Efficient Instruction Encoding for Automatic Instruction Set Design of Configurable ASIPs

Jong-eun Lee
Center for Embedded Computer Systems
Univ. of California, Irvine
Irvine, CA 92697
jonglee@ieee.org

Kiyoung Choi
School of EECS
Seoul National University
Seoul, 151-742 South KOREA
kchoi@azalea.snu.ac.kr

Nikil Dutt
Center for Embedded Computer Systems
Univ. of California, Irvine
Irvine, CA 92697
dutt@cecs.uci.edu

Abstract

Application-specific instructions can significantly improve the performance, energy, and code size of configurable processors. A common approach used in the design of such instructions is to convert application-specific operation patterns into new complex instructions. However, processors with a fixed instruction bitwidth cannot accommodate all the potentially interesting operation patterns, due to the limited code space afforded by the fixed instruction bitwidth. We present a novel instruction set synthesis technique that employs an efficient instruction encoding method to achieve maximal performance improvement. We build a library of complex instructions with various encoding alternatives and select the best set of complex instructions 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 38% over the native instruction set for several realistic benchmark applications running on a typical embedded RISC processor.

1. Introduction

Configurable processors are application-specific synthesizable processors where the instruction set 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 traditional custom designed architectures [1]. With the commercialization of such configurable processors [2][3], and also the increased interest in platform-based SOCs that employ such configurable processors, the problem of instruction set (IS) optimization is receiving a lot of attention from both industry and academia [4][5].

In the design of such application-specific instructions, one of the common ways of improving performance is to make complex instructions from frequently occurring operation patterns [6][8][9], where a complex instruction is an instruction with more than one basic operation. The use of such complex instructions generally leads to better performance, smaller code size, and lower energy[1], since they replace sequences of simple instructions. However, in a processor with a fixed instruction bitwidth, not every operation pattern can be made into a new instruction due to the instruction bitwidth limitation.[2] This paper presents a novel instruction set synthesis technique that employs efficient instruction encoding to achieve maximal performance improvement. Our approach first builds a library of complex instructions with various encoding alternatives and then selects the best set of complex instructions while satisfying the instruction bitwidth constraint. We formulate the problem using integer linear programming (ILP). 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 synthesizes instruction sets that generate up to 38% performance improvement over the processor's native[3] 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 [6][8][9] 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 custom designed ones.

The rest of this paper is organized as follows. Section 2 introduces application-specific instruction set synthesis using motivating examples. Section 3 summarizes related work and Section 4 details the proposed IS synthesis technique with the problem formulation and heuristic algorithm. Section 5 shows the efficacy of our techniques through experiments on a typical embedded RISC processor running realistic applications, and Section 6 concludes the paper.

2. Motivation

We illustrate the potential for significant performance improvements using application-specific instructions on a typical embedded RISC processor (the Hitachi SH-3 [12]) 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

[1] Due to the reduced number of instruction fetches.

[2] We consider fixed instruction bitwidth because it is more common in contemporary embedded RISC processors.

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

0-7803-7607-2/02/$17.00 ©2002 IEEE

arguments and only the opcodes 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.

Another way to improve performance through IS specialization is to combine frequently occurring operation patterns into complex instructions. 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 [6][9], 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 IS synthesis technique detailed in Section 4.

3. Related Work

The synthesis of application-specific instruction set architectures (ISAs) have been approached in different ways: depending on which part of the ISA is decided first, IS-oriented and structure-oriented. In IS-oriented approaches [6][7][8], the IS is first optimized from the application's behavior, which is typically represented as dependency graphs. The hardware is later designed to implement the instruction set, manually or automatically. Among the approaches in this direction, PEAS-I [7] 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 [6]) 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 system-on-a-chip design.

Structure-oriented approaches [9][10] 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.

Another very closely related work is application-specific ISA customization for configurable processors. Zhao et al., in a case study [5] with a commercial configurable core, demonstrated 2 ~ 4 times performance improvement via configuration and application-specific instructions. They changed the data path width, which is readily supported by the core, and designed several custom instructions, which require the application program to be re-coded to accommodate the newly added instructions. Therefore, their work is limited to the application and the core they used. Our approach is significantly different, since it automates the design of application-specific instructions, tries to improve the processor by changing only the IS, and generates the instructions that can be easily supported by compilers without re-coding the program.

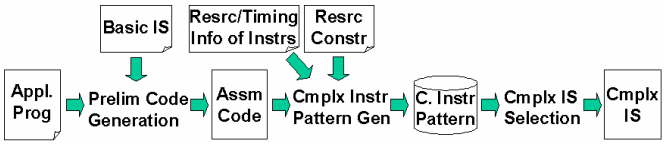

Fig. 1. Instruction set synthesis process.

4. IS Synthesis and Instruction Encoding

Fig. 1 presents the overall flow of our approach. Initially, the compiler takes the application program and generates preliminary assembly code, using only basic instructions (as defined below) and with optimizations such as instruction scheduling disabled. Then the IS synthesis process consists of two major steps: (1) complex instruction pattern library construction, and (2) selection of complex instructions.

A multitude of candidate complex instructions are created from the assembly code of an application. Complex instructions are put into a pattern library and annotated with such information as reference count number (how many times it is used during the execution of the application), number of bits needed for operands encoding, etc. Then the most profitable complex instructions are selected, that collectively satisfy the instruction bitwidth constraint and which would maximize the application performance.

To make it easier to create complex instructions from an application, we first define a basic IS: a *basic instruction* is restricted to have at most one operation to maximize the code space for a complex IS. A complex instruction can be viewed as a pattern of basic instructions. A target processor is then represented by a basic IS, which specifies all operations supported by the target processor, and structural information associated with each basic instruction, which is crucial to decide the usefulness of complex instructions. The structural information reveals functional and pipeline resources for each instruction with their cycle-level timing. From an assembly code of basic instructions we create complex instructions, each of which is essentially a condensed and generalized form of a basic instruction sequence. Complex instructions can be multi-cycle instructions and are created such that their latency is minimized under the resource constraints of the processor (e.g. # of register

read/write ports, # of functional units and buses at each pipeline stage, etc.). High-level synthesis (HLS) techniques such as resource-constrained list scheduling [11] can be used to find the number of cycles saved by using a complex instruction over a basic instruction sequence. If the saving is positive, we create a group of complex instructions, differing in the degree of generalization of operands, and put them into the complex instruction pattern library, to be used in the subsequent step of complex IS selection. We now describe each step in more detail.

4.1 Complex Instruction Pattern Generation

We assume the total instruction bitwidth is fixed but opcode and operand field widths are optimized for the applications. The complex instruction patterns generated here have two distinctive features as detailed below.

First, the opcode field widths are set to use the remaining bits after the widths of the operands are determined. This scheme is more flexible and efficient in terms of total instruction bitwidth usage than fixed-width opcodes for two reasons: (1) complex instructions with more operands (thus requiring more bitwidth) can be allowed if they are used often enough to justify the bitwidth usage, and (2) more complex instructions can be allowed that have fewer operands and hence requiring fewer operand bits. The only condition to be satisfied here 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}). It is obvious that if this condition is met, every instruction can be given an opcode, possibly with a different width.

Second, since there can be multiple choices of field widths for an operand type, we create *operand classes*, where each class defines a set of operand instances with a bitwidth specification. These operation classes are used to define complex instructions. Generally, in a custom instruction set, the operand field width of a certain operand type can vary among instructions to allow more compact encoding of the operands. For example, an immediate field may have only 4 bits in the ADD+LOAD type instructions while it may have 8 bits in an ADDI instruction. Register fields can also have a reduced size to allow more operands to be encoded at the cost of accessing only a subset of the registers in that instruction. An example can be found in the SH-3 microprocessor [12], where some complex load instructions have the R0 register as their destination, so that the destination register need not be specified at all. To support such variability in operand field widths, an operand is allowed to be generalized into multiple operand classes with different bitwidths.

Complex instructions are created for every sequence of N basic instruction instances as long as they take fewer number of execution cycles than their basic instructions versions, where N is a design parameter and more than one value of N can also be used at the same time. A group of complex instructions is created by substituting different operand classes for operands. Because complex instructions can be created from only sequentially appearing instruction instances, the number of created complex instructions increases only linearly in the code size of the application.

Fig. 2 illustrates the process of generating a complex instruction from a sequence of basic instruction instances. Fig. 2 (a) shows a conceptual view of the two sequences of basic instructions substituted by two complex instructions. Operand classes can be defined as in Fig. 2 (b), where "#bits" is the number of bits needed for encoding operands. Fig. 2 (c) shows a group of complex instructions created from a sequence of basic instructions, where "#bits" is defined as before, and *CR* stands for the benefit of a complex instruction in terms of the cycle count reduction.

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 complex instruction. Complex instructions with such a register operand class have their *CR* value (cycle count reduction by a single use of the complex instruction) reduced by the probability of necessitating an additional *move* instruction (in Fig. 2 (c), this probability is assumed to be 0.75).[4]

Complex instructions thus created are compared with those in the library and the library is updated either by including the new complex 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 complex instructions, from which a complex IS is defined.

(a) Basic Instructions

ADDI R1 R1 4
LOAD R1 (R1)
→ *ADDI+LOAD R1 (R1 + 4)*
MULT mac R1 R2
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]	$-8 \le$ IMM < 7	4
	[Imm_6bits]	$-32 \le$ IMM < 31	6
DISP	[Disp_normal]	$-2^7 \le$ DISP $< 2^7$	8
	[Disp_short]	$-2^3 \le$ DISP $< 2^3$	4

(c) Complex Instructions

Basic Instructions Sequence:
ADDI R1 R1 4
LOAD R1 (R1)

Complex Instructions Created:			#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

Fig. 2. Creating complex instructions.

4.2 Complex Instruction Set Selection

Every complex 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 complex 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 complex instruction can be estimated.

Complex instructions contribute to cycle count reduction at the cost of code space. Therefore the problem is to select a set of complex instructions that maximizes the cycle count reduction across the entire application program while respecting the code space requirement. Let $\{CI_i\}$ be the set of complex instructions created, W_i be an associated bitwidth needed for a complex instruction CI_i, and x_i be a binary variable representing whether CI_i is selected or not. Then the code space requirement can be represented as

$$\Sigma x_i \cdot 2^{Wi} \le Constr \quad (1)$$

$$Constr = 2^{IBW} - \Sigma 2^{Bi}, \quad (2)$$

where IBW is the instruction bitwidth (e.g. 16 or 32) and B_i is the number of operand bits needed for each basic instruction. Now the problem of

[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.

selecting the most useful complex instructions reduces to mapping $\{x_i\}$ to $\{0, 1\}$ satisfying Eq. (1) such that the total cycle count reduction from the selected CI_i's is maximized.

Table 1. Complex instructions and associated information

Cmplx Instr	# Bits	*CR*	Covered Basic Instruction Instances	*Benefit* (Total *CR*)	*Cost*
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 complex 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 complex instruction is defined as the sum of the *CR*s 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 complex instruction in terms of the code space taken.

The benefit and cost can change as complex 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 complex instructions, say *Sc*, 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 complex instruction, however, only one of them should have the decreased cost.

Multiple covers occur when more than one complex instruction covers the same basic instruction instance. In Table 1, a_3 is covered by all the complex instructions. Obviously only one complex instruction can actually be substituted for a_3 (and the neighboring basic instructions), which leads to the decrease in their collective benefit. Assuming a possible compiler optimization pass that applies, in a predefined order, substitutions of complex instructions for certain basic instruction patterns, multiple 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 *Sc* 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 *Sc* when CI_1 is already in *Sc* is 15 (= 45 – 30), where 30 is the benefit of CI_1.

A special case of this problem, where no superset instructions or multiple covers take place, can be shown to be a knapsack problem (which is NP-hard) if the *cost* value of complex instructions can be any positive integer while in the original formulation it should be 2^n ($n \geq 0$) [13].

4.3 ILP Problem Formulation

For the description of the problem formulation, we first introduce the following variables. Let the complex instruction library be given as $\{CI_i \mid i = 1, 2, \ldots, n\}$ and each instruction has bitwidth information W_i that is the number of bits needed for operands encoding. With each complex instruction (CI_i) is associated a set of basic instruction instances sequences $\{BII_{ij} \mid j=1, 2, \ldots, m_i\}$, 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,

$$G_{ij} = Repetition_{ij} \cdot Cycle\_Reduc_i \quad (3)$$

where $Repetition_{ij}$ is the repetition count of the block, $Cycle\_Reduc_i$ 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.

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

Then the complex instruction set selection problem can be thought of as selecting BII_{ij}'s, for which corresponding CI_i's can be substituted. Here the actual compiler that will use those complex 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,…,n; j=1,…,m_i)* : 1 if BII_{ij} is selected,
- C_i *(i=1,…,n)* : 1 if CI_i is selected, that is, if any of A_{ij} *(j=1,…,m_i)* is 1,
- X_i *(i=1,…,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

$$\max : \sum_{i=1}^{n} \sum_{j=1}^{m_i} G_{ij} A_{ij} \quad (4)$$

$$\sum_{i=1}^{n} 2^{W_i} X_i \leq Constr \quad (5)$$

The relationship between the binary variables can be represented as

$$C_i = A_{i1} \vee A_{i2} \vee \ldots \vee A_{im_i}, \quad \text{for } {}^{\forall}i \quad (6)$$

and by letting CI_i's superset instructions be $\{CI_{S1}, CI_{S2}, \ldots, CI_{Sk}\}$,

$$X_i = C_i \wedge \overline{C}_{S1} \wedge \overline{C}_{S2} \wedge \ldots \wedge \overline{C}_{Sk}, \quad \text{for } {}^{\forall}i. \quad (7)$$
$$\Leftrightarrow \overline{X}_i = \overline{C}_i \vee C_{S1} \vee C_{S2} \vee \ldots \vee C_{Sk}$$

Equations (6) and (7) can be linearized using the following identities.

$$\overline{X} \Leftrightarrow 1 - X$$
$$C = A_1 \vee A_2 \Leftrightarrow C \geq A_1, C \geq A_2, C \leq A_1 + A_2 \quad (8)$$

Finally, the constraint due to 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 $BII_{I1\,J1}$, $BII_{I2\,J2}$, …, $BII_{Ix\,Jx}$, then

$$A_{I1,J1} + A_{I2,J2} + \ldots + A_{IxJx} \leq 1, \quad (9)$$

which holds for every basic instruction instance.

Equations (4) ~ (9) define the ILP formulation for the complex 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.

4.4 Heuristic Algorithm

Fig. 3 shows our heuristic algorithm proposed for the problem of complex IS selection and ordering based on the above observations. The algorithm works by repeatedly selecting the most promising complex instruction (i.e., the one with the largest benefit per cost ratio). The ordering of the selected complex instructions is decided by their *CR* values: the greater the *CR*, the higher the priority. Among those with the same *CR* value, priority follows the order in which they were selected.

Superset Instructions: Every complex instruction has a set of pointers to more general complex instructions (*More_General_Form* in Fig. 3). A complex instruction is more general when all operands are encoded with more general or equal operand classes. This set can be built up as new complex instructions are created and added to the library. When a complex instruction is selected, its cost is subtracted from the cost of each of the more generalized complex instructions.

Multiple Covers: If a basic instruction instance is covered by more than one complex instruction and those complex 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 complex instruction under the assumption that a compiler substitutes complex instructions for matching patterns in a predefined order, the algorithm introduces two new integer variables for every basic instruction instance. *Max_Cycle_Reduc* of an instruction instance is the

maximum of *CR*'s (*Cycle_Reduc*) of the complex instructions that cover the instruction instance and have been selected so far. *Complex_Instr_Inst* is the instance ID of that complex instruction with the *Max_Cycle_Reduc*. Because the selected complex instructions are ordered by the *Cycle_Reduc* value, a complex instruction (C_j) can only be used when all the instruction instances in the matched basic instruction sequence have *Max_Cycle_Reduc* value of less than *Cycle_Reduc* of C_j. Therefore as a complex instruction is selected, other complex instructions that share some of the covered basic instruction instances (*Instances_Covered*) are affected in their benefit. The effective benefit of choosing one (C_i) is lowered because: (1) for instances that already have greater *Max_Cycle_Reduc* value, C_i will not be used (because it is less effective), and (2) for instances that have less *Max_Cycle_Reduc* value, C_i will be used replacing other complex instructions with lower *Cycle_Reduc*, 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 number of complex instructions in the library and M is the average number of basic instruction sequences covered by a complex 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.

```
Notation B : a basic instruction instance
Notation C : a complex instruction
Initialize Max_Cycle_Reduc : B → integer := 0
Initialize Complex_Instr_Inst : B → integer := 0
Initialize Unselected_Set : Set(C) := {Ci | for all i}
Initialize Selected_List : List(C) := φ
Given Constr : integer
Given More_General_Form : C → Set(C)
Initialize CI_Inst : integer := 1
While ( Unselected_Set is not empty ) {
  Take, from Unselected_Set, Ci that has the largest benefit/cost
  ratio among those whose Cost is not greater than Constr; if not
  found, break;
  Append Ci to Selected_List
  Constr = Constr – Cost(Ci)
  Foreach Cj ∈ More_General_Form(Ci)
    Cost(Cj) = Cost(Cj) – Cost(Ci)
  Initialize Affected_Set : Set(C) := { }
  Foreach Basic_Instr_Seq ∈ Instances_Covered(Ci)
    If ( Cycle_Reduc(Ci) > Max_Cycle_Reduc(Basic_Instr_Seq) ) {
      Foreach Bk ∈ Basic_Instr_Seq {
        Max_Cycle_Reduc(Bk) = Cycle_Reduc(Ci)
        Complex_Instr_Inst(Bk) = CI_Inst
        Add, to Affected_Set, Cl's that cover Bk
      }
      CI_Inst ++
    }
  Foreach Cj ∈ Affected_Set
    Update_Benefit(Cj)
}
Sort Selected_List according to the Cycle_Reduc value
```

Fig. 3. Complex instruction selection and ordering algorithm.

```
Initialize eff_ben : integer := 0
Foreach Basic_Instr_Seqk ∈ Instances_Covered(Ci)
    If ( Cycle_Reduc(Ci) > Max_Cycle_Reduc(Basic_Instr_Seqk) )
        eff_ben += Repetition_Cntk * ( Cycle_Reduc(Ci)
            – Max_Cycle_Reduc(Basic_Instr_Seqk) *
            Number_of_CI_Covers(Basic_Instr_Seqk) )
Benefit(Ci) = eff_ben
```

Fig. 4. Procedure *Update_Benefit(Ci)*.

5. Experiments

For our experiments, we used the SH-3 processor [12] as the representative architecture for the basic IS and structural information such as pipeline configuration. We ran a number of realistic benchmark applications covering multimedia (e.g., H.263 decoder[6], JPEG), control-intensive (e.g., ADPCM[7]) and cryptography (e.g., DES) domains. These applications were processed by the EXPRESS retargetable compiler [14] (targeting the basic IS) to generate preliminary assembly code, which was used for the experiments.

We chose the SH-3 processor because it is representative of popular contemporary RISC cores that also contain DSP-like features such as auto-increment load and MAC (multiply-accumulation). The SH-3 has an *IBW* (instruction bitwidth) of only 16 bits, so that it becomes very important to find a better IS for a more effective utilization of the hardware. From the native IS of the SH-3 architecture, a basic IS was defined with code space of 15442, 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 other versions of the processor family, the code space that is available for a complex IS is about 2^14, which is used as the value of *Constr* in our experiments. Another parameter N (the number of basic instructions that are considered together for complex instruction creation) was set to be 2 to 4 (i.e., up to 4 consecutive basic instructions are considered). In defining operand classes, we used some statistics such as frequent values of immediate/displacement or average register pressure, etc. One definition of operand classes was used for all benchmark applications except for the DES application.

5.1 Comparison of ILP and Heuristic Algorithm

For the comparison of the two proposed 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. 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* [15] was used on a Pentium 866 MHz Linux PC. Table 2 shows the results.

Table 2. Comparison of ILP and heuristic algorithm

Bench-mark (Func)	#inst (code size)	#patt. in the library	#cycle (basic IS)	Exp. total #cycle reduction		run-time	#C.I.'s selected	
				ILP	HA	ILP	ILP	HA
EncAC	157	135	586062	229674	229674	129	19	19
Quant	67	106	302782	139693	139693	11	14	15
Bound	51	78	141502	47533	47533	< 1	10	11
Zigzag	47	81	174801	70033	70033	3	11	12
Decode	217	193	50932	16706	16704	89	25	24
AdpEnc	195	178	932700	239788	239779	104	23	25

In Table 2, the 2nd to 4th columns show the code size (in terms of the number of basic instructions), the number of (unique) complex instructions in the library[8], and the cycle count with the basic IS, respectively. The fifth and sixth columns show the number of cycles reduced by the selected set of complex instructions. The seventh column shows the ILP solver run-time in seconds, and the eighth and ninth columns show the number of selected complex instructions. In all cases, the heuristic algorithm could find the optimal or a near-optimal solution with a very short run-time (less than 1 second), while the ILP problems sometimes cannot be solved due to rounding errors. This demonstrates the efficacy of our heuristic with respect to an ILP formulation.

[6] From Telenor R&D distribution (ver. 2.0), which is originally based on an implementation by MPEG Software Simulation Group, 1994.
[7] From MediaBench, http://www.cs.ucla.edu/~leec/mediabench/ .
[8] Complex instructions that require more bits than *IBW* are not added to the library.

5.2 Comparison of Basic, Synthesized, Native IS's

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 complex IS was generated using our heuristic algorithm. The generated complex IS was used in a back-end optimization pass, which, in a given order, substitutes complex instructions for basic instruction sequences.

Table 3 compares the results for three different IS's: the basic IS, the synthesized IS (= basic + complex), and the native IS. 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 4th to 6th columns show the cycle counts (in millions of cycles) for the three IS's. The performance improvements of the synthesized IS over the basic and the native IS are shown in the 7th and 8th columns, where the performance is defined as the inverse of the number of cycles. The last column shows the number of complex instructions synthesized for each benchmark program.

Table 3. Comparison of synthesized IS with the basic & native IS's

Bench-mark Prog.	#inst (code size)	#patt. in the library	#cycle basic IS	#cycle native IS	#cycle synth. IS	%impr over basic	%impr over native	#C.I.
JPEG	1192	557	2.609	1.855	1.557	**67.6**	**19.1**	37
H263	1475	615	3.090	2.550	1.844	**67.6**	**38.3**	36
ADPCM	437	235	1.674	1.243	1.179	**42.0**	**5.5**	39
DES	1135	517	0.611	0.466	0.399	**53.3**	**16.9**	40

From the table, it is evident that complex instructions improve the performance significantly (40 ~ 67 %) 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 ~ 40 %), and positive (5 ~ 15 %) 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 complex instructions only within basic blocks, it often cannot find performance-improving complex instructions for those blocks in control-intensive applications. And 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 newly generated complex instructions 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 found in only 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.

6. Conclusion

We presented a novel IS synthesis technique 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 38% 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.

Hardware implementation of the synthesized instruction set demands the modification of control path, most importantly instruction decoder. Modifying instruction decoder to support complex instructions may affect the critical path and further the clock speed of the processor, potentially leading to degraded performance improvement. This is particularly true if we compare the synthesized instruction set with the basic one. Native instruction sets, however, often have their own "complex" instructions, sometimes with many odd ones. The instruction set synthesis replaces the native complex instructions with application-specific ones. Therefore, complexity increase of instruction decoder (and also the cycle time increase) by using synthesized instruction sets rather than the native ones are not necessarily substantial. Thorough investigation is needed, though, to quantify the effect of the synthesized instructions on the cycle time and the final performance.

7. Acknowledgements

This research was supported in part by grants from Hitachi Inc., Motorola Corp. and NSF (CCR-0203813). We also thank members of the EXPRESS compiler team in the ACES laboratory 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] A. Cataldo, "Compiler that converts C-code to processor gates advances," EE Times, Oct. 23, 2001, http://www.eet.com/story/OEG20011023S0028 .

[5] Y. Zhao *et al*., "Matching architecture to application via configurable processors," In Proc. ICCD 2001.

[6] M. Arnold and H. Corporaal, "Designing domain-specific processors," In Proc. Codesign Symposium 2001.

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

[8] J. Van Praet *et al*., "Instruction set definition and instruction selection for ASIPs," In Proc. HLS Symposium 1994.

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

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

[11] D. Gajski, *High-Level Synthesis*, Kluwer Academic Publishers, 1992.

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

[13] J. Lee *et al*., "Automatic instruction set design through efficient instruction encoding for application-specific processors," Tech. Report, #02-23, CECS, UC Irvine.

[14] *EXPRESS Retargetable Compiler*, Univ. of California, Irvine, Project website http://www.cecs.uci.edu/~aces/ .

[15] *lp_solve*, Version 3.2, available at ftp://ftp.ics.ele.tue.nl/pub/lp_solve/ .