A Design Space Exploration Framework for Reduced Bit-width Instruction Set Architecture (rISA) Design *

Ashok Halambi Aviral Shrivastava Partha Biswas Nikil Dutt Alex Nicolau

{ ahalambi, aviral, partha, dutt, nicolau }@ics.uci.edu

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

ABSTRACT

Code size is a critical concern in many embedded system applications, especially those using RISC cores. 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 (termed rISA) can potentially reduce the code size by up to 50% with minimal performance degradation. However, contemporary processors incorporate only a simple rISA feature with severe restrictions on register accessibility. We present a compiler-in-the-loop Design Space Exploration framework that is capable of exploring various interesting rISA designs. We also present experimental results using this framework and show rISA designs that improve on the code size reduction obtained by existing rISA architectures.

Categories and Subject Descriptors

C.0 [**Computer Systems Organization**]: Instruction Set Design

General Terms

reduced bit-width Instruction Set Architecture

Keywords

rISA, dual Instruction Set, design space exploration, register pressure, reduced bit-width Instruction Set, compressed Instruction Set, thumb

*rISA term coined in [9]

ISSS'02, October 2–4, 2002, Kyoto, Japan.

1. INTRODUCTION

RISC micro-processors, which offer the benefits of increased design flexibility, high computing power and low on-chip power consumption, are used to design modern, high-end embedded systems. Examples of such systems abound and include cell-phones (Nokia 9210 Communicator[5] using a 32-bit ARM9-based RISC CPU), PDAs (Compaq iPAQ PocketPC[4] using an Intel StrongARM 32-bit RISC), routers (Cisco 12000 Series[3] using MIPS R5000 CPU), etc.

However, these 32-bit RISC processor based systems suffer from the problem of poor code density and require more ROM for storing program code. This is a severe limitation for these large volume, cost sensitive embedded systems. Consequently, there is a lot of interest in reducing program code size to decrease ROM size.

One innovative 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. If an application is fully expressed in terms of "reduced bit-width" instructions, then a 50% code size reduction is achieved as compared to when it is expressed in terms of normal instructions.

A very good example of an architecture supporting "reduced bit-width" 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. This feature is termed the "**r**educed bit-width **I**nstruction **S**et **A**rchitecture" (**rISA**) [9]. The term **rISAize** refers to the process of converting program code to rISA instructions.

Processors with rISA dynamically expand (translate) the narrow rISA instructions into corresponding normal instructions. This translation usually occurs during the decode stage. Typically, each rISA instruction has an equivalent instruction in the normal IS. This makes translation simple and can usually be done without any performance penalty. As the translator 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. ARM/Thumb[2] and

MIPS–16[12] report 30%–40% reduction in code size, with minimal performance penalty for small functions.

The rISA IS, because of bit-width restrictions, can encode only a subset of the normal instructions and allows access to only a small subset of registers. Such severe restrictions make the code-size reduction obtainable by using rISA very sensitive to the compiler quality and the application features. For example, if the application has high register pressure, or if the compiler does not do a good job of register allocation, it might be better to increase the number of accessible registers at the cost of encoding only a few opcodes in rISA. Thus, it is very important to perform compiler-in-the-loop design space exploration (DSE) while designing rISA architectures. Contemporary rISA processors (such as ARM/Thumb, MIPS32/16) incorporate a very simple rISA model with rISA instructions able to access 8 registers (out of 16 or 32 general-purpose registers). In this paper, we show that varying this model by considering the application characteristics and the compiler quality results in substantial improvement of code-size reduction. We present a rISA design space exploration framework that incorporates a compiler designed to optimize for rISA and is able to explore a wide range of architecture design points.

In Section 2 we review some contemporary rISA processors, and outline some proposed rISA modifications and compiler techniques to achieve better compression ratios. In Section 3 we present the rISA model, and in Section 4 we describe the exploration parameters that are used to generate alternative rISA design points. Section 5 presents our compiler-in-the-loop rISA Design Space Exploration framework. Section 6 and Section 7 present experimental results evaluating various rISA designs using the framework, and Section 8 concludes the paper.

2. RELATED WORK

In this section, we first discuss how various contemporary processors support rISA. Then we look at the existing compiler techniques that make use of this architectural feature.

The ARM7TDMI processor[2] from ARM Inc. features a 32-bit "normal" IS, and 16-bit "reduced" IS, called "Thumb". "Thumb" instructions can access 8 registers (out of 16 registers in normal mode), and can encode only small immediate values. Mode change (between the two Instruction Sets) takes place only on a branch, using special instruction "BX"(Branch and mode eXchange). A translation unit converts the "Thumb" instructions into normal instructions in the pre-decode stage of the pipeline.

The MIPS ISA features a 16-bit extension called MIPS16[12]. 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. Instead, code alignment dictates the mode of execution: if a routine is aligned at the half-word boundary it is assumed to be composed of MIPS16 instructions.

The ST100 core[6], from ST Microelectronics is a 32-bit microcontroller/DSP architecture which hosts a 32-bit normal IS, and a 16-bit reduced bit-width IS. The switching between the normal IS and reduced IS is performed by software instructions or by external event.

The Tangent-A5 configurable RISC processor from ARC[1] also supports dual Instruction Sets. However, instead of using a translation unit to expand 16-bit instructions to 32-bit instructions, the Tangent processor executes the 16-bit instructions natively.

All of the contemporary rISA architectures described above allow access to only a fixed set of 8 rISA registers due to bit-width constraints on the operands. Recognizing that the primary obstacle to achieving good code size reduction using rISA is the limited bit-width available to encode the operands, Kwon et. al.[15] propose a new rISA model called Partitioned Register Extension (PARE). In this model, the register file is split into (possibly overlapping) partitions, and each rISA instruction can only write to a particular partition. This reduces the number of bits required to specify the destination register.

rISA has not been a very effective architectural feature mainly because of absence of compiler techniques that exploit this feature to generate good code. There have been some initial efforts to improve the ability of the compiler to handle rISA. Halambi et. al[9] propose a register pressure based heuristic to determine the profitability of using rISA instructions to encode blocks of the program. This heuristic aims to avoid code size increase due to excessive spills/reloads.

In this paper, we present a compiler-in-the-loop rISA design space exploration framework that incorporates the technique mentioned above. We use the EXPRESSION Architecture Description Language (ADL)[8] to model the rISA features. The EXPRESSION description is used to retarget the compiler for the various rISA design points. The novel features of our work include the ability to model and explore a wide variety of rISA design points using the ADL and the retargetable compiler that produces good quality code for the various rISA designs.

3. RISA MODEL

In this section, we briefly describe the rISA processor model. The model defines the rISA IS, and mapping of rISA instructions to normal instructions. A rISA instruction should map to a unique normal instruction. Such a mapping simplifies the translator unit, so that it does not cause any performance delay.

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.

Every sequence of rISA instructions starts with an *mx* instruction and ends in a *rISA_mx* instruction. To ensure that the ensuing normal instruction aligns to the word boundary, a padding *rISA_nop* instruction is needed.

Due to bit-width constraints, a rISA instruction can access only a subset of registers. The register accessibility of each register operand must be present in the rISA model. The width of immediate fields must also be specified.

In addition, there may be special instructions in the rISA model to help the compiler generate better code. A very useful technique to increase the number of registers accessible in rISA mode is to implement a *rISA_move* instruction that can access all registers (This is possible because a move has only two operands and hence has more bits to address each operand.). A technique to increase the size of the immedi-

ate value operand is to implement a *rISA_extend* instruction that completes the immediate field of the succeeding instruction.

Numerous such techniques can be explored to increase the efficacy of rISA architectures. In the next section we describe some of the more important rISA design parameters that can be explored using our framework.

4. RISA DSE PARAMETERS

Conventional rISA architectures like Thumb[2] and MIPS16[12] fix the register set accessible by rISA instructions to be 8. Thus, each operand requires 3 bits for specification. This implies that, for three operand instructions, up to 128 opcodes can be encoded in rISA. The primary advantage of this approach is that most normal 32-bit instructions can also be specified as rISA instructions. However, this approach suffers from the drawback of increased register pressure possibly resulting in poor code size. One modification is to increase the number of registers accessible by rISA instructions to 16. However, in this model, only a limited number of opcodes are available. Thus, depending on the application, large sections of program code might not be implementable using rISA instructions. The design parameters that can be explored include the number of bits used to specify operands (and opcodes), and the type of opcodes that can be expressed in rISA.

Another important rISA feature that impacts the quality of the architecture is the "implicit operand format" feature. 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.

A severe limitation of rISA instructions is the inability to incorporate large immediate values. For example, with only 3 bits 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 will, however, reduce the number of bits available for opcodes (and also the other operands). This trade-off can be meaningfully made only with a compiler-in-the-loop DSE framework.

Various other design parameters such as partitioned register files, shifted/padded immediate fields, etc also should be explored in order to generate a rISA architecture that is tuned to the needs of the application and to the compiler quality. While some of these design parameters have been studied in a limited context, there has been no previous work that seeks to generate rISA designs that combine all of these features. Our exploration framework, as explained in the next section, is able to quantify the impact of these features both individually and in various combinations. Also, since it incorporates the compiler, the quality of each design point is measured accurately.

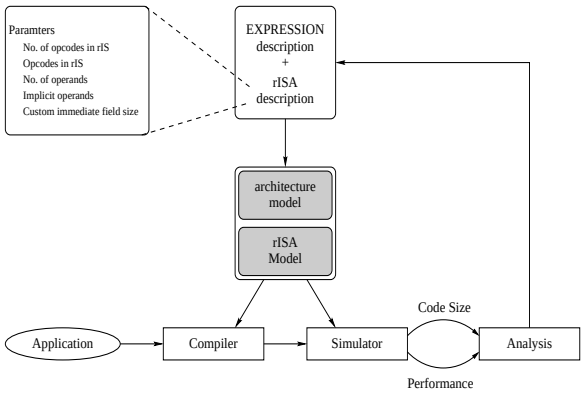

Figure 1: Design Space Exploration flow

5. DESIGN SPACE EXPLORATION

Figure 1 presents the DSE framework used to explore rISA design points. The processor architecture (with the desired rISA features) is described using an Architecture Description Language (ADL) called EXPRESSION[8]. This description is then input to the EXPRESS retargetable compiler[10] and SIMPRESS simulator[11]. The desired applications are then compiled, simulated and the code size and performance numbers are generated for analysis. The various rISA design parameters mentioned in the previous section can be described in EXPRESSION which is then used to retarget the EXPRESS compiler to produce code optimized for those features. Below, we describe our DSE framework in greater detail. First, we describe how we capture rISA information in the EXPRESSION ADL. Then, we describe the rISA optimization techniques incorporated by the EXPRESS compiler.

5.1 ADL based DSE

EXPRESSION is an ADL designed to support design space exploration of a wide class of processor architectures. EXPRESSION contains an integrated specification of both structure and behavior of the processor-memory system. The structure is specified as a net-list of components (i.e., units, storages, ports and connections) along with a high-level description of the pipeline and data-transfer paths in the architecture. The behavior describes the Instruction Set of the processor. Each instruction is defined in terms of its opcode, operands and its format.

Specification of the rISA model in EXPRESSION consists of describing the rISA instructions and the restrictions on the operands. Each rISA instruction is specified as the compressed counterpart of a normal instruction. The register accessibility restrictions are specified by considering the rISA registers as a special class of registers (that is a subset of the general purpose register class). Furthermore, the limitations on the immediate values that can be specified in a rISA instruction are also specified. Finally, some special rISA instructions such as the *mx, rISA_mx, rISA_nop, rISA_extend, rISA_load, rISA_store* and *rISA_move* are identified in the specification. This information in the EXPRESSION description is used to derive a rISA architecture model (Figure 1) that is used by the retargetable compiler and the simulator.

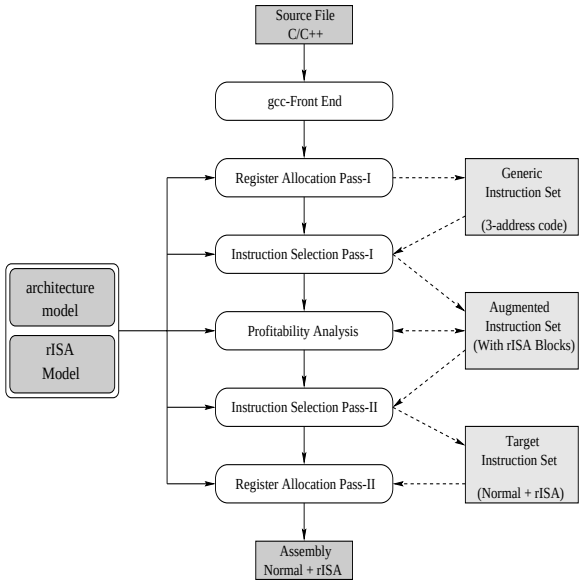

Figure 2: EXPRESS compiler flow

5.2 Compiler-in-the-loop DSE

The heart of the DSE framework is the EXPRESS retargetable compiler. EXPRESS [10] is an optimizing, memory-aware, Instruction Level Parallelizing (ILP) compiler. 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**[14] – a loop pipelining technique, **TiPS** : Trailblazing Percolation Scheduling[13] – a speculative code motion technique, Instruction Selection and Register Allocation. The back-end generates assembly code for the processor ISA.

Figure 2 describes the phases of the EXPRESS compiler that are used to perform rISAization. The Front-End outputs a sequence of generic 3-addr instructions. Since registers are usually the critical constraints in rISA architectures, an initial register allocation is then performed. This ensures that the later stages of instruction selection and scheduling do not aggressively optimize the code (thus inserting spills/reloads). Also, during this stage, the compiler determines those instructions that can be encoded using "implicit operand" formats. Instruction Selection is then performed as a two pass process. In the first pass, instructions that can be converted to rISA are marked. Further, contiguous sequences of such marked instructions are grouped as candidate rISA blocks. A profitability heuristic then analyzes each rISA block and decides whether it is profitable to rISAize the block of instructions. The second pass of Instruction Selection then replaces all generic instructions within profitable rISA blocks with rISA instructions, and all other instructions with normal target instructions. Finally, register allocation is performed respecting the register restrictions of the operands.

The profitability heuristic is a register pressure based heuristic that decides whether or not to convert the generic instructions in a rISA block to rISA instructions. The heuristic estimates the number of spills/reloads needed if the block is rISAized. This is calculated based on the number of live variables in the block, the average number of uses of a definition and the average live range of the variables in the block. For a detailed explanation please refer to [9].

The EXPRESS compiler implements a modified version of Chaitin's solution[7] to Register Allocation. Registers are grouped into (possibly overlapping) register classes. Each program variable is then mapped to the appropriate register class. For example, operands of a rISA instruction belong to the rISA register class (which consists of only a subset of the available registers). The register allocator then builds the interference graph and colors it honoring the register class restrictions of the variables.

Thus EXPRESS compiler retargets itself to generate good quality code for the machine with rISA model described in EXPRESSION. The EXPRESSION description is also used to generate the simulator for the machine. Using the rISA instructions to normal instructions mapping, the translator unit is generated, and is prepended to the decode unit.

By considering the compiler effects during DSE, the designer is able to accurately estimate the impact of the various rISA features. In the next section, we present results of DSE on some applications for the MIPS 32/16-bit architecture.

6. EXPERIMENTS

To demonstrate the efficacy of our Design Space Exploration framework, we conducted some experiments on the MIPS 32/16-bit ISA. To avoid the effects of cache on rISA performance, we perform our experiments on a non-cached RISC MIPS32 machine. Thus the results quoted in this section are purely due to IS design. We chose a set of applications from numerical computation kernels, and DSP application kernels. The MIPS 32/16-bit architecture was specified in the EXPRESSION ADL and then the various design parameters mentioned earlier were explored. Table 1 presents the results of the exploration for a few of the design points in terms of the code size reduction obtained by using the EXPRESS compiler.

The first rISA design point (*rISA_7333*) is a restricted version of the most common rISA model. In this design, each operand is represented using 3 bits. Thus, each operand has access to 8 registers, or to immediate values representable in 3 bits.However, instructions with two operands can access the entire set of 32 registers. Because of the uniform instruction format, the translation unit is very simple for this rISA design.

The second rISA design (*rISA_4444*) allows each operand, access to 16 registers. However, this reduces the number of bits available to specify the opcode to just 4. Thus, this design allows only 16 rISA instructions. We profiled the applications, and then incorporated the 16 most frequently occurring instructions in this model.

The third rISA design point (*rISA_7333_imm*) is similar to the MIPS16. In this design too, each operand has access to 8 registers. However, for instruction formats with the immediate field, unused bits from the opcode field are used to extend the immediate field. Thus, this rISA design allows larger immediate values within rISA instructions at the expense of a more complex translation unit.

The fourth rISA design (*rISA_imp_opnd*) is similar to the second design but also allows "implicit operand format" instructions. This design contains 2 operand rISA formats of

Bench marks	Percentage code size reduction				
	rISA_ 7333	rISA_ 4444	rISA_ 7333_imm	rISA_ imp_opnd	rISA_ hybrid
hydro	18	30	21	26	30
prod	23	23	23	18	23
band	19	25	21	24	29
tri	28	24	31	24	28
lre	13	19	16	25	24
state	17	39	18	36	39
adii	3	16	3	11	18
pred	6	22	6	20	23
dpred	10	9	10	26	22
sum	25	25	25	19	25
diff	23	20	23	20	26
2dpic	4	26	5	24	25
1dpic	2	16	3	23	28
Ire	5	30	6	30	29
ihydro	3	24	4	23	20
min	3	12	5	10	23

Table 1: Percentage code size reduction

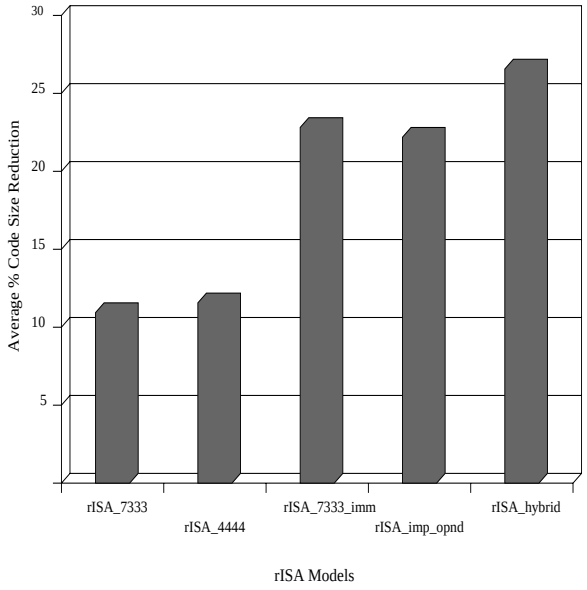

Figure 3: Code size reduction for various rISA architectures

3 operand normal instructions. The number of rISA instructions is limited to 16, however the implicit operand format instructions allow access to the full set of 32 registers.

The fifth rISA design point (*rISA_hybrid*) is a custom ISA in which instructions have variable register accessibility. Complex instructions with different operands of the same instructions having different register set accessibility are also supported. The register set accessible by operands varies from 4 to 32 registers. We profiled the applications to 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.

We use the code size of each application on MIPS32 ISA without rISA as the baseline. We then obtain the code size using each of the rISA designs explained above. The percentage code size reduction numbers shown in Table 1 are the percentage code size reduction when using a rISA design as compared to the baseline code size. The *rISA_hybrid* design which is customized for this application set has the best code size reduction for most of the applications while the *rISA_7333* design has the least code size reduction. The next section, analyzes the numbers plotted in Table 1.

7. ANALYSIS

The *rISA_7333* design does not achieve good code size reduction in benchmarks that have high register pressure (such as the adii application). This is because 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. *rISA_7333* on an average achieved 11% code size reduction over normal IS.

The register pressure problem is mitigated in the *rIS_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 23% improvement over normal IS.

The *rISA_7333_imm* design achieves slightly better compression 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 12% improvement over normal IS.

The *rISA_imp_opnd* design achieves, on average, about the same code size improvement as the *rISA_4444* design point. However, note that the performance benefits of using implicit operands is substantial for some applications such as *state* and *dpred*. *rISA_imp_opnd* achieves about 22% improvement over normal IS.

The *rISA_hybrid*, because it is customized for the application set, achieves the best code size reduction. *rIS_Hybrid* achieves about 26% overall improvement over normal IS.

Figure 3 plots the average reduction in code size in various rISA designs as compared to the code size obtained by using the normal (without rISA) MIPS32 ISA.

It is important to note that the numbers for code size compression due to rISA published so far[2, 12, 9], have been pre-assembly code size numbers. These approach does not consider the code size increase due to the immediate extend instructions that need to be inserted for instructions with larger immediate values. We present results considering the effect of adding *rISA_extend* instructions to complete the immediate value. Consider, for example, the rISA instruction *rISA_add R1 R2 10110101* in the pre-assembly. If the immediate field in *rISA_add* is only 3 bits, then the assembler will convert this to two instructions, *rISA_extend 000000010110; rISA_add R1 R2 101.* Thus the post-assembly code size and number of instructions will be more than the pre-assembly code size. The *rIS_hybrid* design achieves 40% pre-assembly code size reduction over normal code; however after including the *rISA_extend* instructions the code size reduction is 28%.

The presence of extra instructions (due to extend, spill/reload, etc.) in rISA code negatively impacts performance. For the *rISA_hybrid* architecture, the performance degradation varies from 15% up to 40%.

From the experimental results presented, it can be seen that conventional rISA designs are not optimal. However,

the best rISA design depends on the application characteristics and the compiler technology. In this paper, we presented a rISA DSE framework that is capable of exploring a wide range of rISA parameters, and also accurately estimates the impact of each design point.

7.1 Translator Unit Complexity

Another factor that influences the design of rISA architectures is the complexity of the translator logic. Simple rISA designs like *rISA_7333* and *rISA_4444* have a one-to-one mapping from rISA instructions to normal instructions. In more complex rISA design points, like *rISA_hybrid* many rISA instructions can expand to the same normal instruction. However, even for complex rISA designs, all the possible translations can be done in parallel, and the correct one selected using a multiplexor. Thus in either case, the best case delay of translator unit is not dependent on the rISA design. The translation unit of complex rISA may end up having increased area/cost design. Moreover, the translation can be done in the negative cycle (i.e pre-decode stage).

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. In this paper we presented a Design Space Exploration (DSE) framework capable of exploring various rISA parameters. Our DSE framework consists of an Architecture Description Language (ADL) that is used to specify the rISA architectural features, and a retargetable compiler that produces code optimized for rISA. The benefits of such an approach include the ability to explore a wide variety of rISA parameters and an accurate estimation of the impact of the various rISA features. We presented experimental results with rISA design points that improve on the existing rISA architectures. Future work in this area includes the problem of automatically generating customized rISA architectures for sets of applications.

9. ACKNOWLEDGEMENTS

This work was partially supported by grants from DARPA (F33615-00-C-1632), HITACHI and a Motorola Fellowship. The authors would like to thank all the EXPRESSION team members for their valuable support in this framework.

10. REFERENCES

[1] *ARCtangent-A5 microprocessor Technical Manual*. ARC Cores, http://www.arccores.com.

[2] *ARM7TDMI Technical Manual*. ARM, http://www.arm.com.

[3] *Cisco 12000 Series Router Specifications Manual*. CISCO, http://www.cisco.com.

[4] *Compaq iPAQ PocketPC Specifications Maual*. COMPAQ, http://www.compaq.com.

[5] *Nokia 9210 Specifications*. NOKIA, http://www.nokia.nl.

[6] *ST100 Technical Manual*. STMicroelectronics, http://www.st.com.

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

[8] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *Proc. of Design Automation and Test in Europe*, 1999.

[9] A. Halambi, A. Shrivastava, P. Biswas, N. Dutt, and A. Nicolau. An efficient compiler technique for code size reduction using reduced bit-width isas. In *Proc. of Design Automation and Test in Europe*, 2002.

[10] A. Halambi, A. Shrivastava, N. Dutt, and A. Nicolau. A customizable compiler framework for embedded systems. In *SCOPES*, 2001.

[11] A. Khare, N. Savoiu, A. Halambi, P. Grun, N. Dutt, and A. Nicolau. V-SAT: A visual specification and analysis for system-on-chip exploration. In *Proceedings of EUROMICRO-99*, 1999.

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

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

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

[15] X. M. Young-Jun Kwon and H. J. Lee. PARE: instruction set architecture for efficient code size reduction. *Electronics Letters*, 35(24):2098–2099, November 1999.