An Efficient Compiler Technique for Code Size Reduction using Reduced Bit-width ISAs *

Ashok Halambi Aviral Shrivastava Partha Biswas Nikil Dutt Alex Nicolau

Center for Embedded Computer Systems
Department of Information and Computer Science
University of California, Irvine, CA 92697-3425, USA

Abstract

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 function-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 profitability based compiler heuristic that operates at the instruction-level granularity and is able to effectively take advantage of both Instruction Sets. We also demonstrate improved code size reduction, for the MIPS 32/16 bit ISA, using our technique. Our approach more than doubles the code size reduction achieved by existing compilers.

1 Introduction

Programmable RISC processors are increasingly being used to design modern embedded systems. Examples of such systems include consumer electronics items, cell-phones, printers, modems 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 mentioned earlier. Consequently, there is a lot of interest in reducing program code size to decrease ROM size.

One popular architectural modification to achieve code size reduction is the "dual Instruction-Set" feature, with the processor capable of executing two different Instruction-Sets (IS). One, the "normal" set, contains the original IS, and the other, the "reduced bit-width" set, encodes the most commonly used instructions using fewer bits. A very good example is the ARM processor with a 32-bit IS and a 16-bit IS called the Thumb IS. Other processors with a similar feature include the MIPS 32/16 bit TinyRISC, STMicro's ST100 and the ARC Tangent processor. We term this feature the "**r**educed bit-width **I**nstruction **S**et **A**rchitecture" (**rISA**).

Processors with this feature dynamically expand (decompress) the narrow rISA instructions into corresponding normal instructions. This decompression usually occurs during the decode stage. Typically, each rISA instruction has an equivalent instruction in the normal IS. This makes decompression simple and can usually be done without any cycle penalty. As the decompression engine converts rISA instructions into normal instructions, no other hardware is needed to execute rISA instructions. Thus, the main advantage of rISA lies in achieving good code size reduction with minimal hardware additions. However, as more rISA instructions are required to implement the same task, rISA code has slightly lower performance compared to normal code. Experiments conducted using the ARM processor show a 30% reduction in code size, with minimal performance penalty for small functions. [9]

The rISA IS, because of bit-width restrictions, encodes only a subset of the normal instructions and allows access to only a small subset of registers. For example, the ARM Thumb allows access to 8 registers

*This work was partially supported by grants from NSF (MIP-9708067),DARPA (F33615-00-C-1632) and a Motorola Fellowship.

(out of 16 general-purpose ARM registers). Producing code optimized for rISA thus involves making a trade-off between smaller code size (due to greater code density) and increased number of instructions due to higher register pressure (resulting in more spills). In this paper we propose an approach that makes this trade-off in order to achieve code size reduction for rISA.

In Section 2 we introduce the problem of code generation for rISA architectures, and also outline our technique for solving it. In Section 3 we describe previous work on architectures and compiler techniques for rISA. In Section 4 we present the salient features of an architecture that supports rISA. Section 5 describes our compiler technique for such architectures while Section 6 discusses the profitability heuristic used in our technique. Section 7 presents some experiments conducted on the MIPS 32/16 architecture and Section 8 concludes the paper.

2 Problem Description

While it is possible to manually achieve good code size reduction using rISA, code generation for rISA (termed *rISAization*) is a challenging task requiring the compiler to take various factors into account. For example, some operations (such as multiply-accumulate) may require multiple rISA instructions to implement. Further, due to the limited availability of registers, the compiler may need to insert spills (or register moves) thus decreasing the benefits of rISAization.

Existing compilers *rISAize* programs on a function-by-function basis. A function is rISAized if all its instructions can be converted to rISA instructions and if it is profitable (in terms of decreased code size) to convert the entire function. A major drawback of rISAizing at the function level granularity is that the compiler misses out on the opportunity to achieve greater code size reduction by selectively rISAizing sections of the function. For example, it is possible that rISAizing the whole function is not profitable, but rISAizing sections of the function results in an overall code size reduction. Another drawback of current approaches is that rISAization is done either as a post-pass during assembly generation or as a separate instruction selection phase. A technique that can work in conjunction with Instruction Scheduling and Register Allocation may be able to produce better results (both in terms of code-size and performance).

In this paper, we propose a compiler integrated rISAization technique, that operates at the instruction-level granularity and is able to selectively rISAize regions of functions to achieve better code size.

3 Related Work

Several RISC processors for embedded systems support dual Instruction Sets. The ARM7TDMI processor [1] from ARM Inc. features a 32-bit IS and a 16-bit IS extension called the Thumb[9]. Switching between the two Instruction Sets is accomplished through the use of explicit mode change (between Normal and Thumb) instructions. A decompression engine converts Thumb instructions to normal instructions during the decode stage. Thumb instructions are able to access only 8 general purpose registers (out of a possible 16 in normal mode) and can encode only small immediate values. Experimental results show a compression factor of 30% with 10% - 15% decrease in performance using the Thumb IS.

The MIPS ISA features a 16-bit extension called MIPS16 IS[10]. MIPS16 IS contains an *extend* opcode which extends the values of immediate operands that were otherwise not representable because of bit-width constraints. There are no explicit mode change instructions to switch between the 32-bit and 16-bit IS. Rather, code alignment dictates the mode of execution. If a routine is not aligned at the word boundary, it is assumed to be composed of MIPS16 instructions. Experimental results show code size reduction of up to 40% using MIPS16 ISA.

The ST100 Core[13] from ST Microelectronics is a 32-bit Microcontroller/DSP architecture which hosts a complete 32-bit instruction set (GP32) as well as a 16-bit DSP instruction set (GP16) for code compaction. Switching between instruction sets is performed by software instructions or by an external event.

The Tangent-A5 configurable RISC processor from ARC[2] also supports dual Instruction Sets. However, instead of using a decompressor to expand 16-bit instructions to 32-bit instructions, the Tangent processor executes the 16-bit instructions natively.

The ARM Thumb IS was redesigned by Kwon et. al.[14] to compress more instructions and further improve the efficiency of code size reduction. This new IS called Partitioned Register Extension (PARE)[4], 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.

While there has been considerable research in the design of architectures which have dual instruction sets -

a full length set and a rISA set, 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 regions of routines.

In this paper, we present a compiler technique to produce optimized code for dual IS architectures that operates at the instruction-level granularity. Our technique, is able to aggressively reduce code size by implementing sub-regions of functions using higher density rISA operations. To our knowledge, this is a unique feature of our technique that has not been addressed by previous published work.

4 Architecture Model

In this section we briefly mention the salient features of the rISA (Reduced bit-width Instruction Set Architecture) processor model. rISA instructions are space efficient encodings of most frequently used normal instructions. The rISA IS may be "complete", i.e. containing rISA instructions corresponding to each class of normal instructions, or may contain a strict subset of the class of normal instructions. The number of opcodes in the rISA IS also affects the number of bits available to address register (and immediate) operands in the instructions. Every rISA model implements a trade-off between more opcodes and access to more registers. Our technique is capable of efficient compilation for all types of the rISA model.

As rISA processors can operate in either the rISA mode or in the normal mode, a mechanism to specify the mode is necessary. For most rISA processors, this is accomplished using explicit instructions that change the mode of execution. We term an instruction in the normal IS that changes mode from normal to rISA the *mx* instruction, and an instruction in the rISA IS that changes mode from rISA to normal the *rISA_mx* instruction. The MIPS16 ISA has an interesting mechanism for specifying mode changes. All functions encoded using MIPS16 instructions begin at the half word boundary. Thus, calls (and returns) to half word aligned addresses also change the mode from normal to rISA. We assume a rISA model with explicit mode change instructions, as we require the ability to change modes within functions.

In order to mitigate the problem of limited bits to specify the operands (both register and immediate) a number of modifications to the basic rISA model have been proposed. 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[1]. A technique to increase the size of the immediate value operand is to implement an *extend* operation that completes the immediate field of the succeeding instruction.

In the next section, we describe our compiler technique to efficiently compile for architectures that support the rISA model described above. For simplicity of discussion, we assume that the size of a normal instruction is 4 bytes, and that of a rISA instruction is 2 bytes. Note, however, that the sizes are not a restriction in our technique.

5 Compiler Flow

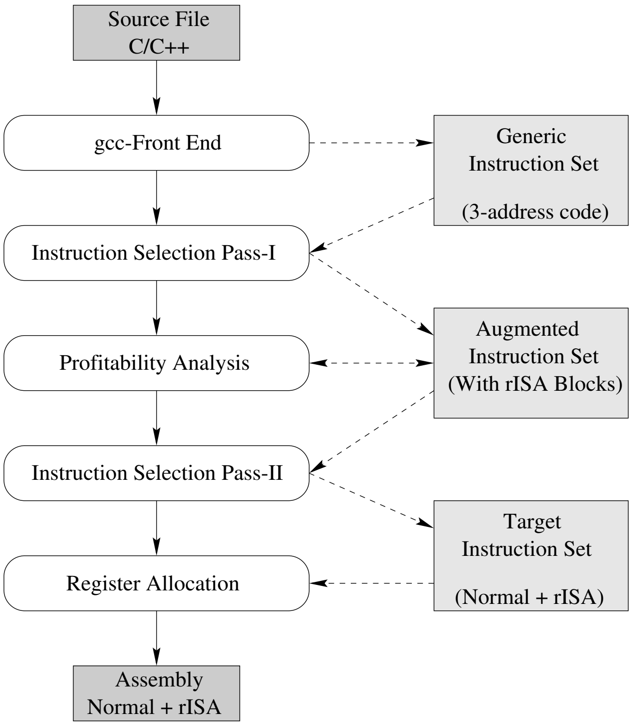

Figure 1. EXPRESS compiler flow

We implemented our rISA compiler technique in the EXPRESS retargetable compiler. EXPRESS [8] is an optimizing, memory-aware, Instruction Level Parallelizing (ILP) compiler. EXPRESS uses the EXPRESSION ADL [6] 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.

[1]This is possible because a move has only two operands and hence has more bits to address each operand.

The front-end is GCC based and performs some of conventional optimizations. The core transformations in EXPRESS include **RDLP**[12] – a loop pipelining technique, **TiPS** : Trailblazing Percolation Scheduling[11] – a speculative code motion technique, Instruction Selection and Register Allocation. The back-end generates assembly code for the processor ISA.

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 function to convert to rISA instructions, and which ones to normal target instructions. Figure 1 shows the phases of the EXPRESS compiler with our rISAization technique.

rISAization involves the following tasks: (1) deciding whether a section of code can be converted to rISA instructions, (2) deciding whether it is profitable to rISAize the section, (3) choosing the appropriate (best) rISA instructions, and (4) inserting the mode change instructions. Tasks (1) and (3) are best accomplished as part of the Instruction Selection process. However, a major difficulty associated with this approach is that the Instruction Selection phase operates on trees of instructions (created by following data dependency chains) rather than on contiguous regions of code while task (2) can only be performed on sequential sections of the function.

In order to solve this problem, we divide Instruction Selection for rISA into two phases. In the first phase instructions that can be converted to rISA are marked. A group of contiguous marked instructions then forms a **rISA Block**. A profitability function analyzes each rISA Block and decides whether it is profitable to rISAize the block of instructions. The profitability analysis algorithm is discussed in greater detail in Section 6.

In the second phase of Instruction Selection, all generic instructions within profitable rISA blocks 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.

The actual register allocation of variables is done during the Register Allocation phase. The EXPRESS compiler implements a modified version of Chaitin's solution [3] 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.

In the next section, we discuss in greater detail the profitability analysis algorithm which determines if it is useful to rISAize a rISA block. In Section 7 we present experiments demonstrating the efficacy of our approach.

6 Profitability Analysis

rISAizing a block of instructions impacts both the code size and the performance of the program. In our technique, this impact 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. Below we describe how the PA function measures the estimated impact on code size and performance. A detailed description of the PA function can be found in [5].

Code Size (CS): Figure 2 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, $CS1 = 4 + 2 = 6 bytes$.

NOP (CS2): Most architectures require that normal instructions be aligned at word boundaries. However, rISAizing a block with odd number of instructions[2] will cause the succeeding normal instruction to be mis-aligned. In such cases, an extra *rISA_nop* (No-operation instruction) needs to be added inside the rISA block. We conservatively estimate that each rISA block needs a rISA_nop instruction. $CS2 = 2 bytes$.

[2] Including the rISA_mx instruction

```
1.  Estimate_CodeSize_Reduction (Block bl)
2.  {
3.      CS1 = Size_Of(mx) + Size_Of(risa_mx);
4.      CS2 = Size_Of(risa_nop);
5.      CS3 = Extra_Spill_Reload_Estimate(bl);
6.      return  Size_Block(bl, NORMAL) - Size_Block(bl, rISA) - CS1 - CS2 - CS3;
7.  }

8.  Extra_Spill_Reload_Estimate(Block bl);
9.  {
                    // Estimate spill code if the block is rISAized.
10.     extra_rISA_reg_press = Avg_Reg_Press(bl, rISA_vars) - K1 * NUM_rISA_REGS;
11.     if (extra_rISA_reg_press > 0)
12.         avail_non_rISA_regs = TOTAL_REGS - NUM_rISA_REGS;
13.         rISA_spills = (extra_rISA_reg_press) * (bl.num_instrs / Avg_Live_Len(bl));
14.     else
15.         avail_non_rISA_regs = TOTAL_REGS - NUM_rISA_REGS - extra_rISA_reg_press;
16.         rISA_spills = 0;

17.     extra_non_rISA_reg_press = Avg_Reg_Press(bl, non_rISA_vars) - K1 * avail_non_rISA_regs;
18.     if (extra_non_rISA_reg_press > 0)
19.         non_rISA_spills = extra_non_rISA_reg_press;
20.     else    non_rISA_spills = 0;

21.     spill_code_if_rISA  = rISA_spills * SIZE_rISA_INSTR + non_rISA_spills * SIZE_NORMAL_INSTR;

                    // Estimate spill code if the block is not rISAized.
22.     extra_normal_reg_press = Avg_Reg_Press(bl, all_vars) - K1 * TOTAL_REGS;
23.     if (extra_normal_reg_press > 0)
24.         normal_spills = extra_normal_reg_press * (bl.num_instrs / Avg_Live_Len(bl));
25.     else    normal_spills = 0;

26.     spill_code_if_normal  = normal_spills * SIZE_NORMAL_INSTR;

27.     extra_spill_code   = spill_code_if_rISA - spill_code_if_normal;
28.     extra_reload_code  = K2 * extra_spill_code * Avg_Uses_Per_Def(bl);

29.     return extra_spill_code + extra_reload_code;
30. }
```

TOTAL_REGS = 8, NUM_rISA_REGS = 8, SIZE_rISA_INSTR=2 bytes, SIZE_NORMAL_INSTR=4 bytes.
K1, and K2 are control constants.

Figure 2. Heuristic to estimate the code size reduction.

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[3] 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 21 in Figure 2). 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 10-16) and spills due to non rISA variables (lines 17-20). 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 life-time). In a block, the extra register pressure (that causes spilling) is the difference between Avg_Reg_Press and the number of available registers (lines 10, 17, 22).

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

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 13, 19, 24).

The next step is to estimate the total number of spills if the block is not converted to rISA instructions (line 26). 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 6) is
$CS = NumInstrs(RisaBlock)*2-CS1-CS2-CS3$. A CS value greater than zero implies that converting the block to rISA instructions is profitable in terms of code size.

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

7 Experiments

In this section we present results from experiments conducted on the MIPS 32-bit/MIPS 16-bit machine model. We measured the code size reduction for some benchmarks using our technique and the gcc compiler for the MIPS 32/16-bit ISA. The benchmarks were chosen from the Livermore Loops suite of mainly numerical computation kernels. Table 1 shows the code size reductions obtained using EXPRESS and gcc for the MIPS 32/16-bit ISA. GCC was run on the benchmarks using the **-Os** option to optimize for code size.

Columns 2-3 show the size of code (in bytes) produced by GCC, while columns 4-5 show the size of code produced by EXPRESS. As can be seen, the regular code size for MIPS32 produced by both compilers (columns 2, 4) is comparable. However, EXPRESS is able to compress code much more effectively as compared to gcc. In fact, for some benchmarks (1dpic, dot) the MIPS16 code produced by gcc is much worse as compared to MIPS32 code while EXPRESS is able to achieve code size reduction. This is primarily because EXPRESS is able to selectively compress sections of the function and thus avoid inserting a large number of spills. For benchmarks where both compilers achieved lower code size for MIPS16, EXPRESS was able to achieve code size reduction of **38%**, while gcc was able to achieve a reduction of **14%**, on an average. These experiments demonstrate effectiveness of our technique in rISAizing functions as compared to existing approaches. By operating at the instruction-level granularity, our technique is able to accurately estimate the impact of rISAization and also to selectively rISAize only profitable (in terms of code size) sections of functions.

Bmarks	GCC code size			EXPRESS code size		
	MIPS32	MIPS16	%	MIPS32	MIPS16	%
hydro	140	126	**10**	132	80	**39**
ccg	216	190	**12**	280	166	**41**
prod	100	74	**26**	76	48	**37**
band	164	140	**15**	156	94	**40**
tri	108	88	**19**	100	54	**45**
lre	152	124	**18**	176	118	**33**
state	264	234	**11**	232	160	**31**
adii	696	728	**-5**	752	510	**32**
pred	248	228	**8**	144	110	**24**
dpred	256	222	**13**	264	134	**49**
sum	108	76	**30**	92	48	**48**
diff	100	62	**38**	80	42	**48**
2dpic	348	270	**22**	288	162	**44**
1dpic	488	570	**-17**	428	256	**40**
fort	860	714	**17**	1008	616	**39**
ehydro	876	852	**3**	1068	634	**41**
lre	192	166	**14**	192	128	**33**
dot	288	452	**-57**	344	234	**32**
mult	172	152	**12**	184	114	**38**
planck	188	202	**-7**	184	138	**25**
ihydro	324	300	**7**	344	236	**31**
min	120	78	**35**	128	88	**31**

Table 1. Code Size reduction using GCC and EXPRESS

We used SIMPRESS[7], 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 the rISA MIPS16 Instruction Set. The memory subsystem was modeled with no caches and a single cycle latency 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. For more details, including performance numbers for individual benchmarks, please refer to [5]. These experimental results show that our technique is able to effectively reduce code size using rISA with minimal performance impact.

8 Summary and Future Work

An architectural feature for improving code density of RISC processors is the reduced bit-width Instruction Set Architecture (rISA) extension. While rISA can potentially achieve huge reduction in code size, existing compilers are ill-equipped to take advantage of this feature. In this paper we present a compiler technique for achieving code size reduction using the reduced bit-width Instruction Set Architecture (rISA). The novel features of our technique include (1) its ability to operate at the instruction-level granularity and achieve greater code size reduction by selectively converting sections of a function; (2) its integration to an existing compiler flow and its ability to work in conjunction with other compiler phases; and (3) a heuristic to estimate the amount of spills/reloads due to restricted availability of registers. This technique has been integrated into the EXPRESS ILP compiler and experimental results show the efficacy of our approach as compared to approaches with function-level granularity. Our technique is able to generate more than twice the amount of code size reduction, on an average, over existing approaches. Future work in this area includes the problem of compiler-in-the-loop design space exploration (DSE) of rISA architectures, and efficient scheduling for such architectures.

References

[1] Advanced RISC Machines Ltd. (http://www.arm.com). *ARM7TDMI Technical Manual.*

[2] ARC Cores (http://www.arccores.com). *ARCtangent-A5 microprocessor Technical Manual.*

[3] P. Briggs, K.D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. In *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation*, 1994.

[4] I-Cheng Chen C. Lefurgy, P. Bird and T. Mudge. Improving code density using compression techniques. In *IEEE, Proceedings of Micro-30*, 1997.

[5] [Omitted for Blind Review]. An efficient compiler technique for reduced bit-width instruction set architectures. Technical report, [Omitted for Blind Review], 1997.

[6] [Omitted for Blind Review]. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *Proc. of Design Automation and Test in Europe*, 1999.

[7] [Omitted for Blind Review]. V-SAT: A visual specification and analysis for system-on-chip exploration. In *Proc. EUROMICRO-99*, 1999.

[8] [Omitted for Blind Review]. A customizable compiler framework for embedded systems. In *SCOPES*, 2001.

[9] L. Goudge and S. Segars. Thumb: Reducing the cost of 32-bit risc performance in portable and consumer applications. In *Proceedings of COMPCON'96*, 1996.

[10] K. Kissell. MIPS16: High-density MIPS for the embedded market. Technical report, Silicon Graphics MIPS Group, 1997.

[11] A. Nicolau and S. Novack. Trailblazing: A hierarchical approach to percolation scheduling. In *Proc. of Intn'l Conf. on Parallel Processsing*, 1993.

[12] S. Novack and A. Nicolau. Resource directed loop pipelining : Exposing just enough parallelism. *The Computer Journal*, 1997.

[13] STMicroelectronics (http://www.st.com). *ST100 Technical Manual.*

[14] Xiarong Ma Young-Jun Kwon and Hyuk Jae Lee. PARE: instruction set architecture for efficient code size reduction. *Electronics Letters 25th Nov'99 Vol. 35 No. 24*, pages 2098–2099, 1999.