Automatic Instruction Set Design Through Efficient Instruction Encoding for Application-Specific Processors

Jong-eun Lee
jelee@poppy.snu.ac.kr

Kiyoung Choi
kchoi@azalea.snu.ac.kr

Nikil D. Dutt
dutt@cecs.uci.edu

Architectures and Compilers for Embedded Systems (ACES)
Center for Embedded Computer Systems, University of California, Irvine, CA 92697

Technical Report #02-23
Center for Embedded Computer Systems
University of California, Irvine, CA 92697, USA

May 2002 — Updated April 2003

Abstract

Application-specific instructions can significantly improve the performance, energy-efficiency, and code size of configurable processors. While generating new instructions from application-specific operation patterns has been a common way to improve the instruction set (IS) of a configurable processor, automating the design of IS's for given applications poses new challenges. This IS synthesis typically requires these questions to be answered: how to create as well as utilize new instructions in a systematic manner, and how to choose the best set of application-specific instructions taking into account the conflicting effects of adding new instructions? To address these problems, we present a novel IS synthesis framework that optimizes the performance for the given data path architecture through an efficient instruction encoding. We build a library of new instructions with various encoding alternatives and select the best set while satisfying the instruction bitwidth constraint. We formulate the problem using integer linear programming and also present an effective heuristic algorithm. Experimental results using our technique generate instruction sets that show improvements of up to about 40% over the native instruction set for several realistic benchmark applications running on typical embedded RISC processors.

Contents

1 Introduction **4**

2 Motivation **6**

3 Related Work **7**
3.1 Automatic IS Synthesis 7
3.2 IS Extension 8

4 Instruction Set Synthesis Framework **9**
4.1 Instruction Sets 9
4.2 Features 10
4.2.1 Structure-Centric IS Synthesis 10
4.2.2 Optimized Instruction Encoding 11
4.3 Generation of Optimal C-Instructions 12
4.3.1 Creating C-Instructions 12
4.3.2 Selecting C-Instructions 13
4.4 Utilization of C-Instructions in Code Generation 13

5 Creating C-Instructions **14**
5.1 Rescheduling the Operations 14
5.2 Generalizing with Operand Classes 15

6 Selecting C-Instructions **16**
6.1 Instruction Set Selection 16
6.2 ILP Problem Formulation 18
6.3 Heuristic Algorithm 19

7 Experiments **20**
7.1 Experimental Setup 20
7.2 Comparison of ILP and Heuristic Algorithm 21
7.3 Comparison of Basic, Synthesized and Native IS's for SH-3 22
7.4 Comparison of Basic, Synthesized and Native IS's for MIPS 23

8 Conclusion **24**

9 Acknowledgements **25**

A Proof of NP-hardness **28**

List of Figures

1 Customizing the instruction set of an application-specific instruction set processor (ASIP). 5
2 Instruction sets. 9
3 Application-specific instruction set synthesis framework. 10
4 Flexible instruction format and the instruction code space. 11
5 Synthesis flow to generate optimal C-instructions. 12
6 Cycle reduction by rescheduling the operations of a basic instruction sequence. RR, EX, MEM, and WB are the four pipe stages, and the horizontal lines represent the cycles or the control steps; the diagrams illustrate the execution of the instructions as they go through the pipeline, performing their operations at different pipe stages in different cycles. The right-hand side diagram shows that the data dependency, represented by arrows, is still satisfied in the C-instruction *CXX*. . . . 14
7 Creating C-instructions from a basic instruction sequence. 16
8 Heuristic algorithm for C-instruction selection. 29
9 Subroutine *update-benefit*. 30
10 Performance of the native and the synthesized IS's, normalized to that of the basic IS (SH-3) 30
11 Performance of the native and the synthesized IS's, normalized to that of the basic IS (MIPS) 31

List of Tables

1 C-instructions and associated information 17
2 Comparison of ILP and heuristic algorithm (SH-3) 22
3 Comparison of synthesized IS's with the basic & native IS's (SH-3) 22
4 Comparison of synthesized IS's with the basic & native IS's (MIPS) 24

Abstract

Application-specific instructions can significantly improve the performance, energy-efficiency, and code size of configurable processors. While generating new instructions from application-specific operation patterns has been a common way to improve the instruction set (IS) of a configurable processor, automating the design of IS's for given applications poses new challenges. This IS synthesis typically requires these questions to be answered: how to create as well as utilize new instructions in a systematic manner, and how to choose the best set of application-specific instructions taking into account the conflicting effects of adding new instructions? To address these problems, we present a novel IS synthesis framework that optimizes the performance for the given data path architecture through an efficient instruction encoding. We build a library of new instructions with various encoding alternatives and select the best set while satisfying the instruction bitwidth constraint. We formulate the problem using integer linear programming and also present an effective heuristic algorithm. Experimental results using our technique generate instruction sets that show improvements of up to about 40% over the native instruction set for several realistic benchmark applications running on typical embedded RISC processors.

1 Introduction

Configurable processors are application-specific synthesizable processors where the instruction set (IS) and/or microarchitectural parameters such as register file size, functional unit bitwidth, etc. can be easily changed for different applications at the time of the processor design. Easier integration and manufacturing, and architectural and implementational flexibility make them better suited for embedded processors in system-on-a-chip (SOC) designs than off-the-shelf processors [1]. With the commercial offerings of such configurable processors [2, 3] and the recent development in the retargetable compiler technology [4–6] as well as the increased interest in the platform-based SOCs employing configurable processors, the problem of IS customization for application-specific IS processors (ASIPs) is drawing more and more attention from both industry and academia [7–12].

Customizing the IS of an ASIP requires several phases of time-consuming processes (Fig. 1). First, the application program can be compiled and simulated for the initial IS of the processor, generating some profiling results. By analyzing the results, one can identify potentially useful "new" instructions, which can be included in the new IS while there may be some other instructions removed from the old IS. This new IS can then be used to retarget the toolchain (compiler, assembler, simulator, etc.), generating a new set of analysis results. These results can lead to another set of new instructions and the process of customizing the IS continues until the desired goal (performance, code size, etc.) is reached. Thus, even with a retargetable toolchain available, designing an IS optimized for a given application involves complex optimization tasks. Furthermore, when it comes to the automatic design of such IS's, which is called IS synthesis, there are even greater challenges. For instance, how can we generate "new" instructions in a systematic manner, that might help achieve the optimization goal, and also how can we utilize them in later compilation stages? Further, considering the conflicting effects of adding application-specific instructions

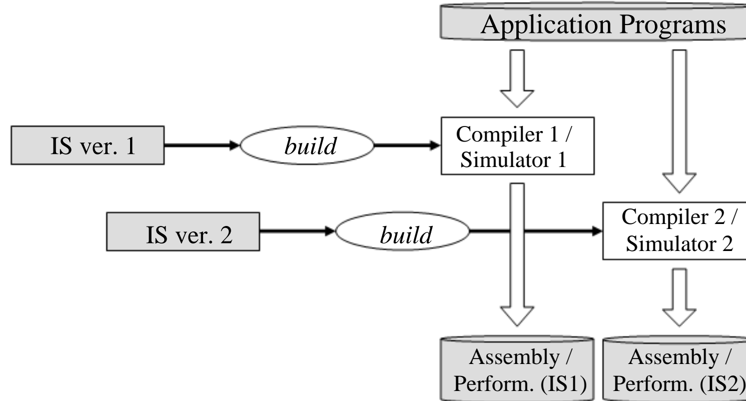

Figure 1. Customizing the instruction set of an application-specific instruction set processor (ASIP).

(potentially increasing the performance; however, at the cost of the increased resources such as the chip area, the cycle time, and the instruction bitwidth), how can we find the best set of such new instructions for the given application as well as the given resources? To address these problems, we need an IS synthesis framework which can, preferably, be used for modern configurable processors.

Since the early 1990's, there have been different approaches to application-specific IS synthesis [13–18]. However, they are often targeted for their own architectural styles (quite different from pipelined RISC architectures), and therefore difficult to apply in the context of RISC-based configurable processors. Also, none of them are designed as a methodology to improve an existing processor, which has become increasingly important in SOC platform-based designs. In this paper, we present an IS synthesis framework that is targeted for modern RISC-based configurable processors, also with advanced features such as multi-cycle instruction support and complex instruction encoding. Our framework takes a structure-centric approach—it tries to improve the processor only by changing the IS for a given data path architecture. In this way we can eliminate the need for costly re-engineering of the entire processor design as well as easily accommodate the need for IS specialization arising in SOC designs. With a fixed data path architecture, customization can be made in such areas as the instruction encoding, the number of (application-specific) instructions, and their definitions, etc. Since such instruction encoding and application-specific instructions can significantly affect the code size, performance, and energy-efficiency, the IS design should be well crafted for any embedded processor that needs to be optimized under limited resources.

Although the proposed IS synthesis framework can generally support different optimization goals, in this paper we focus on performance improvement through IS synthesis. To achieve maximal performance improvement, our technique first builds a library of candidate application-specific instructions with various encoding alternatives satisfying the architectural constraints, and then selects the best set satisfying the instruction encoding constraint coming from the instruction bitwidth limitation. We formulate the problem using an integer linear programming (ILP) framework. But

since solving an ILP problem takes a prohibitively long time even for a moderately sized problem, we also present an effective heuristic algorithm. Experimental results show that our proposed technique can synthesize IS's that generate up to about 40% performance improvement over the processor's native[1] IS for different application domains.

The contributions of our work are three-fold. First, it is aimed at modern RISC pipelined architectures with multi-cycle instruction support—representative of current configurable processors—while most existing methodologies [15, 16, 18] apply to only VLIW-like processors. Second, it tries to improve a given processor hardware through IS specialization, which makes our technique suitable for an emerging class of configurable processors that build on existing, popular ISA families. Third, our technique takes instruction encoding into account so that the obtained IS can be as compact and efficient as those manually designed by experts.

The rest of this paper is organized as follows. In Section 2 we introduce application-specific instruction set synthesis using motivating examples and in Section 3 we summarize related work. In Section 4 we present the IS synthesis framework and outline the synthesis flow. The two major steps in the flow are discussed in more detail in Section 5 and Section 6. In Section 7 we demonstrate the efficacy of our techniques through experiments on typical embedded RISC processors running realistic applications, and conclude the paper in Section 8.

2 Motivation

We illustrate the potential for significant performance improvements using application-specific instructions on a typical embedded RISC processor (the Hitachi SH-3 [19]) and a realistic application (the H.263 decoder algorithm). Initial profiling of the H.263 decoder application shows that about 50% of the actual execution time is spent in a simple function that does not contain any function calls and which consists of only two nested loops, either one of which, depending on a conditional, is actually executed. One of these inner-most loops, when processed by an SH-3 targeted GCC compiler, generates 13 native instructions that executes in 14 cycles (not including branch stall). By devising a custom instruction that takes 6 cycles, the loop is reduced to 4 instructions that execute in 9 cycles. Alternatively, the entire loop can be encoded into a single custom instruction that executes in 7 cycles. In this example, it is obvious that greater performance enhancement and code size reduction can be obtained by introducing custom instructions than by relying on traditional compiler optimizations or assembly coding.

However, those two custom instructions require 4 and 5 register arguments respectively: since one SH-3 register argument requires 4 bits, each of these seemingly attractive custom instructions cannot fit into a 16 bit instruction. On the other hand, if we fix the positions of register arguments for these custom instructions, we do not need to specify arguments and only the opcodes (operation codes) need to be specified (similar to a function call with fixed register arguments). Furthermore, such instructions are easily handled by compilers in a manner similar to function calls. Although such custom instructions can be used very effectively and do not cause an argument encoding problem, they are only suitable for the hot spots of the application, where the introduction of custom instructions can be justified by their heavy dynamic usage counts.

[1] By "native" we mean the existing IS for a processor family.

Another way to improve performance through IS specialization is to combine frequently occurring operation patterns into complex instructions, where a complex instruction is an instruction with more than one micro-operation. Such a methodology is particularly useful since it automates the task of finding promising patterns over the entire application program. Even though there have been similar works for VLIW-like architectures [16, 18], to our knowledge no prior work has addressed pipelined RISC architectures. VLIW-like architectures have ample hardware resources supporting multiple parallel operations. Therefore performance improvements can easily be obtained by defining and using a complex instruction with multiple parallel operations rather than using a sequence of simple instructions. But a typical RISC architecture has little or no instruction level parallelism and thus cannot benefit from parallel operations combined into one complex instruction. However, even in RISC architectures with no explicit instruction level parallelism, we can exploit the parallelism in between the pipeline stages: auto inc/decrement load/store are typical examples. Many other possibilities that combine multiple operations from different pipeline stages are also feasible, and can contribute to performance improvements over the processor's native IS.

Due to the instruction bitwidth constraint, however, instruction sets may not have the full power to express all the possible combinations of the operations in the hardware. Furthermore, each processor family has its own IS quirks. For instance the SH-3 has a two-operand IS with an implicit destination: a typical SH-3 ADD instruction assumes the destination is the same as one of the source operands. This instruction encoding restriction adversely affects the performance, which further motivates the need for an application-specific instruction encoding method.

One practical consideration is that it is very difficult to add any new complex instructions into an existing IS, due to the limited size of the IS code space. One possibility is to use "undefined" instruction opcodes, which, however, often provide too small a code space for complex instructions. Another way to solve this problem is to define a *basic IS* using only a part of the instruction code space. Complex instructions can then be created in an application-specific manner and added into the remaining code space. To gain performance benefit in this manner, the generated complex instructions should be optimized in terms of their encoding so that more of those instructions can be located within the limited instruction code space. These observations motivate our code space-economical instruction set synthesis technique.

3 Related Work

We summarize the related work in two categories: instruction set synthesis and ISA (Instruction Set Architecture) enhancement. The former deals with the automatic generation of instruction sets for specific applications and shares a similar context with this paper. The latter includes manual customization of existing ISAs for specific applications and sometimes exploits similar opportunities.

3.1 Automatic IS Synthesis

The synthesis of application-specific IS's have been approached in different ways: application-centric and structure-centric, depending on which part of the ISA is decided first. In application-centric approaches [16–18], the IS is first optimized from the application's behavior using the

techniques similar to [20]. The hardware is later designed to implement the instruction set, manually or automatically. Among the approaches in this direction, PEAS-I [17] is most similar to our approach in that both approaches assume a basic IS and target pipelined RISC architectures. However, PEAS-I has a fixed set of instructions from which a subset is selected; thus instruction encoding is never an issue, unlike in our approach. Other approaches, however, tend to presume their own architectural styles (e.g., 'transport triggered architecture' in [18]) and thus cannot be applied to modern pipelined RISC processors. More importantly they do not give any hints on how to improve an existing processor architecture, which can be very helpful particularly in the context of configurable processor-based SOC design.

Structure-centric approaches [13–15] have a structural model of the architecture either implicitly assumed or as an explicit input and try to find the best instruction set matching the application. This approach has the advantage that it can leverage existing processor designs. However, previous work in this direction has not fully addressed the issue of instruction encoding: opcode and operand fields all have fixed widths. As a result, the instruction bitwidth cannot be fully utilized as in custom processors, leading to limited performance improvement for the same bitwidth or increased instruction bitwidth and code size. Our approach is significantly different, since multiple field widths are allowed for the same operand type, with different benefits and cost profiles, so that the optimal set of complex instructions, depending on the application, can be selected satisfying the instruction bitwidth constraint.

3.2 IS Extension

The application-specific ISA customization has received much attention recently as commercial configurable processors become available. For instance, Zhao *et al.* [7] demonstrated $2 \sim 4$ times performance improvement in a case study with a commercial configurable processor core. They expanded the data path width through configuration and added several custom instructions such as bit operations, compound instructions, and parallel processing on the widened data path. Our approach is significantly different, since it automates the design of application-specific instructions, aims to improve the processor by changing only the IS (minimizing the change in the data path), and generates the instructions that can easily be supported by compilers without re-coding the program.

The bitwidth optimization technique in this paper has similarity with other bitwidth reduction techniques. Typically in synthesizing application-specific hardware from behavioral description, the smallest bitwidth can be chosen for data path components to achieve smaller area and lesser energy consumption [21, 22]. Even instruction set processors sometimes have reduced-bitwidth instruction set architecture (rISA), of which the main benefit is smaller code size. Examples of the rISA instruction set include Thumb [23] of the ARM7TDMI processor and MIPS16 [24] of the MIPS processor. Recently, Halambi *et al.* [25, 26] have proposed compilation and design space exploration techniques to take advantage of the rISA more effectively. Further extending the rISA scheme, Kwon *et al.* [27] proposed a new ISA called PARE (Partitioned Registers Extension), which is an enhanced version of the Thumb ISA. Using the concept of the partitioned register file, they could further reduce the bitwidth of register operand fields and increase that of immediate or opcode field, thereby reducing the entire code size. In this paper we deal with the problem of

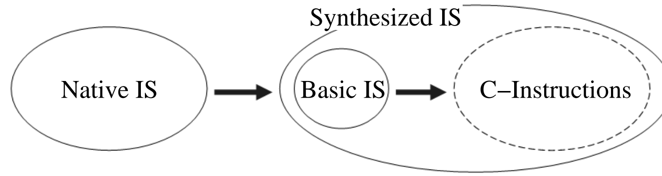

Figure 2. Instruction sets.

synthesizing instruction sets that are optimized in the bitwidth of each operand and opcode field, and also competitive (in terms of the performance) to the native IS by supporting application-specific complex operation patterns with the synthesized instructions.

4 Instruction Set Synthesis Framework

We now present our IS synthesis framework that improves the processor architecture through IS specialization. We introduce various IS's, present the salient features of our framework, and outline the IS synthesis flow.

4.1 Instruction Sets

To be able to generate application-specific instructions automatically, we need to define "instructions" and how a "new" instruction can be defined. Some of the previous ASIP synthesis approaches [17] have assumed a set of potential "special" instructions, out of which an application-specific set of special instructions can be defined for each application. In those approaches, a new instruction can be chosen only from the predefined pool of instructions; therefore, the generation of new, application-specific instructions is limited by, not only dependent on, the predefined set of special instructions. In our IS synthesis framework, instead of explicitly defining the set of possible new instructions, we define a basic IS once for each processor. Using the resource usage and timing information of basic instructions, combinations of basic instructions can be generated with possibly less number of execution cycles or control steps than their basic instructions versions. Instructions built in this way are called *C-instructions*,[2] which are the result of our IS synthesis flow.

Fig. 2 illustrates the relationship between the three IS's relevant to our IS synthesis approach. The native IS is the existing IS of the processor, which is presumably optimized for general applications and may include its own complicated instructions as well as simple ones. From the native IS, the basic IS is defined, which serves three purposes. First, a basic instruction is defined to have only one micro-operation, to facilitate the construction of C-instructions. Second, the basic IS, together with the resource usage and timing information of each basic instruction, represent the processor data path in a ready-to-use form for the IS synthesis. This information includes the number of each resource type (functional units, ports, buses) that may be used at each pipeline stage. Third, the basic IS is defined to have the simplest instruction format, so that the saved code space may be used for new instructions generated for each application. The union of the basic IS and the C-instructions generated for each application is called the synthesized IS for the application.

[2]C-instruction means compound-operation instruction, that is, an instruction with more than one micro-operation.

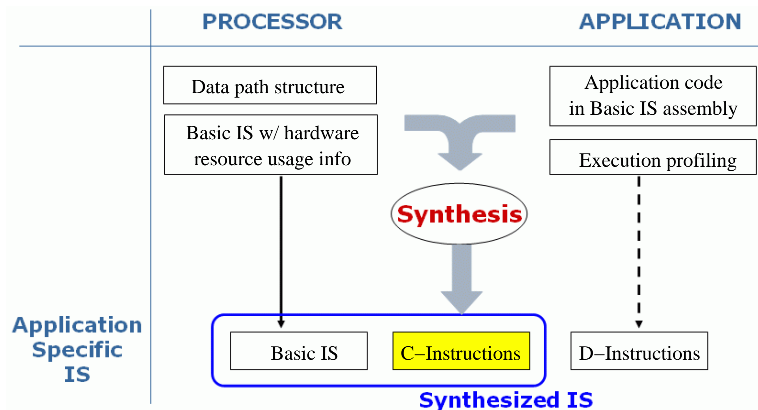

Figure 3. Application-specific instruction set synthesis framework.

In addition to the C-instructions, users may want to have custom instructions that are designed by hand, often with the designers' intuition. In our IS synthesis framework, these instructions are called *D-instructions*, which may be difficult to discover with an automatic method unlike with C-instructions. These needs arise especially when the application-specific instruction is very complex as in FFT (Fast Fourier Transform) operations, or special functional units can be afforded. Although these instructions are very effective in boosting the application performance, they pose very difficult problems for automation, in both the instruction design phase and the code generation phase of compilers. Our IS synthesis methodology deals only with the automated generation and application of C-instructions for application-specific IS processors.

4.2 Features

In our IS synthesis scheme, an application-specific IS consists of the basic instructions, C-instructions, and D-instructions as illustrated in Fig. 3. While the basic instructions are provided by the user as a part of the architecture description, the C-instructions are synthesized using the information from both the data path architecture and the application program. The D-instructions, which may be generated manually by an expert, take effect on the code space allocated for the C-instructions. In this subsection, we highlight the two main features of our proposed IS synthesis framework.

4.2.1 Structure-Centric IS Synthesis

Currently, a typical approach to the application-specific IS synthesis is first identify the most promising operation patterns from the application and later implement the application-specific instructions using additional hardware such as special functional units. However, in that approach it is difficult to find an optimal IS (e.g., of maximal performance for the given power, area budget) due to the difficulty of foreknowing the hardware implementation characteristics during the synthesis step. As a result, an iterative optimization strategy is unavoidable, which may involve hardware synthesis for the IS design. Also, since the data path is heavily changed for the new IS, the huge cost of re-engineering the entire processor design is another disadvantage of this approach.

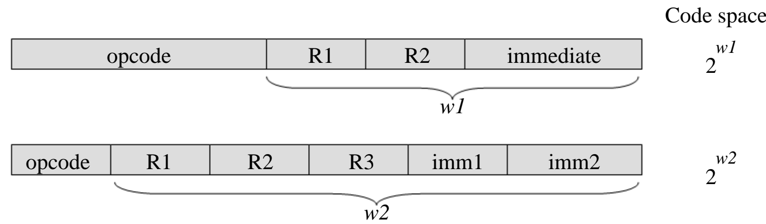

Figure 4. Flexible instruction format and the instruction code space.

Our IS synthesis framework takes a structure-centric approach. In the structure-centric IS synthesis, an optimal IS can be more directly found within the limited hardware change allowed, as the data path structure is given and frozen from the beginning of the IS synthesis process. The IS synthesis system tries to find a better IS exercising the data path more efficiently for the given application. Hence, our IS synthesis framework explicitly takes inputs of the data path architecture as well as of the application program. Another benefit of this approach is that it readily enables IS specialization—improving the processor architecture through only the IS modification—which has become important in the platform-based SOCs employing configurable processors.

4.2.2 Optimized Instruction Encoding

Previous work on IS synthesis has taken a very simplistic view on the instruction format; typically, all synthesized instructions have the same format, with the number of (maximum) operands and their bitwidths determined before the IS synthesis. However, custom-designed IS's of embedded processors have varying formats for different instructions in order to maximize the utilization of the limited instruction bitwidth, and even encoded operands are often used. For example, an immediate field may take 8 bits in the ADDI instruction while it may take only 4 bits in the ADD+LOAD type instructions. Register fields may also have a reduced width to allow more operands to be encoded, at the cost of accessing only a subset of the registers in that instruction. For example, the SH-3 microprocessor has complex load instructions, which have the (implicit) R0 register as their destination so that the destination register need not be specified at all.

Our IS synthesis framework addresses this problem of diverse instruction formats through the *instruction code space* and the *operand class*. First, to allow different instruction formats in a single IS, our framework does not constrain the opcode/operand field widths; rather, the operand fields may have different bitwidths (determined by the synthesis process itself) and only the opcode field width is later set to use the remaining bits of the instruction bitwidth. Thus, the only condition to be satisfied in this scheme is that the sum of the code space (defined as 2 ^{the number of bits needed for operands}) of every instruction should not exceed the allowed total code space (defined as 2 ^{the instruction bitwidth}) (see Fig. 4). Obviously, this condition guarantees that every instruction will be given an opcode, possibly with a different field width. Second, to support the operand encoding as well as the multiple choices for operand field widths, our technique employs the concept of operand class, which encapsulates the field width, the set of compatible operand instances, and an (optional) encoding scheme. The operand classes provide a convenient means to generalize the operands encountered in the assembly code of the application into various versions

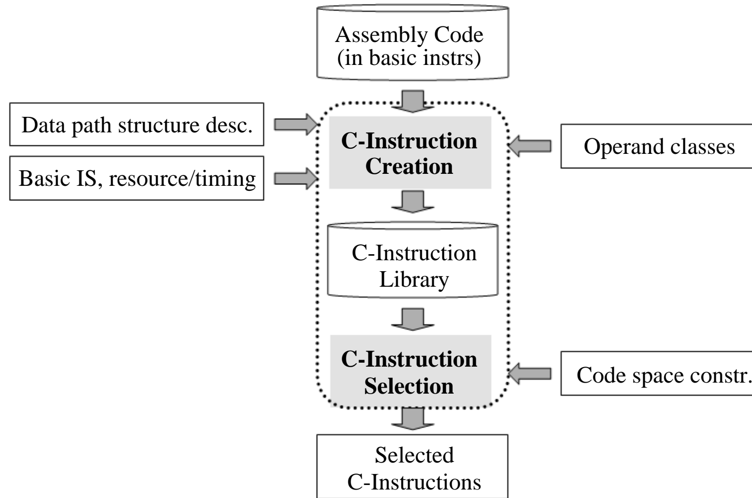

Figure 5. Synthesis flow to generate optimal C-instructions.

of C-instruction operands, which result in different code space requirements. The instruction code space and the operand classes enable the IS synthesis framework to flexibly and efficiently manipulate the instruction formats within the given instruction bitwidth and further to optimize them according to the application's characteristics.

4.3 Generation of Optimal C-Instructions

Now, the goal of the IS synthesis is to generate an optimal set of C-instructions for a given application, satisfying both architectural and instruction encoding-related constraints (see Fig. 3). Initially, the application program is presented in basic IS assembly code, which can be generated by a compiler targeted for the basic IS. From this preliminary assembly code, an application-specific IS is synthesized in two major steps: (1) creating all possible C-instructions and (2) selecting the most profitable ones, as shown in Fig. 5.

4.3.1 Creating C-Instructions

In this phase, we build a library of all the C-instruction patterns that can be created from the application and which may contribute for the optimization goal (such as performance). A C-instruction is essentially a condensed and generalized form of a basic instruction sequence. Thus, for every sequence of the basic instructions (of a limited length) appearing in the application's assembly code, we generalize the sequence making into C-instructions if it can be condensed into a C-instruction with a shorter latency than the basic instructions sequence. We can find out the latency of a C-instruction and the execution cycle saving, by scheduling the operations of the basic instructions with the resource usage (of each instruction) and the resource constraints (of the entire processor) taken into account. If the saving is positive, we create a number of C-instructions varying in generalization (by applying different operand classes), and add them to the C-instruction library. We describe this phase in more detail in Section 5.

4.3.2 Selecting C-Instructions

In the second phase, we select the most profitable set of C-instructions from the library built in the first phase. Though every C-instruction is beneficial for the optimization goal, due to the code space limitation only a subset of the C-instructions may be included in the final set. This leads to an intuitive definition of the benefit (how much does it increase the optimization goal) and the cost (how much code space is used) of a C-instruction. Using the profiling information of the application we can calculate the benefit and the cost of a C-instruction easily. Then, the C-instruction selection can be seen as an optimization problem where the objective is to find the set of C-instructions maximizing the collective benefit within a given code space limit. However, as soon as we start dealing with sets of C-instructions, the situation becomes more complicated because the benefit of a set may not be equal to the sum of the benefits of each. In Section 6 we give an exact treatment of this problem with an ILP formulation and also present an efficient heuristic algorithm for it.

4.4 Utilization of C-Instructions in Code Generation

The selected C-instructions and the basic IS comprise the synthesized IS as illustrated in Fig. 2. While the basic IS is a reduction of the native IS extracting only the essential instructions, the set of selected C-instructions is an extension of the basic IS optimized for each application. Being defined in the code space saved by the reduction to the basic IS, the C-instructions do not require any more code space or instruction bitwidth than the native IS. Only the instruction decoder and the control path (such as muxes controlling the input to functional units) need to be modified. Along with the ASIP modification, the software toolkit can also be retargeted for the synthesized IS as in Fig. 1. However, modifying the application source code is not needed to utilize the C-instructions, unlike with manual IS customization such as [7] and D-instructions (Fig. 3); in the latter case, very complicated patterns such as functions or loops can be made into application-specific instructions, making it hard to identify the patterns without user hints.

Our IS synthesis framework allows for an easy utilization of C-instructions in the instruction selection phase of the compilation through the use of the basic IS. Recall that a C-instruction is a condensed and generalized form of a basic instruction sequence. Therefore, we can substitute a C-instruction for any corresponding basic instruction sequence appearing in the application code. Thus, a simple strategy to utilize C-instructions is to first compile the application using only the basic IS and then substitute C-instructions for the corresponding basic instruction sequences. In this way we can retarget the instruction selection phase very easily.[3] However, the outcome of the instruction selection may depend on the order of the substitutions being made, since the same basic instruction (in the code) may belong to multiple sequences of different C-instructions. We may consider an optimal algorithm for this in order to maximize the benefit of the C-instructions, but here we use a simple scheme for the sake of faster compilation as well as a more predictable outcome after the instruction selection phase. Our scheme is simply to give a higher priority to a C-instruction with larger per-use contribution, i.e., more cycle count saving per use. Using this

[3] After this substitution phase, some phases such as register allocation or instruction scheduling may need to be run again.

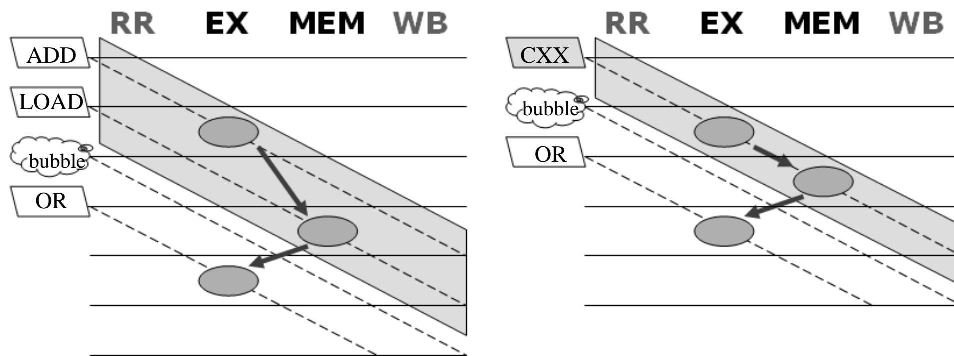

Figure 6. Cycle reduction by rescheduling the operations of a basic instruction sequence. RR, EX, MEM, and WB are the four pipe stages, and the horizontal lines represent the cycles or the control steps; the diagrams illustrate the execution of the instructions as they go through the pipeline, performing their operations at different pipe stages in different cycles. The right-hand side diagram shows that the data dependency, represented by arrows, is still satisfied in the C-instruction *CXX*.

scheme greatly helps with estimating the effective benefit of a set of C-instructions and is crucial to the heuristic selection algorithm in Section 6.

5 Creating C-Instructions

In this phase, we create C-instructions from the application assembly code by (1) condensing (rescheduling the operations of) basic instruction sequences and (2) generalizing (replacing) the operands with compatible operand classes, which are described in the following subsections. A group of C-instructions is created for every sequence of at most N basic instruction instances (N is a design parameter) if it is possible to reduce the number of execution cycles through rescheduling. Since we create C-instructions from only sequentially appearing instruction instances, the number of created C-instructions increases only linearly in the code size of the application. Created C-instructions are compared with those in the library and the library is updated either by including the new C-instruction or by increasing the reference count of the matched one in the library. At the end of this pattern generation step, the library contains distinct C-instructions, from which the most profitable set of C-instructions is selected to define an application-specifically synthesized IS.

5.1 Rescheduling the Operations

The objective of this step is to find out how many execution cycles can be saved by rescheduling the operations of basic instructions and making a C-instruction. Although the target architecture is a RISC pipeline, (where there is not so much parallelism as in VLIW processors) we can exploit the parallelism in between the pipeline stages. Moreover, unlike in VLIW processors where each pipeline is simple and there are a small number of operation combinations (in each pipeline) to be included in the IS, modern pipelined processors often have such large combinations of micro-

operations that they are difficult to encode using the limited instruction bitwidth. Therefore, especially as the processor data path becomes more complex, the need for optimal instruction encoding and perhaps application-specific approaches also increases.

```
ADD    R1  R1   4
LOAD   R1  (R1)
OR     R1  R1
```

Fig. 6 shows a very simple example of rescheduling the operations of basic instructions for a assembly code sequence shown above. The left-hand side diagram shows the optimal (shortest-latency) execution for the assembly code, which is also optimally scheduled. In this case, the first two instructions effectively takes three execution cycles, which is not due to the data dependency or the lack of resource; rather, it is because there is no such instruction that can represent the two micro-operations with the same instruction. Therefore, if we make a C-instruction with the two operations, we can remove the unnecessary instruction fetch, which translates into one execution cycle saving as in the right-hand side. Note that the last instruction (OR) is shown to ensure that there is no negative effect of using the C-instruction rather than the basic instructions version.

As illustrated by the example, we can evaluate the savings (Cycle Reduction, or *CR*) of a C-instruction, by scheduling the operations of the basic instructions with the resource constraint (of the data path) taken into account. While we assumed that there are data forwarding paths available for all stage-pairs in the example, if only limited paths are available (which should be specified as a constraint in the data path architecture), it should also be taken into account during the scheduling. More details about the scheduling and resource allocation of operations graphs can be found in [28], for instance.

5.2 Generalizing with Operand Classes

Fig. 7 illustrates the process of generating a C-instruction from a sequence of basic instruction instances. Fig. 7 (a) shows a conceptual view of the two sequences of basic instructions substituted by two C-instructions. An example of operand classes is shown in Fig. 7 (b), where '#Bits' is the number of bits needed for encoding operands. Fig. 7 (c) shows a group of C-instructions created from a sequence of basic instructions, where '#Bits' is the same as before, and *CR* stands for the benefit of a C-instruction in terms of the cycle count saving.

Note that operand classes with smaller bitwidths have not only less cost in terms of bitwidth (and hence code space) usage, but less benefit also. To wit: (1) immediate fields of smaller size have a smaller chance to be matched in other parts of the application program and (2) register fields of smaller size may potentially necessitate an additional *move* instruction, which decreases the benefit of the C-instruction. C-instructions with such a register operand class have their *CR* value (cycle count reduction by a single use of the C-instruction) reduced by the probability of necessitating an additional *move* instruction (in Fig. 7 (c), this probability is assumed to be 0.75).[4]

[4]Register operand classes with reduced bitwidth rely on the compiler's register allocation capability. Operand classes with a single register can be dealt with relatively easily while those operand classes with multiple but not all registers require more complex register allocation.

(a) Basic Instructions

```
ADDI   R1 R1 4
LOAD   R1 (R1)
       ⇒ ADDI+LOAD   R1 (R1 + 4)
MULT   Mac R1 R1
MOV    R1 Mac
LOAD   R2 (R3)
ADD    R1 R1 R2
       ⇒ MULT+MOV+LOAD+ADD   R1 R2 R3
```

(b) Operand Classes

Type	Operand Classes	Instances	#Bits
REG	[Gen_reg]	R0 ∼ R15	4
	[Frame_pointer]	FP	0
	[R0_implicit]	R0 ∼ R13	0
IMM	[Imm_const4]	4	0
	[Imm_4bits]	$-8 \leq IMM < 7$	4
	[Imm_6bits]	$-32 \leq IMM < 31$	6
DISP	[Disp_normal]	$-2^7 \leq DISP < 2^7$	8
	[Disp_short]	$-2^3 \leq DISP < 2^3$	4

(c) C-Instructions

Basic Instruction Sequences:

```
ADDI   R1 R1 4
LOAD   R1 (R1)
```

C-Instructions Created:

C-Instruction	Operands	#Bits	*CR*
ADDI_LOAD_1	[Gen_reg] [Imm_const4]	4	1
ADDI_LOAD_2	[Gen_reg] [Imm_4bits]	8	1
ADDI_LOAD_3	[Gen_reg] [Imm_6bits]	10	1
ADDI_LOAD_4	[R0_implicit] [Imm_const4]	0	0.25
ADDI_LOAD_5	[R0_implicit] [Imm_4bits]	4	0.25
ADDI_LOAD_6	[R0_implicit] [Imm_6bits]	6	0.25

Figure 7. Creating C-instructions from a basic instruction sequence.

6 Selecting C-Instructions

6.1 Instruction Set Selection

Every C-instruction in the library has the following information for instruction set selection: the number of bits needed for operands, the number of cycles saved by a single use of the C-instruction (*CR*), and a list of basic instruction instances covered. Using the information together with the repetition count of each basic block (which can be supplied through profiling), the total number of cycles reduced by a C-instruction can be estimated.

C-instructions contribute to cycle count reduction at the cost of code space. Therefore the problem is to select the set of C-instructions that maximizes the cycle count reduction across the entire application program while respecting the code space constraint. Let $\{CI_i\}$ be a set of C-instructions selected, W_i be the bitwidth needed for the operands of a C-instruction CI_i, and x_i be a binary variable which is '1' if CI_i is selected. Then the code space constraint implies

$$\sum x_i \cdot 2^{W_i} \leq CS^C, \tag{1}$$

where the total code space for C-instructions CS^C may be given as $2^{IBW} - \sum 2^{B_i}$, with IBW being the instruction bitwidth (e.g., 16 and 32) and B_i the number of operand bits of each basic instruction. D-instructions, if any, similarly decrease the total code space CS^C. Therefore, the problem of selecting the most profitable C-instructions reduces to mapping $\{x_i\}$ to $\{0,1\}$ satisfying (1) such that the total cycle count reduction from the selected CI_i's is maximized.

Table 1. C-instructions and associated information

C-Instruction	#Bits	*CR*	Covered Basic Instruction Instances	*Benefit* (Total *CR*)	*Cost* (Code space)
CI_1	4	1	$(a_1\ a_2\ a_3)\ (a_9\ a_{10}\ a_{11})$	30	2^4
CI_2	6	1	$(a_1\ a_2\ a_3)\ (a_9\ a_{10}\ a_{11})\ (a_{13}\ a_{14}\ a_{15})$	50	2^6
CI_3	5	2	$(a_3\ a_4\ a_5)$	30	2^5

Table 1 shows a simple example of C-instructions and the associated information. In the fourth column each a_i represents a basic instruction instance (e.g., one line of assembly code). The benefit of a C-instruction is defined as the sum of the *CR*s (Cycle Reductions) from all instances of basic instruction sequences covered, with the block repetition count taken into account. The last column shows the cost of selecting the C-instruction in terms of the code space taken.

The benefit and cost may change as C-instructions are selected. This complication arises due to two factors: *superset* instructions and *multiple covers*. In Table 1, CI_2 is a *superset instruction* of CI_1 because CI_2 has the same opcodes and operands as those of CI_1 but has more general operand classes.[5] Therefore CI_2 covers more basic instruction instances but also requires more bits for representation. In this case, CI_1 is a special case of CI_2, so if CI_2 is included in the set of selected C-instructions, say S^C, then CI_1 is in effect already included with no cost. On the other hand, the cost of CI_2 should be decreased to $2^6 - 2^4$ when it is known that CI_1 has been selected. When there are more than one superset instructions of a C-instruction, however, only one of them should have the decreased cost.

Multiple covers occur when more than one C-instruction covers the same basic instruction instance. In Table 1, a_3 is covered by all the C-instructions. Obviously only one C-instruction can actually be substituted for a_3 (and the neighboring basic instruction instances), which leads to the decrease in their collective benefit. Assuming a possible compiler optimization pass that applies, in a predefined order, substitutions of C-instructions for certain basic instruction patterns, multiple

[5] An operand class is more general than another if the set of its instances includes that of another.

covers lead to the reduction of effective benefit of those with lower priority. In Table 1, if only CI_1 and CI_3 have been selected for S^C and CI_3 has higher priority than CI_1, the total benefit (number of reduced cycles) becomes 45 instead of the sum of all the benefits. In other words, assuming CI_3 has higher priority than CI_1, the benefit of including CI_3 in S^C when CI_1 is already in S^C is 15 $(= 45 - 30)$, where 30 is the benefit of CI_1.

In Appendix A, we show that a special case of this problem, where no superset instructions or multiple covers take place, can be reduced to a knapsack problem (which is NP-hard) if the *cost* value of C-instructions can be any positive integer. Since the original formulation allows the *cost* to be only $2^n (n \geq 0)$, there remains a possibility that this problem turns out to be tractable, yet we have not found any such algorithm and it seems highly unlikely as well.

6.2 ILP Problem Formulation

For the description of the problem formulation, we first introduce the following variables. Let the C-instruction library be given as $\{CI_i \,|\, i = 1, 2, \cdots, n\}$ and each instruction has code space information $s_i = 2^{W_i}$, where W_i is the number of bits needed for operands encoding. With each C-instruction (CI_i) is associated a set of basic instruction instances sequences $\{BII_{ij} \,|\, j = 1, 2, \cdots, m_i\}$, each member of which matches CI_i. Lastly, G_{ij} is the benefit or expected cycle count reduction by replacing BII_{ij} with CI_i. G_{ij} can be defined as in the following equation,

$$G_{ij} = Repetition_{ij} \cdot CycleReduc_i \tag{2}$$

where $Repetition_{ij}$ is the repetition count of the block, $CycleReduc_i$ is the CR of CI_i. Note that the CR value has been compensated for the penalty or probability that CI_i necessitates an additional *move* instruction when CI_i has register operand generalized with reduced field size.

The C-instruction set selection problem can be seen as selecting BII_{ij}'s, for which corresponding CI_i's can be substituted. Here the actual compiler that will use those C-instructions is assumed to be able to find the optimal set of BII_{ij}'s from a sequence of basic instruction instances. Now let us define the following binary variables for the ILP problem formulation:

- $A_{ij} (i = 1, \cdots, n; j = 1, \cdots, m_i)$: 1 if BII_{ij} is selected.
- $C_i (i = 1, \cdots, n)$: 1 if CI_i is selected, that is, if any of $A_{ij} (j = 1, \cdots, m_i)$ is 1.
- $X_i (i = 1, \cdots, n)$: 1 if CI_i is selected and all its superset instructions are not selected.

Then the objective function (the expected total cycle count reduction) and the code space constraint can be represented as

$$\max: \sum_{i=1}^{n} \sum_{j=1}^{m_i} G_{ij} A_{ij} \tag{3}$$

$$\sum_{i=1}^{n} s_i X_i \leq CS^C \tag{4}$$

The relationship between the binary variables can be represented as

$$C_i = A_{i1} \vee A_{i2} \vee \cdots \vee A_{im_i}\,, \quad \text{for } ^{\forall} i \tag{5}$$

and by letting CI_i's superset instructions be $\{CI_{s_1}, CI_{s_2}, \cdots, CI_{s_k}\}$,

$$X_i = C_i \wedge \overline{C}_{s_1} \wedge \overline{C}_{s_2} \wedge \cdots \wedge \overline{C}_{s_k} \tag{6}$$

or equivalently

$$\overline{X}_i = \overline{C}_i \vee C_{s_1} \vee C_{s_2} \vee \cdots \vee C_{s_k}\ , \ \text{ for } {}^{\forall} i. \tag{7}$$

Note that (5) and (7) can easily be linearized using simple identities as

$$\overline{X} \Longleftrightarrow 1 - X \tag{8}$$

$$C = A_1 \vee A_2 \vee A_3 \Longleftrightarrow C \geq A_1,\ C \geq A_2,\ C \geq A_3,\ C \leq A_1 + A_2 + A_3. \tag{9}$$

Finally, the constraint to avoid the multiple covers can be stated as follows:

> If there is more than one basic instruction instances sequence covering a basic instruction instance, then only one of them can be selected.

In other words, if a basic instruction instance is covered by these $BII_{I_1J_1}, BII_{I_2J_2}, \cdots, BII_{I_xJ_x}$, then

$$A_{I_1J_1} + A_{I_2J_2} + \cdots + A_{I_xJ_x} \leq 1\,, \tag{10}$$

which holds for every basic instruction instance.

The above (3), (4), (5), (7), and (10) define the ILP formulation for the IS selection problem. But since solving an ILP takes a prohibitively long time even for a moderately sized problem, we also present a heuristic algorithm in the next subsection.

6.3 Heuristic Algorithm

Fig. 8 shows our heuristic algorithm proposed for the problem of C-instruction selection and ordering based on the above observations. In the algorithm listings, *B* is used for a basic instruction instance while *C* is used for a C-instruction. So, for instance, Line 2 indicates *MoreGeneralForm* is a function mapping a set of C-instructions for every C-instruction.

The algorithm works by repeatedly selecting the most promising C-instruction (i.e., the one with the largest benefit per cost ratio). The ordering of the selected C-instructions is decided by their *CR* values (Line 32 of Fig. 8): the greater the *CR*, the higher the priority. Among those with the same *CR* value, priority follows the order in which they were selected. In the algorithm description, *B* and *C* (and their indexed versions) represent a basic instruction instance and a C-instruction instance, respectively.

Superset Instructions: Every C-instruction has a set of pointers to more general C-instructions (*MoreGeneralForm* in Line 2). A C-instruction is more general when all operands are encoded with more general or equal operand classes. This set can be built up as new C-instructions are created and added to the library. When a C-instruction is selected, its cost is subtracted from the cost of each of the more generalized C-instructions (Lines 12 – 14).

Multiple Covers: If a basic instruction instance is covered by more than one C-instruction and those C-instructions are all selected, the total benefit of the selected instructions is less than the sum

of each benefit. To accurately quantify the benefit of selecting a new C-instruction under the assumption that a compiler substitutes C-instructions for matching patterns in a predefined order, the algorithm introduces two new integer variables for every basic instruction instance. *MaxCycleReduc* (Line 3) of an instruction instance is the maximum of *CR*'s (*CycleReduc*) of the C-instructions that cover the instruction instance and have been selected so far. *CInstrInst* (Line 4) is the instance ID of that C-instruction with the *MaxCycleReduc*. Because the selected C-instructions are ordered by the *CycleReduc* value, a C-instruction (C_j) can only be used when all the instruction instances in the matched basic instruction sequence have *MaxCycleReduc* value of less than *CycleReduc* of C_j. Therefore as a C-instruction is selected, other C-instructions that share some of the covered basic instruction instances (*InstancesCovered* in Line 16) are affected in their benefit. The effective benefit of choosing one (C_i) is lowered (Fig. 9) because: (1) for instances that already have greater *MaxCycleReduc* value, C_i will not be used (because it is less effective), and (2) for instances that have less *MaxCycleReduc* value, C_i will be used replacing other C-instructions with lower *CycleReduc*, which are already selected and would be used if C_i is not selected.

The complexity of this algorithm is $O(L^2 \cdot M)$, where L is the library size (i.e., the number of C-instructions in the library) and M is the average number of basic instruction sequences covered by a C-instruction. M can be as large as the total number of basic instruction instances (i.e., code size) but is typically much smaller than that. This complexity comes from the *update-benefit* part, which iterates over M basic instruction sequences covered by C_i and can be called as many as L^2 times.

7 Experiments

In this section we present the experimental results showing the effectiveness of the proposed heuristic algorithm and applying our technique to improve the native instruction sets of the SH-3 [19] and the MIPS [29] processors. For deriving C-instruction sets, a number of realistic benchmark applications were used covering multimedia (e.g., H.263 decoder, JPEG), control-intensive (e.g., ADPCM) and cryptography (e.g., DES) domains.[6] After describing the experimental setup, we first show the effectiveness of the heuristic algorithm using small benchmark programs. Then for each of the two architectures, we compare the synthesized IS with both the basic IS and the native IS.

7.1 Experimental Setup

For our experiments, we used the SH-3 and the MIPS processors as representative architectures. The MIPS architecture is one of the most widely used embedded processor cores and the SH-3 architecture has DSP-like features such as auto-increment load and MAC (multiply-accumulation) operation, which are increasingly popular in contemporary RISC cores. Furthermore the SH-3 has an *IBW* (instruction bitwidth) of only 16 bits, which renders it more important to find a better IS for a more effective utilization of the hardware.

[6] H.263 decoder is from the Telenor R&D distribution (ver. 2.0), which is originally based on an implementation by MPEG Software Simulation Group, 1994. ADPCM is from the MediaBench, http://www.cs.ucla.edu/~leec/mediabench/ .

For each architecture, we defined a basic IS detailed with resource and timing information, which can be extracted from the native IS and the structural information (e.g., pipeline configuration). A number of realistic benchmark applications covering multimedia (e.g., H.263 decoder, JPEG), control-intensive (e.g., ADPCM) and cryptography (e.g., DES) domains were processed by the EXPRESS retargetable compiler [5] (targeting the basic IS) to generate preliminary assembly code, which was used for the input of the IS synthesis process.

For the SH-3 architecture (*IBW* is 16 bits), a basic IS was defined with code space of 15,442, which is a little less than 2^{14} or quarter of the total code space. Since about half (2^{15}) of the total code space is used for system control function or reserved for later versions of the processor family, the code space available for C-instructions is about 2^{14}, which was used as the value of CS^C (the code space for C-instructions) in the SH-3 architecture experiments. For the MIPS architecture (*IBW* is 32 bits), a basic IS was defined with code space of 141,742,240, which is less than 2^{28}. Since the native IS of the MIPS takes about 2^{30} of code space, we used $(2^{30} - 2^{28})$ as CS^C in MIPS architecture experiments.[7]

Another parameter N (the number of basic instructions that are considered together in creating C-instructions) was chosen to be 4 (i.e., up to 4 consecutive basic instructions are considered). In defining operand classes, we used statistics information such as frequent values of immediate/displacement or average register pressure, etc. In each processor architecture, one definition of operand classes was used for all benchmark applications except for the DES application.

After the IS synthesis process, the application was recompiled using the same retargetable compiler targeted for the synthesized IS. The execution cycle counts were obtained through a method combining simulation (at the basic block level) and profiling (at the application level). We assumed perfect caches; however, considering the limited-cache-size effect will reveal greater performance improvement with our technique, since our technique effectively reduces the code size as well.

7.2 Comparison of ILP and Heuristic Algorithm

For the comparison of the two proposed instruction selection methods—ILP and heuristic algorithm (HA), we used small benchmark programs so that the ILP solver would terminate within a reasonable amount of time. The basic IS and the C-instructions are taken from the SH-3 architecture model. Each small benchmark program written in C contains only one function other than `main`. For each benchmark program, a pattern library was created and then the two selection methods were applied. For an ILP solver, a public domain software *lp_solve* [30] was used on a Pentium 866 MHz Linux PC. Table 2 shows the results.

In Table 2, the 2nd to 4th columns show the code size (in terms of the number of basic instructions), the number of (unique) C-instructions in the library,[8] and the cycle count with the basic IS, respectively. The 5th and 6th columns show the number of cycles reduced by the selected set of C-instructions. The next two columns show the number of selected C-instructions, and the last column shows the ILP solver runtime in seconds. In all the cases, the heuristic algorithm could find the optimal or a near-optimal solution with a very short runtime (less than 1 second), while the

[7] Since the MIPS architecture has many unused opcodes, we excluded those unused ones from the possible code space of C-instructions for fair comparison between the synthesized IS and the native IS.

[8] C-instructions that require more bits than *IBW* are not added to the library.

Table 2. Comparison of ILP and heuristic algorithm (SH-3)

Benchmark	#instr.	#patt. in the library	#cycle (basic IS)	Exp. total #cycle reduction		#selected instrs		runtime (ILP)
				ILP	HA	ILP	HA	
EncAC	157	135	586062	229674	229674	19	19	129
Quant	67	106	302782	139693	139693	14	15	11
Bound	51	78	141502	47533	47533	10	11	< 1
Zigzag	47	81	174801	70033	70033	11	12	3
Decode	217	193	50932	16706	16704	25	24	89
AdpEnc	195	178	932700	239788	239779	23	25	104

ILP problems sometimes cannot be solved due to rounding errors. This demonstrates the efficacy of our heuristic with respect to an ILP formulation.

7.3 Comparison of Basic, Synthesized and Native IS's for SH-3

To evaluate the effectiveness of the proposed IS synthesis technique, four realistic applications were used as benchmark programs: JPEG encoder, H.263 decoder, ADPCM coder/decoder, and DES (Data Encryption Standard) algorithm. The first two are multimedia applications, the third one is control-intensive, and the last one is a number crunching application with many bit-level operations. For each benchmark program, a different synthesized IS was generated using our heuristic algorithm.

Table 3 compares the results for three different IS's: the basic, the synthesized, and the native IS's. Both the synthesized and the native IS's include the basic IS, but the synthesized IS is specialized for each application while the native IS (the IS of the SH-3 architecture) remains the same for all the applications. In the table, the 2nd to 4th columns show the cycle counts for the three

Table 3. Comparison of synthesized IS's with the basic & native IS's (SH-3)

Benchmark	#cycle (native IS)	#cycle (basic IS)	#cycle (synth. IS)	%Perf. impr. over Basic	%Perf. impr. over Native	#C-instrs	%code space
JPEG	1.63E+6	2.24E+6	1.36E+6	**65.3**	**20.5**	29	99.9
H.263	1.32E+9	1.58E+9	9.62E+8	**64.3**	**37.4**	35	99.2
ADPCM	2.03E+7	2.76E+7	1.96E+7	**41.0**	**3.7**	40	94.5
DES	5.06E+5	6.50E+5	4.36E+5	**48.9**	**16.0**	41	99.5

IS's. The performance improvements of the synthesized IS over the basic and the native IS are shown in the next two columns, where the performance is defined as the inverse of the number of cycles.[9] The last two columns show the number of C-instructions selected for each benchmark program and the code space usage, respectively. Fig. 10 compares the performance of the native and the synthesized IS's normalized to that of the basic IS.

From Table 3, it is evident that C-instructions improve the performance significantly (41 $\sim$ 65 %) compared to the basic IS. Furthermore, our approach generates consistent performance improvements over the native IS. The amount of this improvement, however, depends on the application: it is significant for multimedia applications (nearly 20 $\sim$ 40 %), and positive (4 $\sim$ 16 %) for the other domains.

One of the reasons the synthesized IS gives only a slight improvement for the ADPCM application is that the application is control-intensive and has small basic blocks often with a few instructions. Since the proposed scheme creates C-instructions only within basic blocks, it often cannot find performance-improving C-instructions for those blocks in control-intensive applications. In the H.263 application, one of the reasons the synthesized IS gives especially good improvement is that the application has a number of multiplication operations (which take place in a different pipeline stage than ALU operations in the SH-3 architecture), which makes it possible to exploit the parallelism in between the pipeline stages. On the other hand, all the other applications have very few multiplication operations.

The results in Table 3 also show that, contrary to conventional wisdom, the code size and the performance improvement obtained through the use of application-specific instructions do not always go against each other. This shows that the performance improvement by application-specific instructions is not very dependent upon the size of the application, although it is certain that the amount will diminish as the application grows bigger and more complex.

The C-instructions synthesized using our approach include subword access instructions (e.g., byte load), their combinations with other operations, and several kinds of load-shift and shift-store instructions, as well as those already included in the native IS such as ADDI+LOAD. One of the interesting instructions only found in multimedia applications is the ADD instruction with three different register operands, the absence of which, however, is one of the distinguishing features of the native IS.

7.4 Comparison of Basic, Synthesized and Native IS's for MIPS

For the MIPS architecture, we performed similar experiments using the same benchmark applications. The results are shown in Table 4. For all the applications, both the native IS and the synthesized IS showed a better performance in case of the MIPS than the SH-3, which is quite reasonable considering the difference in the instruction bitwidths. Fig. 11 compares the performance of the native and the synthesized IS's normalized to that of the basic IS.

Since the MIPS architecture has larger code space than the SH-3 architecture, the MIPS native IS does not suffer so much from the instruction encoding restriction as in the SH-3. But the MIPS

[9] Although we didn't include the effects of the clock speed in the performance, we believe that the comparison between the synthesized and the native IS's are fair because native IS's have their own complex instructions as well which can cause the same effects.

Table 4. Comparison of synthesized IS's with the basic & native IS's (MIPS)

Benchmark	#cycle (native IS)	#cycle (basic IS)	#cycle (synth. IS)	%Perf. impr. over Basic	%Perf. impr. over Native	#C-instrs	%code space
JPEG	1.45E+6	2.15E+6	1.23E+6	**74.3**	**17.4**	45	88.4
H.263	1.19E+9	1.63E+9	8.26E+8	**97.3**	**44.2**	70	92.3
ADPCM	1.74E+7	2.44E+7	1.64E+7	**48.7**	**6.0**	45	69.0
DES	3.74E+5	5.73E+5	3.51E+5	**63.2**	**6.6**	56	40.4

defines only simple instructions: composite-type instructions such as auto-increment load are not found in the native MIPS instruction set. Our approach, on the other hand, can use a part of the code space for C-instructions that are frequently used in the applications. The results in Table 4 shows our approach generates consistent performance improvement over the MIPS native IS. The amount of the improvement is similar to that in the SH-3: nearly 20 $\sim$ 40 % for multimedia application and positive (around 6 %) for the other domains.

Because of the larger code space for C-instructions in the case of the MIPS, our technique can achieve more performance improvement provided that the application has many profitable[10] combinations of instructions (as in the H.263 application). One of the noticeable differences between the two architectures is seen in the DES application, where the synthesized IS shows the least improvement over the native IS in the MIPS case. It is because most of the frequent operations in the DES application require the same resource (ALU)—thus, combining them into a new instruction hardly contributes to improving the performance—and there is little room for improving the instruction encoding either, which is different from the SH-3 case.

Overall, the amount of the performance improvement over the native IS is varied depending on both the application and the architecture, but our approach generated a consistent performance improvement for every case experimented. Especially for multimedia applications, our approach shows significant improvements in both architectures.

8 Conclusion

We presented a novel IS synthesis framework employing an efficient instruction encoding method for configurable ASIPs. The technique improves the processor architecture through IS specialization, which makes the technique suitable for the emerging class of configurable ASIPs being deployed in contemporary SOCs and platform-based designs. We formulated the problem using integer linear programming and presented an efficient heuristic algorithm. Our experimental results demonstrate that through the use of efficient instruction encoding, our technique can generate up to

[10]Instructions that use different resources from different pipeline stages are profitable because they may be combined into a C-instruction with a shorter overall cycles.

about 40% performance improvement over the native IS of a typical embedded RISC processor, for different domains of applications. We believe our approach is thus very useful for designers of systems that need customization of programmable engines, but which leverage software investments made in existing RISC-based ISAs.

We have several directions for future work. For instance, currently C-instructions are created from only sequentially appearing basic instructions within the basic block boundaries, which limits the advantage of the proposed technique especially in control-dominated applications. This problem needs to be addressed to get an equally effective IS even for control-dominated applications. We also intend to investigate the effects of combining this IS synthesis approach with other phases of compiler optimization. Finally, an energy-aware instruction set optimization or exploration can be a very interesting direction.

9 Acknowledgements

This research was supported in part by grants from NSF (CCR-0203813 and CCR-0205712), Motorola Corporation and Hitachi Ltd. We also thank members of the EXPRESS compiler team for their assistance.

References

[1] R. Gonzalez, "Xtensa: A configurable and extensible processor," *IEEE Micro*, 2000.

[2] *Tensilica Inc.*, http://www.tensilica.com.

[3] *ARC Cores Inc.*, http://www.arc.com.

[4] *Target Compiler Technologies N.V.*, Leuven, Belgium, http://www.retarget.com/.

[5] *EXPRESS Retargetable Compiler*, University of California, Irvine, http://www.cecs.uci.edu/~aces/.

[6] *LISATek Inc.*, http://www.lisatek.com/.

[7] Y. Zhao, *et al.*, "Matching architecture to application via configurable processors: A case study with boolean satisfiability problem," in *Proc. ICCD*, 2001.

[8] J.-E. Lee, K. Choi, and N. Dutt, "Efficient instruction encoding for automatic instruction set design of configurable ASIPs," in *Proc. ICCAD*, 2002, pp. 649–654.

[9] F. Sun, *et al.*, "Synthesis of custom processors based on extensible platforms," in *Proc. ICCAD*, 2002, pp. 641–648.

[10] A. Mizuno, *et al.*, "Design methodology and system for a configurable media embedded processor extensible to VLIW architecture," in *Proc. ICCD*, 2002, pp. 2–7.

[11] A. Cataldo, "Compiler that converts C-code to processor gates advances," *EE Times*, Oct. 23, 2001, available at http://www.eet.com/story/OEG20011023S0028.

[12] J. Fisher, "Customized instruction-sets for embedded processors," in *Proc. DAC*, 1999, pp. 253–257.

[13] B. K. Holmer, "Automatic design of computer instruction sets," Ph.D. dissertation, Univ. California, Berkeley, 1993.

[14] I.-J. Huang and A. Despain, "Synthesis of application specific instruction sets," *IEEE Trans. on CAD*, 1995.

[15] H. Choi, *et al.*, "Synthesis of application specific instructions for embedded DSP software," *IEEE Trans. on Computers*, 1999.

[16] J. Van Praet, *et al.*, "Instruction set definition and instruction selection for asips," in *Proc. HLS Symposium*, 1994.

[17] A. Alomary, *et al.*, "PEAS-I: A hardware/software co-design system for ASIPs," in *Proc. EURO-DAC*, 1993.

[18] M. Arnold and H. Corporaal, "Designing domain-specific processors," in *Proc. Codesign Symposium*, 2001.

[19] *SH-3/SH-3E/SH3-DSP Programming Manual*, Hitachi, Ltd., http://www.hitachi-eu.com/hel/ecg/products/micro/pdf/sh7700p.pdf, 2000.

[20] F. Onion, A. Nicolau, and N. Dutt, "Incorporating compiler feedback into the design of ASIPs," in *Proc. ED&TC*, 1995.

[21] Y. Cao and H. Yasuura, "A system-level energy minimization approach using datapath width optimization," in *Proc. ISLPED*, 2001.

[22] J. Choi, J. Jeon, and K. Choi, "Power minimization of functional units by partially guarded computation," in *Proc. ISLPED*, 2000.

[23] L. Goudge and S. Segars, "Thumb: Reducing the cost of 32-bit risc performance in portable and consumer applications," in *Proc. COMPCON*, 1996.

[24] "MIPS16: High-density MIPS for the embedded market," Silicon Graphics MIPS Group, Tech. Rep., 1997.

[25] A. Halambi, *et al.*, "An efficient compiler technique for code size reduction using reduced bit-width ISAs," in *Proc. DATE*, 2002.

[26] ——, "A design space exploration framework for reduced bit-width instruction set architecture (rISA) design," in *Proc. ISSS*, 2002.

[27] Y.-J. Kwon, X. Ma, and H. Lee, "PARE: instruction set architecture for efficient code size reduction," *IEEE Electronics Letters*, Nov. 1999.

[28] D. Gajski, *et al.*, *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, 1992.

[29] D. Patterson and J. Hennessy, *Computer Organization and Design: The Hardware/Software Interface, 2nd ed.* Morgan Kaufmann Publishers, 1997.

[30] *lp_solve, version 3.2*, ftp://ftp.ics.ele.tue.nl/pub/lp_solve/.

Appendix

A Proof of NP-hardness

This section shows that the problem of selecting an optimal set of complex instructions is NP-hard under a generalizing assumption that the *cost* value of complex instructions can be any integer, by showing the knapsack problem (an NP-hard problem) reduces to the complex IS selection problem.

Problem) The knapsack problem can be formulated as following: given a set of items $S = \{1, \cdots, n\}$, where item i has size s_i and value v_i, and a knapsack capacity C, find the subset $S' \subset S$ which maximizes the value of $\sum_{i \in S'} v_i$ while satisfying $\sum_{i \in S'} s_i \leq C$, i.e., fitting in a knapsack of size C. From Section 6.2, the complex IS selection problem can be defined as determining the values of binary variable X_i's maximizing (3) while satisfying (4), (5), (7), and (10). Show that the knapsack problem reduces to a generalized version of the IS selection problem where the s_i in (4) can be any integer.

Proof) Let's think of a special case of the complex IS selection problem where (a) only one version of complex instruction is created from a sequence of basic instruction instances (no superset instructions) and (b) all complex instructions are created from non-overlapping sequences of basic instruction instances (therefore no multiple covers). Then (7) becomes $X_i = C_i$ by the condition (a) and (10) is always satisfied from the condition (b). Furthermore from the latter condition, since there is no multiply covered basic instruction instance, there is no need to set different values for A_{ij}'s $(j = 1, \cdots, m_i)$ in (5). Rather, it is best to set every A_{ij} $(j = 1, \cdots, m_i)$ to have the same value as C_i to maximize (3). Then what remains is a knapsack problem with the value of v_i defined as $v_i = \sum_{j=1}^{m_i} G_{ij}$. Thus the knapsack problem reduces to the generalized version of the complex IS selection problem. ■

While the above proof is based on the assumption that the *cost* value of complex instructions can be any integer, in the original formulation of the problem it is constrained as integers of the form $2^n (n \geq 0)$. Therefore the original problem may be found to have polynomial algorithms, yet we have not found any such algorithm and it seems highly unlikely as well, due to the additional constraints of (5), (7), and (10). We want to mention here, however, that we do not exclude the possibility of finding such an algorithm or the proof that the original problem actually is NP-hard.

algorithm select-c-instructions

1: Given CS^C : integer
2: Given $MoreGeneralForm : C \rightarrow$ Set(C)
3: Initialize $MaxCycleReduc : B \rightarrow$ integer $:= 0$
4: Initialize $CInstrInst : B \rightarrow$ integer $:= 0$
5: Initialize $UnselectedSet$: Set$(C) := \{C_i \,|\, \text{for } ^\forall i\}$
6: Initialize $SelectedList$: List$(C) := \phi$
7: Initialize CII : integer $:= 1$
8: **while** $UnselectedSet$ is not empty **do**
9: Take, from $UnselectedSet$, C_i that has the largest *Benefit/Cost* ratio among those whose *Cost* is not greater than CS^C; if it cannot find one, **break;**
10: Append C_i to $SelectedList$
11: $CS^C = CS^C - Cost(C_i)$
12: **for all** $C_j \in MoreGeneralForm(C_i)$ **do**
13: $Cost(C_j) = Cost(C_j) - Cost(C_i)$
14: **end for**
15: Initialize *AffectedSet* : Set$(C) := \{\,\}$
16: **for all** $BasicInstrSeq \in InstancesCovered(C_i)$ **do**
17: **if** $CycleReduc(C_i) > MaxCycleReduc(BasicInstrSeq)$ **then**
18: **for all** $B_k \in BasicInstrSeq$ **do**
19: $MaxCycleReduc(B_k) = CycleReduc(C_i)$
20: $CInstrInst(B_k) = CII$
21: Add, to *AffectedSet*, C_l's that cover B_k
22: **end for**
23: $CII = CII + 1$
24: **end if**
25: **end for**
26: **for all** $C_j \in AffectedSet$ **do**
27: *update-benefit*(C_j)
28: **end for**
29: **end while**
30: Sort *SelectedList* according to the *CycleReduc* value

end select-c-instructions

Figure 8. Heuristic algorithm for C-instruction selection.

algorithm update-benefit (C_i : C-instruction)

1: Initialize *eff_ben* : integer := 0
2: **for all** $BasicInstrSeqk \in InstancesCovered(C_i)$ **do**
3:     **if** $CycleReduc(C_i) > MaxCycleReduc(BasicInstrSeq_k)$ **then**
4:         $eff_ben = eff_ben + RepetitionCnt_k \cdot [CycleReduc(C_i) - MaxCycleReduc(BasicInstrSeq_k) \cdot Number\text{-}of\text{-}CI\text{-}Covers(BasicInstrSeq_k)]$
5:     **end if**
6: **end for**
7: $Benefit(C_i) = eff_ben$

end update-benefit

Figure 9. Subroutine *update-benefit*.

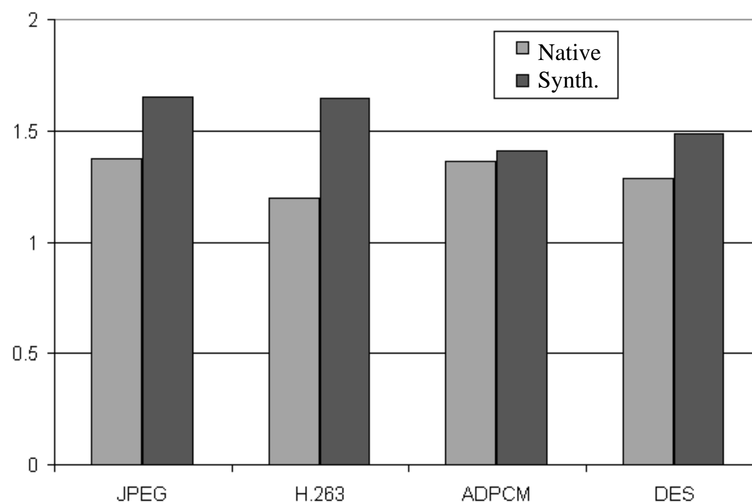

Figure 10. Performance of the native and the synthesized IS's, normalized to that of the basic IS (SH-3)

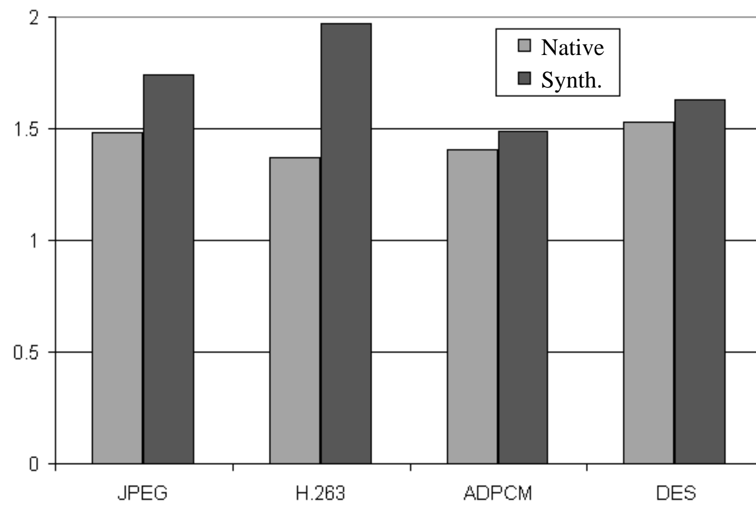

Figure 11. Performance of the native and the synthesized IS's, normalized to that of the basic IS (MIPS)