Area-Efficient Instruction Set Extension Exploration with Hardware Design Space Exploration*

I-WEI WU, CHUNG-PING CHUNG AND JEAN JYH-JIUN SHANN
Department of Computer Science
National Chiao Tung University
Hsinchu, 300 Taiwan

Instruction set extension (ISE) is an effective approach to improve the processor performance without tremendous modification in its core architecture. To execute ISE(s), a processor core must be augmented with a new functional unit, called application specific functional unit (ASFU), which consists of multiple hardware implementation options of ISEs (ISE_HW). Obviously, since ISE_HW increases the production cost of a processor core, minimizing the area size of ISE_HW becomes important for ISE exploration. On the other hand, because of different requirements in space and speed, ISE_HW usually has multiple hardware implementation options. Under pipeline-stage timing constraint, some of these options may have the same performance improvement but entail different hardware costs. According to this phenomenon, the area size of ISE_HW can be reduced by performing hardware design space exploration of ISE_HW. Therefore, in this paper, we propose an ISE exploration algorithm that explores not only ISE but also the hardware design space of ISE_HW. Compared with the previous research, our approach resulted in significant improvement in area efficiency and the execution performance.

Keywords: instruction set extension (ISE), customizable processor, application-specific instruction-set processor (ASIP), design space exploration, area efficient

1. INTRODUCTION

Recently, next-generation digital entertainment and mobile communication devices are driving the need for high-performance processing solutions. In order to satisfy this demand, current embedded processor architectures, such as Tensilica Xtensa, ARC ARC tangent, MIPS CorExtend and Nios II, provide designer with the possibility to define custommized instruction, or called instruction set extension (ISE). An ISE could be considered a frequently executed operation pattern in the target application(s). By executing the operation pattern(s) on a specific functional unit, called application specific functional unit (ASFU), the execution performance of the target application(s) can be improved.

Generating ISE mainly consists of two steps: ISE exploration and ISE selection. ISE exploration is to find out legal operation patterns in the given application(s), while ISE selection aims to choose ISEs among all explored ISEs. Note that most attention is paid on ISE exploration in this paper. In general, an ISE can be explored by (1) using a given operation pattern set [1, 2] or (2) creating a new operation pattern set [3-6]. The first approach matches all operation patterns of a given operation pattern set with each basic block within the application(s). After matching all patterns, the legal matched pattern(s) with maximal size is considered as ISE(s). In the second approach, each operation pattern is

Received December 14, 2009; revised April 6, 2010; accepted June 14, 2010.
Communicated by Pen-Chung Yew.
* This paper was partially supported by the National Science Council of Taiwan, R.O.C. under contract No. NSC 98-2221-E-009-158-MY3.

grown from an operation under ISE exploration constraints. Moreover, the algorithms of the second approach can further be divided into two categories. The first category is to enumerate all or partial operation patterns and then to select several proper and legal ones as ISEs [3-5]. In this category, the exact algorithm proposed by Pozzi *et al*. mapped ISE search space, such as a basic block, to a binary tree, and then discarded some portion of the tree that violates ISE exploration constraints [5]. Nevertheless, this algorithm is highly computation-intensive, so it does not process for a larger search space. To overcome this problem, Pozzi *et al*. derived another algorithm from it by applying genetic algorithm to reduce the execution time. The second category is to divide all operations within a basic block into two parts, including software (meaning an operation will be executed on original ALU) and hardware (meaning an operation will be executed on the ASFU) [6]. A legal (*i.e.*, conforms to all ISE exploration constraints) set of reachable operations in the hardware part is regarded as an ISE. More detailed discussion of ISE exploration algorithm could be referred to [7]. The algorithm proposed in this paper belongs to the second category.

Deploying ISE on a processor core can improve the execution performance but will also incur extra hardware cost/area size. Consequently, how to save the area size of the hardware implementation of ISE (ISE_HW) without compromising performance improvement is a crucial consideration during exploring ISE. In general, ISE_HW can be realized at different cost and speed requirements. Among these different hardware implementation options, several of them have the same execution cycle(s) but demand different hardware costs under the pipeline-stage timing constraint. This calls for improving the area efficiency by exploring the hardware design space of ISE_HW.

Although exploring the hardware design space of ISE_HW is an important consideration for ISE exploration, most of the research has not taken this consideration into account, except for the work proposed by Sun *et al*. [8]. In Sun's approach, performing hardware design space exploration of ISE_HW is to make the execution time of ISE fit in one (processor) cycle at the original processor core's clock period. In other words, hardware design space exploration is only performed when the execution time of ISE is more than one (processor) cycle. According to our simulation, most ISE's execution time is smaller than one (processor) cycle. This implies that Sun's approach would fail to improve the area efficiency of ISE_HW in many cases.

In order to reduce the hardware cost of ISE_HW without compromising the performance improvement, in this paper we propose an algorithm that integrates ISE exploration and hardware design space exploration of ISE_HW. To perform these two tasks concurrently, we design a tree-like data structure, called solution tree, to join the solution spaces of both ISE exploration and hardware design space exploration of ISE_HW. In addition, to improve the exploration result further, we also make use of the ant colony optimization (ACO) algorithm [9, 10]. Our results revealed that the proposed algorithm can significantly increase the area efficiency without losing performance improvement.

The rest of this manuscript is organized as follows. The background and problem dealt with in this paper are detailed in section 2. Section 3 then presents the proposed approach. Section 4 shows the simulation results and discussion. Conclusions are finally drawn in section 5.

2. BACKGROUND AND PROBLEM STATEMENT

This section first gives an overview of ISE design flow and then describes the problem dealt with in this paper.

2.1 ISE Design Flow

Fig. 1 shows the ISE design flow. Initially, after application profiling, basic blocks with longer execution time or higher frequency of execution are selected as the input of ISE exploration. ISE exploration finds out legal operation patterns as ISEs, which must conform to ISE exploration constraints. In ISE merging, several isomorphic ISEs explored in different parts of the application are merged together. ISE *B* is merged into ISE *A*, if ISE *B* is a subgraph of or identical to ISE *A*. After ISE merging, ISE selection sorts all explored ISEs in a predefined order, such as the execution performance improvement, and then chooses ISE to achieve the highest performance improvement under predefined constraints. To achieve higher hardware utilization, hardware sharing is also performed at this stage. Hardware sharing is the assignment of a hardware resource to multiple operations belonging to different ISEs. In this paper, we focus on ISE exploration only.

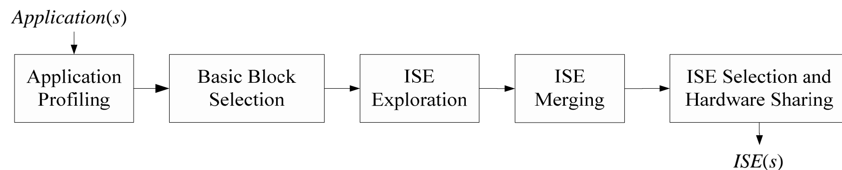


Fig. 1. ISE design flow.

The constraints used in ISE explorations (1-5) and ISE selections (2 and 5) are described as follows,

1. Pipeline-stage timing: Under this constraint, the execution time of ASFU must fit in an integral number of cycles at the original clock period of the processor core.
2. ISA format: ISA format implies two constraints, the number of input/output operands employed by an ISE and the number of ISEs being selected.
3. Register file: The number of input/output operands adopted by an ISE cannot exceed the number of the register file read/write ports.
4. Convex property: The convex property is that ISE's output cannot connect to its input via other operations not grouped in ISE to ensure that a feasible scheduling may exist.
5. Silicon area: This constraint restricts how much silicon area can be used for a single and/or all ISEs.

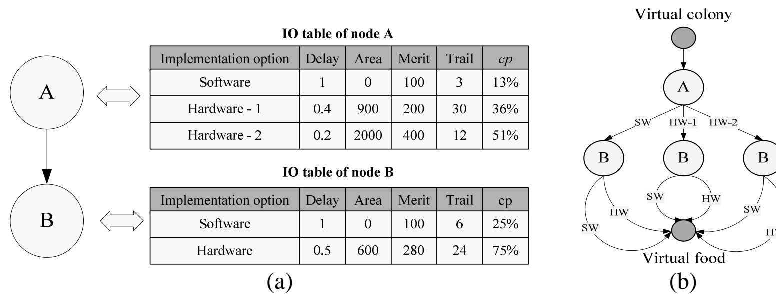
2.2 Problem Statement

The problem that we deal with in this paper is (1) ISE exploration, and (2) hardware design space exploration of ISE_HW. ISE exploration aims to find out legal operation patterns (*i.e.*, ISEs) from selected basic blocks, whereas hardware design space explora-

tion is to select a proper hardware implementation option for an ISE_HW.

Before exploring ISE, each basic block is transformed to a data flow graph (DFG) firstly. A DFG is represented by a directed acyclic graph G, in which every vertex represents an operation and each edge denotes the dependence between two operations. An ISE is considered a subgraph within the DFG.

To represent all implementation options of an operation, we append a new data structure, called implementation option (IO) table, on each operation in the DFG. Each entry in the IO table comprises six fields, namely *Implementation option*, *Delay*, *Area*, *Merit*, *Trail*, and *cp*. The name, execution time, and the area size of each implementation option are shown in *Implementation option*, *Delay* and *Area* fields, respectively. The *merit* value is the benefit of an implementation option being chosen, and the *trail* is analogous to the pheromone in the ACO algorithm and represents the number of times the implementation option being chosen in previous iterations. The detailed description of these two values is given in section 3. *cp* is abbreviated from **c**hosen **p**robability and represents the probability of each implementation option being chosen at the current iteration. A new graph G^+ is generated after appending the IO table to G. Fig. 2 (a) shows an example of G^+ which consists of two operations, A and B.


IO table of node A

Implementation option	Delay	Area	Merit	Trail	*cp*
Software	1	0	100	3	13%
Hardware - 1	0.4	900	200	30	36%
Hardware - 2	0.2	2000	400	12	51%

IO table of node B

Implementation option	Delay	Area	Merit	Trail	cp
Software	1	0	100	6	25%
Hardware	0.5	600	280	24	75%

Fig. 2. (a) An example of G^+; (b) An example of solution tree.

The proposed ISE exploration may be formulated as follows: Given $G^+(V, E, IO_T)$, explore subgraph set $S = \{s_i \subseteq G^+ \mid i = 0 \sim n-1\}$ and IO choice set $IO_Choice = \{io_c_i \mid i = 0 \sim |V| - 1\}$ such that $\sum_{i=0}^{n-1} cycle_reduction(s_i)$ is maximal and $\sum_{i=0}^{|V|-1} hardware_cost(io_c_i)$ is minimal under constraints $IN(s_i) \leq N_{\text{in}}$, $OUT(s_i) \leq N_{\text{out}}$, s_i is convex, and memory and flow control operations $\notin s_i$.

In $G^+(V, E, IO_T)$, V denotes a set of operations in the basic block, E represents a set of directed edges that each means the execution dependency between two operations, and IO_T is a set of IO (implementation option) tables that each represents the IO table for an operation in G. Subgraph set S is a group of ISEs explored in DFG G, each subgraph s_i represents an ISE, and cycle_reduction(s_i) means the number of reduced cycles of subgraph s_i. IO choice set, *IO_Choice*, represents the choice of implementation options of all operations in V, io_c_i is an integral number indicating which implementation option is chosen by operation i; $hardware_cost(io_c_i)$ represents the hardware cost of the implementation option io_c_i. $IN(s_i)$ ($OUT(s_i)$) is the number of input (output) operands used (generated) by a subgraph s_i. The user-defined values N_{in} and N_{out} denote the number of register file read and write ports, respectively. For a feasible instruction schedul-

ing, an ISE must comply with the convex property. Moreover, in this work, memory and flow control operations are forbidden from being grouped into an ISE.

In order to simultaneously explore an ISE and its corresponding hardware design space in a basic block, we propose a solution tree to combine both solution spaces together. In the solution tree, the operation can be treated as a node in the tree, and different implementation options are different branches (links) between nodes in the tree. Fig. 2 (b) depicts a solution tree of the example in Fig. 2 (a), in which *SW* and *HW* denote the software and hardware implementation options, respectively. Each path from the virtual colony to the virtual food denotes a possible solution of the ISE exploration problem dealt in this work. Noticeably, the virtual colony and the virtual food are two nodes which are appended additionally and are used to analogize the ant colony and the food location in ACO algorithm.

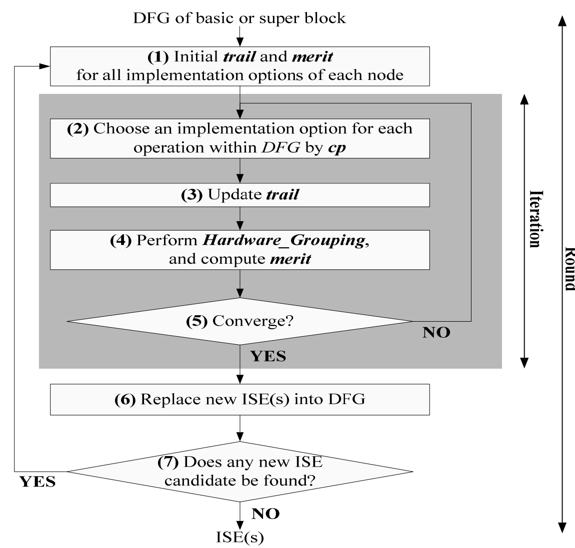


Fig. 3. The flow chart of ISE exploration algorithm.

3. ISE EXPLORATION ALGORITHM

Fig. 3 shows the flow chart of the proposed algorithm. Initially, in step 1, the algorithm sets up initial values for the trail and merit values of each implementation option of all operations. Notably, the hardware implementation options has higher initial merit value than software one such that the algorithm would preferentially choose the hardware implementation option at the start of execution to achieve higher execution performance improvement. In step 2, the algorithm selects one implementation option for each operation based on *cp* (**c**hosen **p**robability). *cp* is calculated by the trail and the merit values, as revealed in Eq. (1). The trail value represents the number of times that an implementation option has been selected in previous iterations. The trail value of implementation option y of operation x is denoted by $trail_{x,y}$, and $trail_{x,0}$ is designated as the trail value of the software implementation option. The merit value is defined as the bene-

fit of an implementation option being selected. The merit value of implementation option y of operation x is represented by $merit_{x,y}$, and $merit_{x,0}$ is designated as the merit value of the software implementation option. cp of implementation option y of operation x, $cp_{x,y}$, is defined as follows,

$$cp_{x,y} = \frac{\alpha \times trail_{x,y} + (1-\alpha) \times merit_{x,y}}{\sum_{i=0}^{io_x} \alpha \times trail_{x,i} + (1-\alpha) \times merit_{x,i}}, \tag{1}$$

where io_x is the number of the implementation options of operation x, α is used to determine the relative influence of trail and merit values, and

$$\sum_{y=0}^{io_x} cp_{x,y}. \tag{2}$$

In step 3, the algorithm updates the trail value using Eq. (3). Updating the trail value is similar to deposit and evaporate the pheromone on path from colony to food. The trail value of the implementation option chosen in step 2 would be increased by ρ, a positive constant value, while those of other implementation options would be decreased.

$$trail_{x,y} = \begin{cases} trail_{x,y} + \rho \\ trail_{x,y} - \rho \end{cases}, \text{ if } \begin{array}{l} \text{implementation option } y \text{ is selected} \\ \text{implementation option } y \text{ is not selected} \end{array} \tag{3}$$

In step 4, the algorithm computes the merit value of each implementation option for all operations. The algorithm first checks each operation to determine whether the operation has a hardware implementation option or not. If yes, then the algorithm executes the *Hardware_Grouping* function, which packs an operation and its reachable operations, whose implementation option are selected as hardware in current iteration, to form a virtual ISE. Besides, *Hardware_Grouping* function also checks the legality and calculates the execution time as well as the hardware cost of this virtual ISE. Based on the result of *Hardware_Grouping* function, the algorithm then computes the merit value using the merit function. The details of *Hardware_Grouping* function and merit function are described in sections 3.1 and 3.2, respectively.

Then, the algorithm replaces the new ISE(s) into DFG and checks the end condition. If the convergence condition is not fulfilled, then the algorithm returns to step 2 for the next iteration. When the convergence condition is achieved, the algorithm determines whether any new ISE is found in this round. If yes, the algorithm starts to perform next round; otherwise, the algorithm would terminate.

The convergence condition is either of the following two cases: (1) for all operations in DFG, cp of one of the implementation options exceed P_END (is set to 99% in this paper); or (2) after executing P_TIMES (is set to 100 in this paper) number of iterations, the execution performance improvement is still the same. A larger P_END or P_TIMES has a higher opportunity to obtain a better result, but typically takes a longer time to converge. In order to make all cases converge within a reasonable time, we have carefully tuned all parameters used in the algorithm. Accordingly, there is no case that cannot converge in our simulation. After achieving convergence, an implementation option with the

highest *cp* is taken. Note that each operation may have only one taken implementation option. An ISE is a set of connected/reachable operations whose taken implementation options are all hardware.

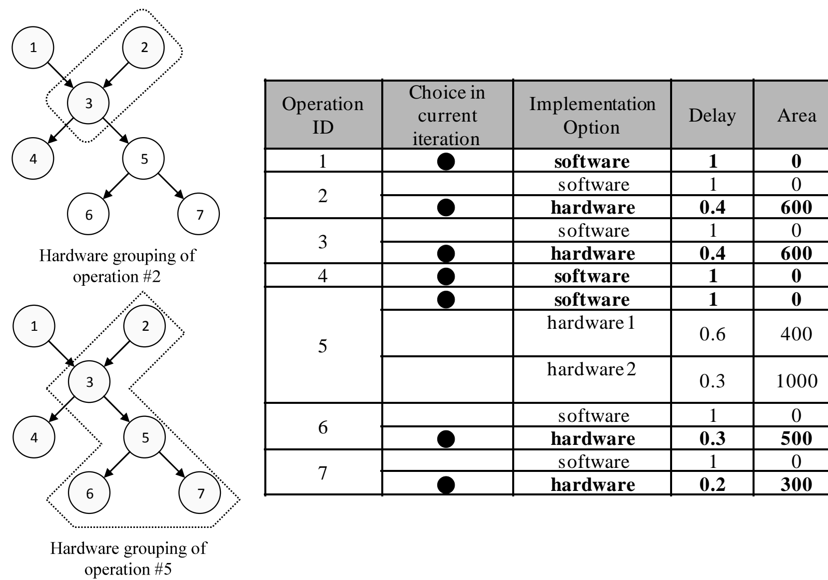
3.1 Hardware_Grouping

Hardware_Grouping aims to generate all necessary information for the merit function. The works of *Hardware_Grouping* consist of (1) grouping an operation and its reachable operations which have selected the hardware implementation option in current iteration as a virtual ISE; and (2) checking the legality and calculating the number of reduced cycles as well as the area cost for this virtual ISE. *Hardware_Grouping* is always performed if an operation has hardware implementation option, since an operation has opportunity to be packed into an ISE as long as it has hardware implementation option during ISE exploration. The virtual ISE of implementation option y of operation x is denoted as $vS_{x,y}$. For operation x using implementation option y, the execution cycle reduction and the area size are denoted by $reduced_cycle_{x,y}$ and $area_{x,y}$, respectively. $reduced_cycle_{x,y}$ is derived as follows,

$$reduced_cycle_{x,y} = \sum_{i \in vS_{x,y}} \text{execution cycle of } IO_{i,0} - \text{ISE_cycle}(vS_{x,y}) \tag{4}$$

where i is a operation within $vS_{x,y}$ and $IO_{i,0}$ represents the software implementation option of operation i. ISE_cycle($vS_{x,y}$) is a function to calculate the execution cycle of $vS_{x,y}$ by summing the delay of hardware implementation option of operations lay on the critical path of $vS_{x,y}$.

Fig. 4 is an example for illustrating the *Hardware_Grouping* function. For each op-



Operation ID	Choice in current iteration	Implementation Option	Delay	Area
1	●	**software**	**1**	**0**
2		software	1	0
	●	**hardware**	**0.4**	**600**
3		software	1	0
	●	**hardware**	**0.4**	**600**
4	●	**software**	**1**	**0**
5	●	**software**	**1**	**0**
		hardware 1	0.6	400
		hardware 2	0.3	1000
6		software	1	0
	●	**hardware**	**0.3**	**500**
7		software	1	0
	●	**hardware**	**0.2**	**300**

Fig. 4. Examples of Hardware_Grouping.

eration, the delay and area of each implementation option are listed and the chosen implementation option in the current iteration is specified in the table. Operations grouped by a dotted line in a DFG are treated as a virtual ISE. For operation #2, *Hardware_Grouping* groups operation #2 and #3 as a virtual ISE, as shown in the top left of Fig. 4. Because operation #2 has only one hardware implementation option, only one virtual ISE is generated, *i.e.*, $vS_{2,1}$. The evaluation results of $vS_{2,1}$ are $reduced_cycle_{2,1}$ = 1 (cycles) and $area_{2,1}$ = 1200 (μm^2). The bottom left of Fig. 4 is another example, in which *Hardware_Grouping* groups operation #5 and operations #2, #3, #6 and #7 as a virtual ISE. Since operation #5 has two hardware implementation options, two virtual ISEs, $vS_{5,1}$ and $vS_{5,2}$, are produced. Their evaluation results are $reduced_cycle_{5,1}$ = 3 (cycles) and $area_{5,1}$ = 2400 (μm^2) as well as $reduced_cycle_{5,2}$ = 3 and $area_{5,2}$ = 3000.

3.2 Merit Function

The merit function computes the benefits of different implementation options of each operation using the results of *Hardware_Grouping*. Briefly, the merit function consists of three parts, performance improvement checking (Part 1), legality checking (Part 2), and area cost and performance improvement checking (Part 3). Fig. 5 shows the pseudo code of the merit function. We assume that the merit of software implementation option, $merit_{x,0}$, is set to a constant as a baseline.

```
If (reduced_cycle_{x,y} == 0)
        merit_{x,y} = merit_{x,y} × β_{no_p};                                   Part 1
----------------------------------------------------------------------------------
Else If (vS_{x,y} violates in/out constraint || vS_{x,y} violates convex constraint)
        If (vS_{x,y} violates in/out constraint)
                merit_{x,y} = merit_{x,y} × β_{io};
        If (vS_{x,y} violates convex constraint)
                merit_{x,y} = merit_{x,y} × β_{convex};                         Part 2
----------------------------------------------------------------------------------
    Else
        merit_{x,y} = (reduced_cycle_{x,MAX} + 1) × merit_{x,0};
        If (reduced_cycle_{x,y} = reduced_cycle_{x,MAX})
            merit_{x,y} = merit_{x,y} × area_{x,MAX} / area_{x,y}
        Else
            merit_{x,y} = merit_{x,y} × area_{x,MAX} / (reduced_cycle_{MAX,y} + 1 − reduced_cycle_{x,y})   Part 3
```

Fig. 5. Pseudo code of merit function.

Initially, in Part 1, the algorithm checks whether $reduced_cycle_{x,y}$ is equal to zero or not. If it is equal to zero, then choosing hardware implementation option y for operation x could not contribute any performance improvement. Therefore, the algorithm multiplies $merit_{x,y}$ ($y \geq 1$) by a constant β_{no_p} ($0 < \beta_{no_p} < 1$) to lower the chance of hardware implementation option y being chosen in the following iterations. The calculation of the merit function is then terminated. If $reduced_cycle_{x,y}$ is larger than 0, then go to Part 2.

Part 2 verifies whether $vS_{x,y}$ violates register file and/or convex constraints. If yes,

i.e., $vS_{x,y}$ is illegal, then $merit_{x,y}$ is multiplied by constant β_{io} and/or β_{convex} ($0 < \beta_{io} < 1$ and $0 < \beta_{convex} < 1$) to reduce the opportunity for selecting hardware implementation option y. To avoid the optimal solution being excluded, when a constraint is violated, the merit values is penalized only by decreasing a constant rather than refusing an operation to be grouped into any ISE in the following iterations. If all constraints are conformed, then enter Part 3; otherwise, the calculation of the merit value is then terminated.

The basic concept of Part 3 is: (1) if $vS_{x,y}$ could improve the performance, then the merit value of all hardware implementation options of operation x should be larger than that of the software one, and (2) under the same performance improvement, the hardware implementation option with less hardware cost should have larger merit value. Initially, $merit_{x,y}$ is assigned as the product of $reduced_cycle_{x,\mathrm{MAX}} + 1$ and $merit_{x,0}$, where $reduced_cycle_{x,\mathrm{MAX}}$ is the maximum achievable execution cycle reduction of operation x. The algorithm then checks whether the execution cycle reduction of implementation option y is equal to $reduced_cycle_{x,\mathrm{MAX}}$. If yes, then the algorithm adjusts the merit value, according to the difference of the hardware cost between the yth and the implementation option with the largest hardware cost of operation x. Here, $area_{x,\mathrm{MAX}}$, represents the maximal hardware cost of the implementation option of operation x. If the execution time reduction of hardware implementation option y is less than $reduced_cycle_{x,\mathrm{MAX}}$, then the merit of implementation option y is divided by the difference between $cycle_saving_{x,\mathrm{MAX}} + 1$ and $reduced_cycle_{x,y}$.

3.3 Algorithm Complexity

Due to the nondeterminism of ACO algorithm, the complexity of the whole algorithm is not able to be estimated. Therefore, we estimate the complexity of the individual step of our algorithm. The time complexity of performing steps 1, 2, 3, 6, and merit value calculation in the flow of ISE exploration shown in Fig. 3 is $O(n)$, where n is number of operations in the basic block. The time complexity of performing *Hardware_Grouping* for all operations is $O(mn^2)$, where m is number of hardware implementation options for an operation. The time complexity of steps 5 and 7 are $O(1)$, since these steps could be done in a constant time.

4. SIMULATION RESULTS

Section 4 demonstrates the simulation results of the proposed and the related approaches [5]. To validate the advantages of the proposed approach, several numerical results, such as performance improvement, hardware cost, and area efficiency, are given.

4.1 Simulation Setup

In this simulation, we evaluated two ISE exploration algorithms: one was proposed by ourselves, and another was the genetic algorithm proposed by Pozzi [5]. Because of the nondeterminism of genetic algorithm, it is also impossible to estimate the complexity of the whole algorithm of Pozzi, and the complexity of a single iteration is $O(n)$, where n is the number of operations in the basic block. Both evaluated algorithms were imple-

Table 1. Benchmark used in simulation.

	Benchmarks
Mibench	adpcm decode/encode, basicmath, bitcount, blowfish, CRC32, JPEG decode/encode, rijndael decode/encode, SHA, stringsearch, susan
Mediabench	epic decode/encode, FFT, inverse-FFT, G.721 decode/encode, MPEG2 decode/encode

mented by us in C++ language. All benchmarks used in this simulation were selected from Mibench [11] or Mediabench [12] and listed in Table 1. To enlarge the solution space for a better solution, all benchmarks were compiled with function inlining and loop-unrolling.

In addition, to demonstrate the benefit of performing hardware design space exploration, four scenarios were evaluated in this simulation: (1) the proposed algorithm; (2) the proposed algorithm with fastest (largest) hardware implementation option (*i.e.*, without performing the hardware design space exploration); (3) Pozzi's algorithm with fastest (largest) hardware implementation option; and (4) Pozzi's algorithm with slowest (smallest) hardware implementation option. Moreover, we also made the following assumptions in this simulation:

1. The ISA used in this work was MIPS R3000. The issue width of the processor core adopted was one, and its execution frequency was 300MHz. The base architecture of the processor consisted of only one ALU and one register file with 32 of 32-bits entries and two read ports (R) and one write port (W).
2. The software execution cycle of all instructions was one cycle, except for multiplication (3 cycles) and division (12 cycles) operations.
3. Memory access (*Reason*: difficult to estimate the execution cycle), flow control (*Reason*: simplify the complexity of the control circuit of ISE_ HW), and division as well as the floating point (*Reason*: high hardware cost) operations could not be packed into ISE.

In this simulation, hardware was synthesized in 0.13μm technology. All operations packed into ASFU were 32-bit. The synthesized results of hardware implementation options are shown in Table 2 and the hardware costs of various register files differed from the numbers of read (R)/write (W) ports are shown in Table 3.

The parameters used in this work are summarized as follows,

Table 2. Hardware implementation options setting.

Operation	Delay (ns)	Area (μm^2)	Operation	Delay (ns)	Area (μm^2)
Add/Sub	0.50	5768.0	Multiply	1.66	30814.6
	0.83	3569.2		2.49	24534.2
	1.33	3336.4		3.16	19262.1
AND	0.03	230.8	OR/XOR	0.03	242.7
	0.09	217.3		0.12	217.3
NOR	0.02	173.1	Barrel shifter	0.26	4773.1
	0.03	163.0		0.31	3739.4
				1.48	1544.6

Table 3. Area of register file architectures (32 × 32-bit).

Spec.	Area (μm^2)	Number of single adders (Area/5768.0μm^2)
2R/1W	121144.2	21.0
4R/2W	195283.0	33.9
6R/3W	252503.7	43.8
4R/8W	323060.0	56.0

1. α ($0 < \alpha < 1$): the relative influence of merit and trail in Eq. (1). A larger α means that the current result relies more on the trail, so that the algorithm behaves more like a greedy approach.
2. ρ ($\rho \in$ integer): the trail update factor in Eq. (3). A smaller ρ makes the algorithm converge slowly, but has higher opportunity to obtain a better solution.
3. β_{no_p}, β_{io}, and β_{convex} ($0 < \beta < 1$): the penalty for a no-speedup or illegal ISE. Violating the register read/write port and/or the convex constraints will generate an illegal ISE, but a no-speedup ISE is usually legal. Thus, to avoid the generation of illegal ISE, violating register and/or convex constraints must have larger penalty than no speedup; consequently, β_{io} and β_{convex} are smaller than β_{no_p}.

To properly select all parameters used in the simulation, we adopted a trial-and-error approach. In this simulation, the initial merit value of the software and hardware implementation option were 100 and 200, respectively; the initial trail value of all implementation options was 0; α was 0.3; the evaporating factor ρ was 4; and the merit function had $\beta_{no_p} = 0.9$, $\beta_{io} = 0.7$, and $\beta_{convex} = 0.5$.

4.2 Experimental Results

In this subsection, we first compare the execution performance improvement and the hardware cost of our and Pozzi's algorithms [5]. Then, based on the results of the performance improvement and the hardware cost, the area efficiency of these two algorithms is also shown in this subsection. The abbreviations used in the following figures are listed as followings:

- "Our" and "Ge" denote the proposed algorithm and that of Pozzi [5], respectively.
- "D" means to perform hardware design space exploration of ISE_HW, while "F" and "S" denote the fastest (largest) and slowest (smallest) hardware implementation option being used to realized ISE_HW, respectively. Note that "S" and "F" does not perform hardware design space exploration of ISE_HW.
- The numbers between two dashes indicates the numbers of read/write ports of the register file in use.

The common ISE selection criterion is (1) under a limited number of ISEs; (2) under a limited area budget; and (3) under both limitations. To simplify the ISE selection, in this work, all explored ISEs were sorted in the decreasing order of their performance improvement and chosen based on this sorting result under a limited number of ISEs (*i.e.*, the first criterion).

The processing time of Our was usually larger than that of Ge. In most cases, exploration results of Ge could be obtained within 10 seconds. However, the processing time of our highly depends on the size of the basic block and the numbers of read/write ports of the register file in use. For example, if the size of the basic block is 96, the processing times of 2/1, 4/2, 6/3, and 8/4 (read/write ports) were 0.09, 0.35, 2.13, and 3.46 seconds, respectively; if the size of the basic block is 359, then 1.06, 29.04, 35.39, and 409.16 seconds are required, respectively. Since ISE exploration is performed statically, it is acceptable to take more time for better results.

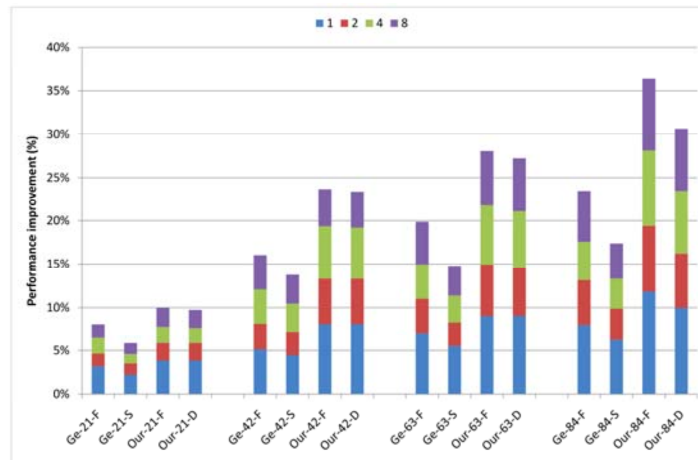


Fig. 6. Performance improvement.

Fig. 6 depicts the performance improvement of Our and Ge approaches. Recall that ISEs were selected based on their performance improvement. In this figure, each segment within a bar from bottom to top indicates the execution performance improvement when 1st, 2nd, 3rd-4th, and 5th-8th ISE are selected, respectively. The average size of ISE was very close between Our and Ge, and Our outperformed Ge in all cases. The main reason for the better performance of Our is that ISEs explored by Our had higher similarity than that by Ge. ISEs with higher similarity implies that more ISEs may be merged and, thus, more hardware resource may be shared among them. Therefore, Our may achieve higher performance improvement under the same number of ISEs or the same hardware cost. The reason why ISEs explored by Our had higher similarity than that of Ge is that ISE was explored in a nondeterministic fashion in Ge because of the nature heuristic of genetic algorithm, while that in a more deterministic fashion in Our since it rely on not only ACO algorithm but also a "deterministic" merit function.

In addition, the performance gap between Our and Ge increased when relaxing the register file constraint, *i.e.*, increasing the number of register read/write ports. Relaxing the register file constraint means that more operations may be packed into an ISE legally. Accordingly, the structure and the supporting operation types among ISEs will become more diverse. This would lower the similarity of ISE, especially for Ge in which no merit function is used to guide the choosing of operations to form an ISE.

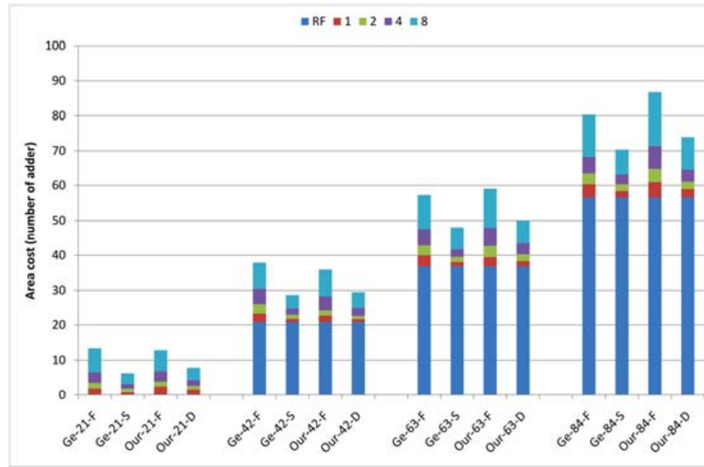


Fig. 7. Hardware_cost (ISE_HW and the extra hardware cost of register file).

Fig. 7 depicts the hardware cost of Our and Ge algorithms. In this figure, each bar consists of several segments where the bottom one is the extra hardware cost of the register file and others indicate the hardware cost when 1st, 2nd, 3rd-4th, and 5th-8th ISE are selected, respectively. The hardware cost shown in Fig. 7 is defined as follows,

$$\text{Hardware_cost} = \frac{\sum_{i=1}^{n} \text{area size of ISE}_i + \text{extra area size of register file}}{\text{Area size of a single adder}}, \tag{5}$$

where n is the number of total selected ISEs and the area size of a single adder is 5768.0 μm^2. Obviously, Our-D can achieve a good trade-off between the performance improvement and the hardware cost. Furthermore, one interesting point emerging from this figure is that most of the hardware cost was consumed by the register file; that is, although relaxing the register file constraint could increase the performance improvement, the hardware cost paid on the register file also increased significantly.

To identify the benefit of performing hardware design space exploration of ISE_HW, we defined a measurement criterion, called area efficiency, to represent how much performance improvement could be achieved per unit area consumed by the extra hardware cost as follows,

$$\text{Area_efficiency} = \frac{\sum_{i=1}^{n} \text{performance improvement of ISE}_i}{\text{Extra hardware cost}}. \tag{6}$$

Fig. 8 illustrates the difference ratio of the area efficiency with and without extra area size of register file between Ge-F and other scenarios. In all of the cases, Our-D had the highest area efficiency among all scenarios. This evidences the benefit of performing hardware design space exploration of ISE_HW. In addition to achieve higher area efficiency, performing hardware design space exploration can also have better performance improvement under the same area constraint. After deducting the extra area size of regi-

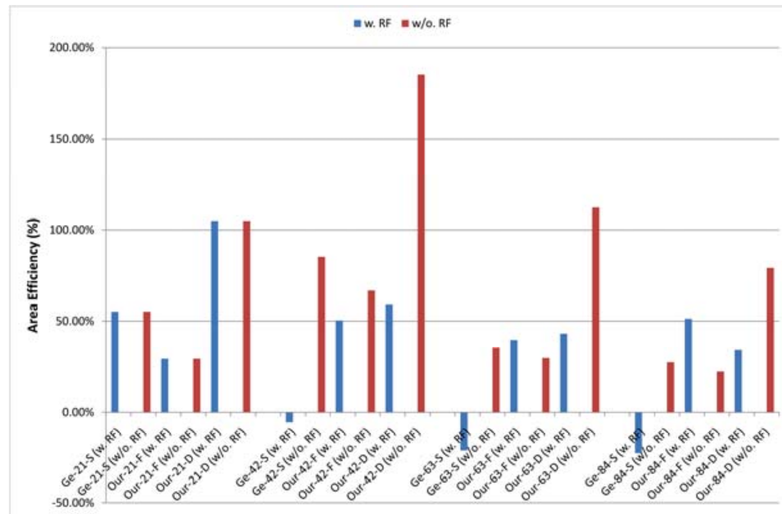


Fig. 8. Area_efficiency.

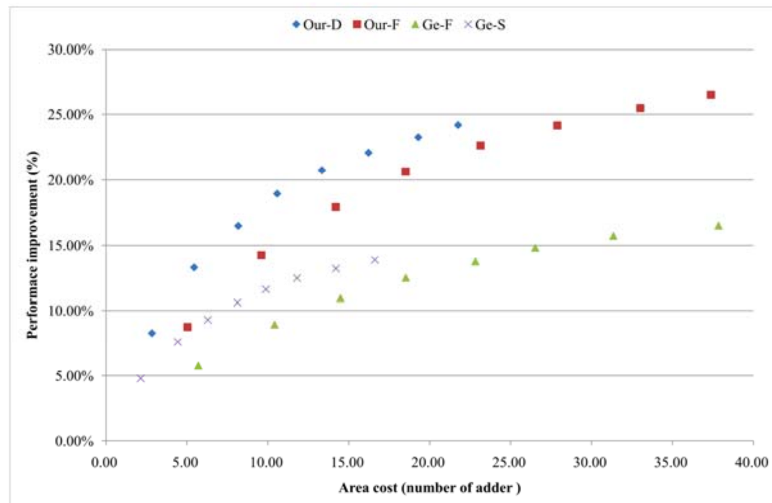


Fig. 9. Performance improvement vs. area cost (average).

ster file, three interesting findings appear in Fig. 8. First, all scenarios (Ge-S, Our-F and Our-D) had higher area efficiency of ISE_HW than Ge-F. This implies that the extra area size of register file offsets most benefit of the performance improvement. Second, among all register file constraints, each scenario of "42" (Our-42 and Ge-42) had the highest area efficiency. Third, except for "21" (Our-21 and Ge-21), the area efficiency decreased in proportion to the number of register read/write ports. Relaxing the register port constraint increases the number of operations being packed into ISE. However, packing more operations into ISE usually leads to the increment of the number of operations on the critical path of ISE. This increment reduces the opportunity of using a slower and smaller

hardware implementation option for most of them. Consequently, the area efficiency is difficult to gain by performing hardware design space exploration when relaxing the register port constraint.

Fig. 9 shows the relationship between performance improvement and area cost. In this figure, every scenario has 8 points and each point means an ISE being selected. Our-D achieved best performance improvement under same area budget (set a fixed value at X-axis). This implies that Our-D had best area efficiency under same area budget. In addition, power consumption is one of the critical design considerations in many systems. The power consumption can be divided into two types: dynamic and static. Both types of power consumption are directly proportional to the area size of the circuit. This implies that an ISE with higher hardware cost would consume lager power. Because Our-D had less area consumption under same performance improvement (set a fixed value at Y-axis), Our-D had less static/dynamic power consumption under the same performance improvement.

5. CONCLUSION

This paper reveals the benefits of integrating hardware design space exploration into ISE exploration. In general, ISE_HW has multiple hardware implementation options because of different area and latency requirement. However, several of them are the same in the performance improvement but different in the area cost under the pipeline-stage timing constraint. This phenomenon provides an excellent opportunity to avoid unnecessary waste of hardware cost by exploring the hardware design space of ISE_HW. To integrate hardware design space exploration into ISE exploration, we proposed an ISE exploration algorithm that explores not only ISEs but also their corresponding hardware design space. Simulation results demonstrated that the proposed algorithm significantly improved area efficiency.

Additionally, we recommend addressing several issues in future work. First, several studies [13, 14] have shown that combining multiple meta-heuristic algorithms, such as combining genetic algorithm with ACO one, to solve the optimization problem may outperform only single one. Therefore, whether other meta-heuristic algorithms and our proposed algorithm together will further improve the results would be an interesting issue for further research. Second, the parameters (α, ρ, β_{no_p}, β_{io}, and β_{convex}) would influence the simulation results. However, in this work, we adopted one set of parameters for all cases. Therefore, it would be an interesting future work to explore the space of these parameters to reveal the relationship between these parameters and the features of the processors and applications, for examples, the number of register file read/write ports and the size of basic block.

REFERENCES

1. D. S. Rao and F. J. Kurdahi, "Partitioning by regularity extraction," in *Proceedings of Design Automation Conference*, 1992, pp. 235-238.
2. C. Liem, T. May, and P. Paulin, "Instruction-set matching and selection for DSP and ASIP code generation," in *Proceedings of European Design and Test Conference*, 1994, pp. 31-37.

3. P. Yu and T. Mitra "Disjoint pattern enumeration for custom instructions identification," in *Proceedings of International Conference on Field Programmable Logic and Applications*, 2007, pp. 273-278.
4. C. Galuzzi, K. Bertels, and S. Vassiliadis, "A linear complexity algorithm for the automatic generation of convex multiple input multiple output instructions," *International Journal of Electronics*, 2008, pp. 27-29.
5. L. Pozzi, K. Atasu, and P. Ienne, "Exact and approximate algorithms for the extension of embedded processor instruction sets," *IEEE Transactions on Computer Aided Design*, Vol. 26, 2006, pp. 1209-1229.
6. N. T. Clark, H. Zhong, and S. A. Mahlke, "Automated custom instruction generation for domain-specific processor acceleration," *IEEE Transactions on Computers*, Vol. 54, 2005, pp. 1258-1270.
7. C. Galuzzi and K. Bertels, "The instruction-set extension problem: a survey," in *Proceedings of International Workshop on Applied Reconfigurable Computing*, 2009, pp. 209-220.
8. F. Sun, S. Ravi, and N. K. Jha, "Custom-instruction synthesis for extensible-processor platforms," *IEEE Transactions on Computer-Aided Design*, Vol. 23, 2004, pp. 216-228.
9. M. Dorigo, V. Maniezzo, and A. Colorni, "Ant system: Optimization by a colony of cooperating agents," *IEEE Transactions on Systems*, *Man and Cybernetics*, Vol. 26, 1996, pp. 29-41.
10. G. Wang, W. Gong, and R. Kastner, "Application partitioning on programmable platforms using the ant colony optimization," *Journal of Embedded Computing*, Vol. 2, 2006, pp. 119-136.
11. M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of IEEE Annual Workshop on Workload Characterization*, 2001, pp. 3-14.
12. C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A tool for evaluating and synthesizing multimedia and communications systems," in *Proceedings of International Symposium on Microarchitecture*, 1997, pp. 330-335.
13. M. L. Pilat and T. White, "Using genetic algorithms to optimize ACS-TSP," in *Proceedings of the 3rd International Workshop on Ant Algorithms*, 2002, pp. 282-287.
14. D. Gong and X. Ruan, "A hybrid approach of GA and ACO for TSP," in *Proceedings of the World Congress on Intelligent Control and Automation*, Vol. 3, 2004, pp. 2068-2072.

I-Wei Wu (吳奕緯) received his B.S. and M.S. degrees from the Department of Electronics Engineering, Feng Chia University of Taiwan in June 2000, June 2002 respectively. He is now a Ph.D. student in the Department of Computer Science, National Chiao Tung University of Taiwan. He is interested in custommizable processor, low power computer architecture, and compilation technologies for embedded system.

Chung-Ping Chung (鍾崇斌) received the B.E. degree from the National Cheng Kung University, Tainan, Taiwan, Republic of China in 1976, and the M.E. and Ph.D. degrees from the Texas A&M University in 1981 and 1986, respectively, all in Electrical Engineering. He was a lecturer in Electrical Engineering at the Texas A&M University while working towards the Ph.D. degree. Since 1986 he has been with the Department of Computer Science at the National Chiao Tung University, Hsinchu, Taiwan, R.O.C., where he is a Professor and the director of the Institute of Biomedical Engineering. From 1998, he was on leave from the University and joined the Computer and Communications Laboratories, Industrial Technology Research Institute, R.O.C. as the director of the Advanced Technology Center, and then the Consultant of the General Director's Office, until 2002. He also served as the editor-in-chief in the Information Engineering Section of the Journal of the Chinese Institute of Engineers (EI, SCI), R.O.C., in 2000 to 2005. He is now the editor of the Journal of Information Science and Engineering (SCI). He has led the Computer Systems Laboratory in CS Department, NCTU since 1992, served as the consultant and reviewer for numerous information and IC companies and government organizations, published over two hundred refereed technical papers, and obtained approximately ten patents. His research interests include computer architecture, parallel processing, embedded system and SoC design, and parallelizing compiler.

Jean Jyh-Jiun Shann (單智君) received the B.S. degree in Electronic Engineering from Feng Chia University, Taiwan in 1981, the M.S.E. degree in Electrical and Computer Engineering from University of Texas at Austin, U.S.A. in 1984, and the Ph.D. degree in Computer Science and Information Engineering from National Chiao Tung University, Taiwan in 1994. She is an Associate Professor in the Department of Computer Science at the National Chiao Tung University. Her research interests include computer architecture, parallel processing, embedded systems, and virtual machines.