \frac{extra\_normal\_reg\_press \times bl.num\_instrs}{Avg\_Live\_len(bl)}$
25:   **else**
26:     $normal\_spills = 0$
27:   **endif**

28:   **spill_code_if_normal** $= normal\_spills \times SIZE\_NORMAL\_INSTR$
29:   **extra_spill_code** $= spill\_code\_if\_rISA - spill\_code\_if\_normal$
30:   **extra_reload_code** $= K2 \times extra\_spill\_code \times Avg\_Uses\_Per\_Def$
31:   **return** $extra\_spill\_code + extra\_reload\_code$
```

Parameters

```
32:   TOTAL_REGS = 16, rISA_REGS = 8
33:   SIZE_rISA_INSTR = 2 bytes, SIZE_NORMAL_INSTR = 4 bytes
34:   K1 and K2 are control constants
```

Fig. 9. Profitability heuristic for rISAization.

NOP (CS2). Most architectures require that normal instructions be aligned at word boundaries. However, rISAizing a block with odd number of instructions[1] will cause the succeeding normal instruction to be misaligned. In such cases, an extra *rISA_nop* (No-operation instruction) needs to be added

[1]Including the rISA_mx instruction.

inside the rISA block. We conservatively estimate that each rISA block needs a rISA_nop instruction. $CS2 = 2$ bytes.

Spills/Reloads (CS3). Due to limited availability of registers, rISAizing a block may require a large amount of spilling (either to memory or to non-rISA registers). As this greatly impacts both code size and performance it is important to accurately estimate the number of spills (and reloads) due to rISAization. The PA function estimates the number of spills and reloads due to the rISA block by calculating the average register pressure[2] due to the variables in the block.

The first step is to calculate the amount of spill code inserted if the block is rISAized (line 20 in Figure 9). The block may contain variables that need to be allocated to the rISA register set and variables that can be allocated to any registers. Thus, rISA spill code is estimated as the total of spills due to rISA variables (lines 05–13) and spills due to non-rISA variables (lines 14–19). The constant *K1* can be used to control the importance of spill code in estimation.

The function *Avg_Reg_Press* returns the average register pressure for variables of a particular type (rISA or non-rISA) in a block. The function *Avg_Live_Len* returns the average distance between the definition of a variable in a block and its last use (i.e., its lifetime). In a block, the extra register pressure (that causes spilling) is the difference between Avg_Reg_Press and the number of available registers (lines 06, 14, 22). Each spill reduces the register pressure by 1 for the life time of the variable. So, a block with size *num_instrs* requires *num_instrs/Avg_Live_Len* spills to reduce the register pressure by 1. Thus, the number of spills required to mitigate the register pressure is equal to the extra register pressure multiplied by the number of spills required to reduce register pressure by 1 (lines 09, 16, 24).

The next step is to estimate the total number of spills if the block is not converted to rISA instructions (line 28). This is accomplished in a manner similar to that of estimation of rISA variables.

As each spill also requires reloads to bring the variable to a register before its use, it is necessary to also calculate the number of extra reloads due to conversion to rISA. The PA function estimates the number of reloads as a factor of number of spills in the rISA Block. The constant *K2* can be used to control the importance of reload code in estimation.

The total reduction in code size of the block due to rISAization (line 04) is $CS = 2 \times \mathit{NumInstrs(rISABlock)} - CS1 - CS2 - CS3$. A CS value greater than zero implies that converting the block to rISA instructions is profitable in terms of code size.

5.2.2 *Performance (PF).* The impact of converting a block of instructions into rISA on performance is difficult to estimate. This is especially true if the architecture incorporates a complex instruction memory hierarchy (with caches). Our technique makes a crude estimate of the performance impact based on the latency of the extra instructions (due to the spills/reloads, and due to the mode change instructions). A more accurate estimate can be made by also

[2]Register Pressure is defined as the number of variables live at the point in the program.

considering the instruction caches and the placement of the blocks in program memory.

5.3 Instruction Selection

EXPRESS uses a tree-pattern-matching-based algorithm for Instruction Selection. A tree of generic instructions is converted to a tree of target instructions. In case a tree of generic instructions can be replaced by more than one target instruction tree, the one with lower cost is selected. The cost of a tree depends upon the user's relative importance of performance and code-size. Our approach towards compiling for rISA, looks at the rISA conversion as a natural part of the Instruction Selection process. The Instruction Selection phase uses a profitability heuristic to guide the decisions of which section of a routine to convert to rISA instructions, and which ones to normal target instructions. All generic instructions within profitable rISABlocks are replaced with rISA instructions and all other instructions are replaced with normal target instructions. Replacing a generic instruction with a rISA instruction involves both selecting the appropriate rISA opcode, and also restricting the operand variables to the set of rISA registers.

5.4 Inserting Mode Change Instructions

After Instruction Selection, the program comprises of sequences of normal and rISA instructions. A list of contiguous rISA instructions may span across basic block boundaries. To ensure correct execution, we need to make sure that any possible execution path, whenever there is a switch in the instructions from normal to rISA or vice-versa, there is an explicit and appropriate mode change instruction. There should be a *mx* instruction when the instructions change from normal to rISA instructions, and a *rISA_mx* instruction when the instructions change from rISA instructions to normal instructions.

If the mode of instructions change inside a basic block, then there is no choice but to add the appropriate mode change instruction at the boundary. However, when the mode changes at basic block boundary, the mode change instruction can be added at the beginning of the successor basic block or at the end of the predecessor basic block. The problem becomes more complex if there are more than one successors and predecessors at the junction. In such a case, we want to add the mode change instructions so as to minimize the performance degradation. So we want to add them so that they will be executed the least. We use profile information to find out the execution counts of the basic block and then solve the optimality problem.

To further motivate the problem, consider two consecutive basic blocks, b_i and b_j. Suppose b_i is the successor of b_j. Further suppose that the end mode of b_i and the start mode of b_j, is the same, then there is no need to add a mode change instruction. However, if there is another execution path from basic block b_k to b_j, and the last instruction of b_k is of a different mode, then explicit mode change instructions need to be inserted.

There are two choices to insert the mode change instruction, we can either insert the mode change instruction as the last instruction of b_k, or as the first instruction of b_j. In the second solution, we will have to insert a mode change

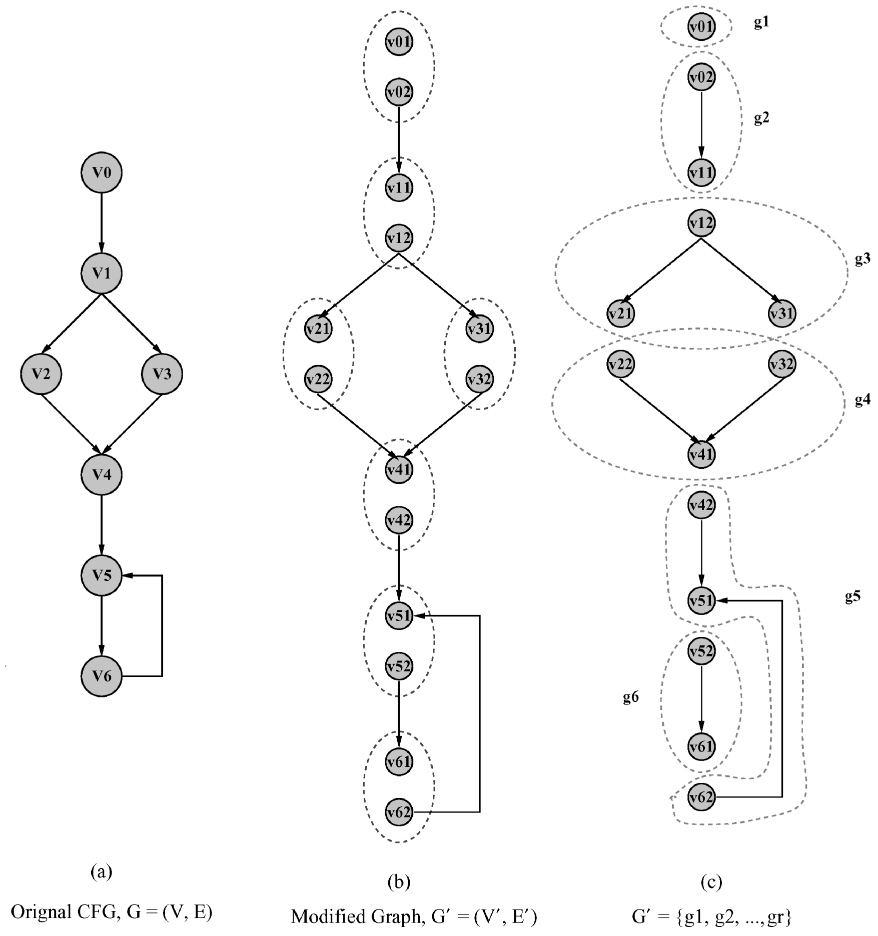

Fig. 10. Mode change instruction insertion.

instruction as the last instruction in b_i too. Thus, the first solution seems to be the winner. However, if execution frequency of b_k is greater than sum of execution frequencies of b_i and b_j, then first solution results in more increase in *Dynamic Code Size*. In general, there can be many control flow edges coming in b_j, and going out of b_k, making the problem more complex.

Along every possible execution path, whenever instructions change from normal to rISA, or otherwise, there should be an explicit mode change instruction. The cost of inserting a mode change instruction is equal to the execution frequency of the basic block. The problem is thus to insert mode change instructions with least cost.

The insertion of mode change instructions is performed in two steps. If mode change occurs inside a basic block, corresponding mode change instructions are inserted at the boundary of rISA Block.

After the first step, the CFG (Control Flow Graph) can be visualized as a directed graph $G = (V, E)$, where V represents the basic blocks, and E represent the Control Flow edges as shown in Figure 10(a). G has two distinguished vertices, the start vertex v_0 and the end vertex v_n. Three functions are thus defined on V,

—*ExecFrequency*: $V \rightarrow N$ gives the execution frequency for each vertex.

—*EntryMode* : $V \rightarrow \{Normal, rISA\}$ gives entry mode of the basic block represented by the vertex is Normal or rISA. *EntryMode*(V_i) is *rISA* if the first instruction of the basic block is a rISA instruction. Otherwise, it is *Normal*.

—*ExitMode*: $V \rightarrow \{Normal, rISA\}$. *ExitMode*$(V_i)$ is *rISA* if the last instruction of the basic block is a rISA instruction. Otherwise, it is *Normal*.

We get *ExecFrequency* for each basic block from the profile information. The functions *EntryMode* and *ExitMode* are computed for each basic block.

We can switch the *EntryMode*, or *ExitMode* of a vertex by inserting a mode change instruction at the start of the basic block, or at the end of the basic block respectively. However, switching the *EntryMode* or *ExitMode* of the vertex v_i costs *ExecFrequency*(v_i).

The problem of mode change instruction insertion is to find *EntryMode* and *ExitMode* for each vertex so that, for each edge $(v_i, v_j) \in E$,

$$ExitMode(v_i) == EntryMode(v_j)$$

such that the switching cost is minimized. The switching cost essentially represents the *Dynamic Code Size*.

To solve this problem, we transform our graph G. We break each vertex v_i into two vertices, v_{i1} and v_{i2} in graph G' as shown in Figure 10(b). Vertex v_{i1} represents the entry of v_i, and v_{i2} represents the exit of v_i. All the incoming edges into v_i now come to v_{i1}, and all the outgoing edges from v_i, now emanate from v_{i2}. Two functions are defined on vertices of G',

—*ExecFrequency*(v_{ij}) = *ExecFrequency*(v_i)

—*Mode*(v_{i1}) = *EntryMode*(v_i)

—*Mode*(v_{i2}) = *ExitMode*(v_i)

The new Graph $G' = (V', E')$ is a forest of connected components. Our problem now reduces into finding $Mode$ for each vertex so that all the vertices in a connected component have the same mode.

We identify all the connected components of $G' = \{g_1, g_2, \ldots, g_k\}$, as depicted in Figure 10(c). Each connected component is a subgraph $g_i = (V_i, E_i)$, containing a subset of vertices,

$$V_i \subset V', V_i = \{u_1, u_2, \ldots, u_r\}.$$

These vertices are partitioned into two (possibly empty) sets V_{Normal} and V_{rISA}.

$V_i = V_{Normal} \bigcup V_{rISA}$, and $V_{Normal} \bigcap V_{rISA} = \phi$.

Cost of converting all vertices to rISA Mode =
$\Sigma_{i=1..|V_{Normal}|} ExecFrequency(u_i)$.

Cost of converting all vertices to Normal Mode =
$\Sigma_{i=1..|V_{rISA}|} ExecFrequency(u_i)$.

We pick the lower cost conversion and thus decide upon the *Mode* of each vertex in G' and hence *EntryMode* and *ExitMode* of each vertex in G. Finally, we insert the appropriate mode change instructions. To change the *EntryMode* of a basic block from Normal to rISA, we add a *mx* instruction as the first instruction of the basic block. To change the *EntryMode* of a basic block from rISA to Normal, we add a *rISA_mx* instruction as the first instruction of the basic block. To change the *ExitMode* of a basic block from Normal to rISA, we add a *mx* instruction as the last or second last instruction of the basic block. To change the *ExitMode* of a basic block from rISA to Normal, we add a *rISA_mx* instruction as the last or second last instruction of the basic block.

Note that, if the last instruction of a basic block is a branch operation, and the machine does not have a delay slot, then the mode change instruction has to be added as the second last instruction in the basic block.

5.5 Register Allocation

The actual register allocation of variables is done during the Register Allocation phase. The EXPRESS compiler implements a modified version of Chaitin's solution [Briggs et al. 1994] to Register Allocation. Since code blocks that have been converted to rISA typically have a higher register pressure (due to limited availability of registers), higher priority is given to rISA variables during register allocation.

5.6 Insert NOPs

The final step in code generation is to insert *rISA_nop* instruction in *rISABlocks* that have odd number of rISA instructions. This is a straightforward step, and we do not discuss it.

6. EXPERIMENTS

In this section, we first demonstrate the efficacy of our compilation scheme over an existing rISA architecture. We show that the register-pressure-based profitability function to decide which regions of code to rISAize performs good consistently. We then design several interesting rISA design points, and study the code compression obtained by using them. We show that the code compression achieved is very sensitive to the rISA design, and that a custom rISA can be designed for a set of applications to achieve high code compression.

We perform our experiments over MIPS32/16 ISA on a set of applications from numerical computation kernels (Livermore Loops), and DSP application kernels, that are often executed in embedded processors.

6.1 Compiler Comparison

Our first experiment is to study the efficacy of our compilation technique. We compare the code compressions we achieve with the code compression GCC can achieve using the MIPS32/16 rISA. To this effect, we first compile the applications using GCC for MIPS32 ISA. We then compile the applications using GCC for MIPS32/16 ISA. We perform both the compilations using-Os flags with the GCC to enable all the code size optimizations. The percentage code compression

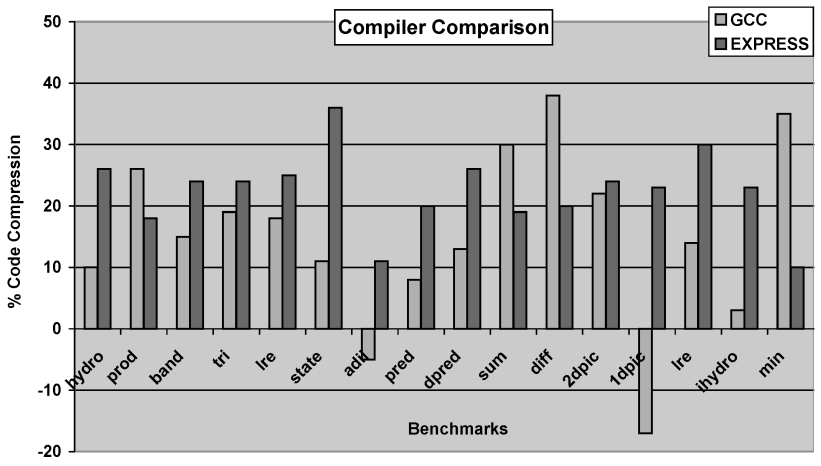

Fig. 11. Percentage code compressions achieved by GCC and EXPRESS for MIPS32/16.

achieved by GCC for MIPS16 is computed and is represented by the light bars in Figure 11. The MIPS32 code generated by GCC is compiled again using the register pressure based heuristic in EXPRESS. The percentage code compression achieved by EXPRESS is measured and plotted as dark bars in Figure 11.

It can be clearly seen from Figure 11 that the register-pressure-based heuristic performs consistently better than GCC and successfully prevents code inflation. GCC achieves, on an average, 15% code size reduction, while EXPRESS achieved an average of 22% code size reduction. We used SIMPRESS [Khare et al. 1999], a cycle accurate simulator, to measure the performance impact due to rISAization. We simulated the code generated by EXPRESS on a variant of the MIPS R4000 processor that was augmented with rISA MIPS16 Instruction Set. The memory subsystem was modeled with no caches and a single-cycle main memory. The performance of MIPS16 code is, on an average, 6% lower than that of MIPS32 code, with the worst case being 24% lower. Thus, our technique is able to reduce the code size using rISA with a minimal performance impact.

6.2 Sensitivity on rISA Designs

Due to highly constrained design of rISA, the code compression achieved is very sensitive to the rISA chosen. rISA design space is huge and several instruction set idiosyncrasies make it very tough to characterize. To show the variation of code compression achieved with rISA, we take a designer's approach. We systematically design several rISA, and study the code compression achieved by them. We start with the extreme rISA designs of *rISA_7333* and *rISA_4444* and gradually improve upon them.

The first rISA design point is (*rISA_7333*). In this rISA, the operand is represented by 7-bits, and each operand is encoded in 3-bits. Thus, there can be 2^7 instructions in this rISA, but each instruction can have access to only 8 registers, or be a constant that can be represented in 3-bits. However, instructions that have 2 operands (like move) have 5-bit operands; thus, they can access 32

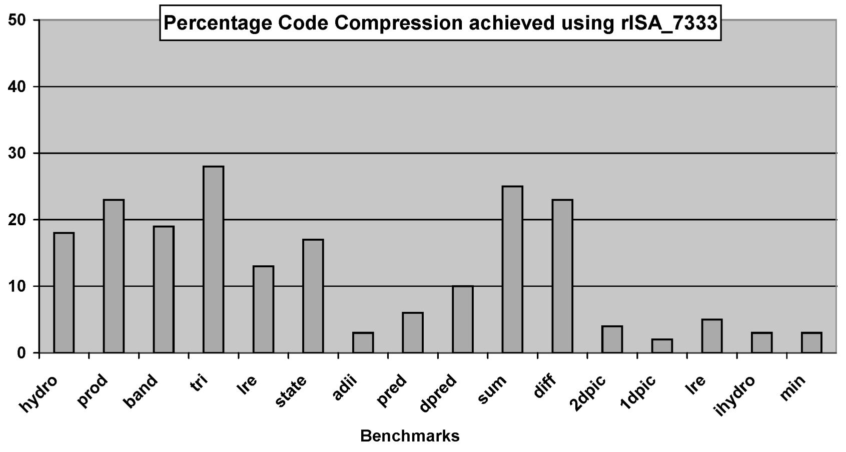

Fig. 12. Percentage code compression achieved by using *rISA_7333*.

(= all the registers in our architecture model) registers. Owing to the uniformity in the instruction format, the translation unit is very simple for this rISA design.

Figure 12 shows that the *rISA_7333* design on an average achieves 12% code compression. EXPRESS is unable to achieve good code compressions for applications that have high register pressure, for example, *adii*, and those with large immediate values. In such cases, the compiler heuristic decides not to rISAize large portions of the application to avoid code size increase due to extra spill/reload and immediate extend instructions.

The first limitation of *rISA_7333* is overcome in the second rISA design, that is, *rISA_4444*, which takes us to the other limit. In this rISA, the opcode as well as each operand is encoded in 4-bits. Although only 16 instructions are allowed in such a rISA design, it allows each operand to access 16 registers. We profiled the applications and incorporated the 16 most frequently occurring instructions in this rISA.

Figure 13 shows that the register pressure problem is mitigated in the *rISA_4444* design. It achieves better code-size reduction for benchmarks that have high register pressure, but performs badly on some of the benchmarks because of its inability to convert all the normal instructions into rISA instructions. *rISA_4444* achieves about 22% improvement over normal instruction set.

We now attack the second problem faced in *rISA_7333* (small immediate values). For instructions that have immediate values, we decrease the size of opcode, and use the bits to accommodate as large an immediate value as possible. This design point is called *rISA_7333_imm*. Because of the nonuniformity in the size of the opcode field, the translation logic is complex for such a rISA design.

As Figure 14 shows, the *rISA_7333_imm* design achieves slightly better code compressions as compared to the first design point since it has large immediate fields, while having access to the same set of registers. *rISA_7333_imm* achieves about 14% improvement over normal instruction set.

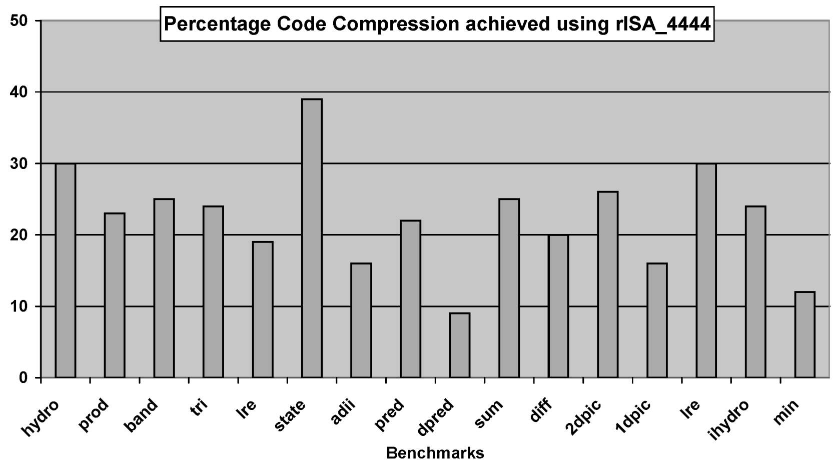

Fig. 13. Percentage code compression achieved by using *rISA_4444*.

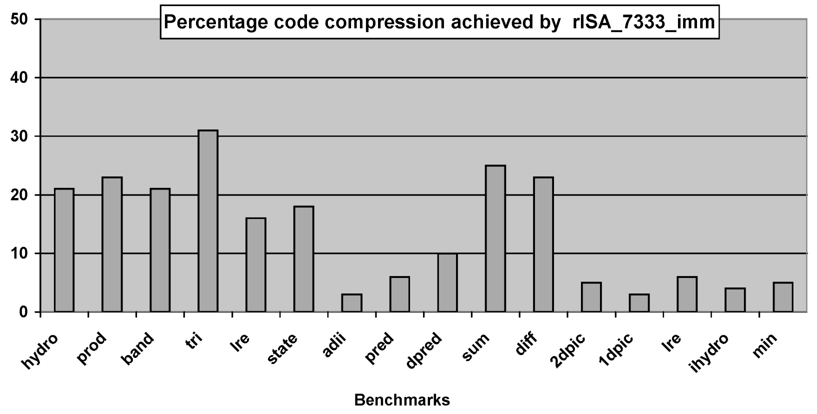

Fig. 14. Percentage code compression achieved by using *rISA_7333_imm*.

Another optimization that can be performed to save precious bit-space is to encode instructions with same operands with different opcode. Since fewer operands are required to express such an instruction, it requires fewer instruction bits, which can be used in two ways: first is the direct way by providing increased register file access to the remaining operands, and second (a more indirect way) is that this instruction can afford a longer opcode, another instruction which has tighter constraints on the opcode field (e.g., an instruction with immediate operands) can switch opcode with this instruction. We apply the implied operands feature in *rISA_7333* and obtain our forth rISA design that is, *rISA_7333_imp_opnd*. This rISA design matches the MIPS16 rISA.

Figure 15 plots the code compression achieved by *rISA_7333_imp_opnd*. The graphs show that the *rISA_7333_imp_opnd* on average achieves about the same code size improvement as the *rISA_4444* design. Note that the performance benefits of using implicit operands is substantial for some applications such as *state* and *dpred*. *rISA_7333_imp_opnd* achieves about 22% improvement over normal instruction set.

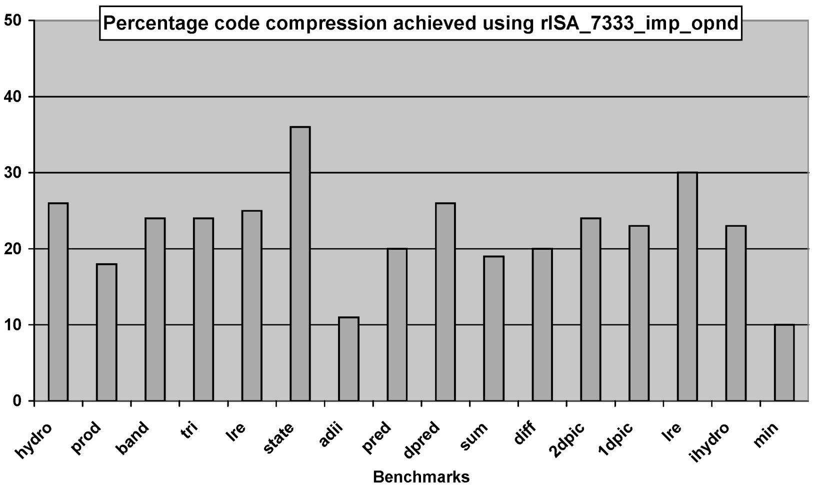

Fig. 15. Percentage code compression achieved by using *rISA_7333_imp_opnd*.

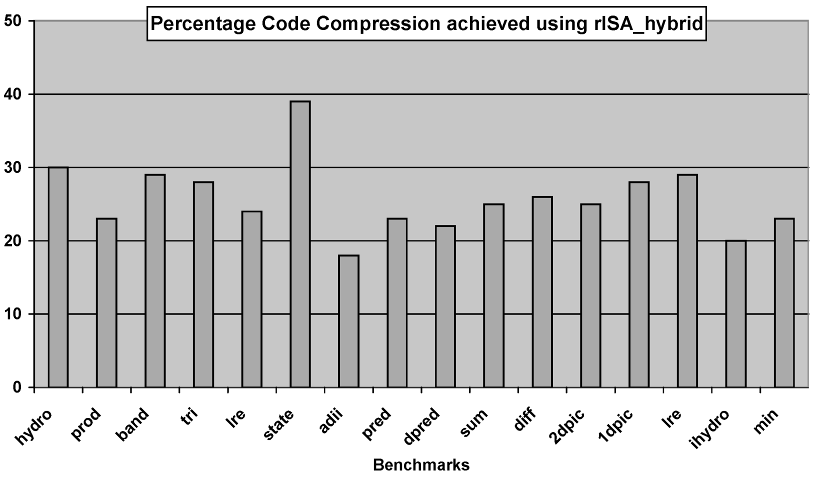

Fig. 16. Percentage code compression achieved by using *rISA_hybrid*.

The fifth rISA design point that is, *rISA_hybrid*, is a custom ISA for each benchmark. All the previous techniques are used to define a custom rISA, for each benchmark. In this rISA, design instructions can have variable register accessibility. Complex instructions with operands having different register set accessibility are also supported. The register set accessible by operands varies from 4 to 32 registers. We profiled the applications and manually (heuristically) determine the combinations of operand bit-width sizes that provide best-code size reduction. The immediate field is also customized to gain best-code size reduction.

Figure 16 plots the code compression achieved by *rISA hybrid*. Because it is customized for the application set, the *rISA hybrid* achieves the best-code size reduction. *rISA_Hybrid* achieves about 26% overall improvement

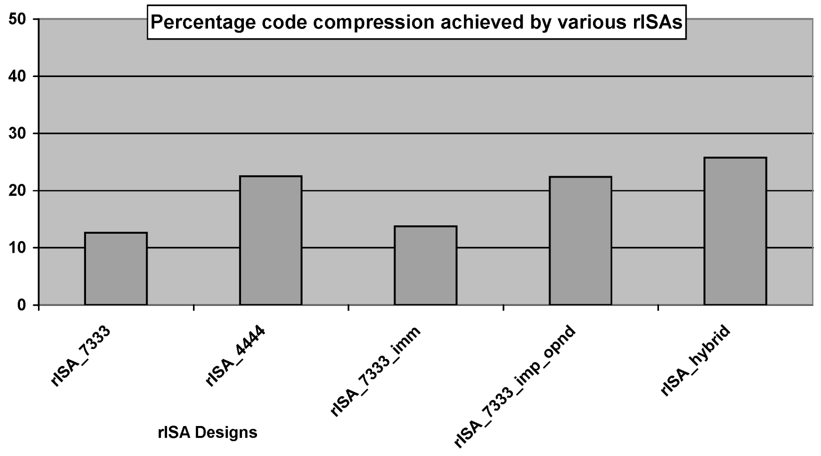

Fig. 17. Code size reduction for various rISA architectures.

over normal instruction set. The code compression is consistent across all benchmarks.

Figure 17 shows the average (over benchmarks) reduction in code compression achieved in various rISA designs. It can be deduced from the experimental results presented that the code compression achieved is very sensitive to the application characteristics and the rISA itself. Choosing the correct rISA for the applications can result in up to 14% more code compression (26%–12%). Thus, it is very important to design and tune the rISA to the applications.

We have shown that the register pressure heuristic consistently achieves high code compressions (up to 22%). We also observe that the code compression obtained is very sensitive on the application and the rISA itself. Therefore, there is a need to explore the rISA design space effectively and that high degrees of code compression can be achieved by tuning the rISA for specific applications.

7. SUMMARY AND FUTURE WORK

rISA (or reduced bit-width Instruction Set Architecture) refers to architectures that support execution of instructions of two different widths. rISA is a popular architectural feature in modern processors not only to decrease code size, but also to reduce power consumption. However, the code-generation techniques exploiting the rISA architectural feature still remain rudimentary. A critical problem in generating code for rISA is deciding upon which parts of code to convert into rISA instructions. Existing techniques perform this conversion at routine-level granularity, and therefore are not able to achieve high code compressions. In this article, we propose instruction-level granularity of conversion and demonstate consistently high degrees of code compression. Furthermore, indiscriminate conversion into rISA instructions can cause an increase in the code size due to spilling caused by increase in register pressure. Traditional approaches did not take this factor into account. In this article, we proposed a register-pressure-based proftiability heuristic to avoid the regions of code, whose conversion leads to an increase in the code size due to register spilling. Compared to previous code-generation techniques, our compilation consistently obtains high degrees of code compression. In addition, we use our compilation technique to explore several rISA designs. We observe that code compression achieved by a rISA is dependent not only on the application characteristics, but also on the rISA design itself. Our results imply that it is very important for application specific processors to tune the rISA design. However, the rISA design space is huge and full exploration cannot be performed. Our future work will include developing intelligent heuristics to effectively explore rISA design space and develop strategies to tune rISA design to application characteristics.

ACKNOWLEDGMENTS

We wish to thank Radu Cornea for his help with the register allocation in our framework. The authors would like to thank all the ACES (Architectures and Compilers for Embedded Systems) team members for their valuable support in this framework.

REFERENCES

Advanced RISC Machines, Ltd. 2003. *ARM7TDMI (Rev 4) Technical Reference Manual*. Advanced RISC Machines, Ltd. Cambridge, England, http://www.arm.com/pdfs/DDI0234A_7TDMIS_R4.pdf.

ARC Cores. 2005. *ARCtangent-A5 Microprocessor Technical Manual*. ARC Cores, Herts, England, http://www.arc.com/documentation/productionbriefs.

Benini, L., Macii, A., and Nannarelli, A. 2001. Cached-code compression for energy minimization in embedded processors. In *Proceedings of ISLPED*.

Briggs, P., Cooper, K., and Torczon, L. 1994. Improvements to graph coloring register allocation. In *Proceedings of PLDI*.

Halambi, A., Grun, P., Ganesh, V., Khare, A., Dutt, N., and Nicolau, A. 1999. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *Proceedings of DATE*.

Halambi, A., Shrivastava, A., Dutt, N., and Nicolau, A. 2001. A customizable compiler framework for embedded systems. In *Proceedings of SCOPES*.

Khare, A., Savoiu, N., Halambi, A., Grun, P., Dutt, N., and Nicolau, A. 1999. V-SAT: A visual specification and analysis for system-on-chip exploration. In *Proceedings of EUROMICRO*.

Krishnaswamy, A. and Gupta, R. 2003. Enhancing the performance of 16-bit code using augmenting instructions. In *Proceedings of LCTES*.

Kwon, Y.-J., Ma, X., and Lee, H. J. 1999a. PARE: Instruction set architecture for efficient code size reduction. *Electronics Letters 25th Nov'99*, *35*, 24, 2098–2099.

Kwon, Y.-J., Parker, D., and Lee, H. J. 1999b. Toe: Instruction set architecture for code size reduction and two operations execution. In *Proceedings of CASES*.

Lekatsas, H., Henkel, J., and Wolf, W. 2000. Code compression for low power embdedded system design. In *Proceedings of DAC 2000*.

LSI LOGIC. 2000. *TinyRISC LR4102 Microprocessor Technical Manual*. LSI LOGIC, Milpitas, CA, http://www.lsilogic.com/files/docs/techdocs/microprocessors/MIPSTinyRISC/4102/lr4102ds_3000.pdf.

Nicolau, A. and Novack, S. 1993. Trailblazing: A hierarchical approach to percolation scheduling. In *Proceedings of ICPP*.

Novack, S. and Nicolau, A. 1997. Resource directed loop pipelining: Exposing just enough parallelism. *The Comput. J*.

Shrivastava, A. and Dutt, N. 2004. Energy efficient code generation exploiting reduced bit-width instruction set architecture. In *Proceedings of ASPDAC*.

ST Microelectronics. 2004. *ST100 Technical Manual*. ST Microelectronics, Geneva, Switzerland, http://www.st.com/stonline/books/pdf/docs/10538.pdf.

Received June 2004; revised January 2005; accepted February 2005