A Hardware/Software Partitioning Algorithm for Pipelined Instruction Set Processor

Nguyen Ngoc Binh, Masaharu Imai,
Akichika Shiomi, and Nobuyuki Hikichi*

Department of Information and Computer Sciences
Toyohashi University of Technology
Toyohashi, 441 Japan

* Software Research Associates, Inc.
Tokyo, 170 Japan

Abstract

This paper proposes a new method to design an optimal instruction set for pipelined ASIP development using a formal HW/SW codesign methodology. The codesign task addressed in this paper is to find a set of HW implemented operations to achieve the highest performance of a pipelined ASIP under a given gate count and power consumption constraint. The method enables to estimate the performance and pipeline hazards of the designed ASIP very accurately. The experimental results show that the proposed method is effective and quite efficient.

1 Introduction

Pipelining is a simple yet effective technique for increasing parallelism and the utilization of Functional Units (FUs) [1]. It is frequently used for increasing the performance of an instruction set processor by overlapping the execution of instructions. The total execution cycles needed for a series of instructions are reduced in proportion to the amount of overlap in their processing (i.e., the number of stages in the pipeline).

In designing the pipelined instruction set processor it is necessary to deal with all types of pipeline hazards, especially when multi cycle and pipelined FUs are used. In the traditional Application Specific Integrated Processor (ASIP) design methodology, system architects decide which operations will be implemented in hardware (HW) or in software (SW). In order to produce an efficient design in a reasonable turn-around-time (TAT), an efficient HW/SW codesign partitioning method should be used. In practice, design process still depends on the designer's skill, and the best possible design is usually achieved after investigating a number of design candidates.

This paper focuses on the optimization of a pipelined instruction set processor using a formal HW/SW codesign methodology. In the case of area and power consumption constrained design, some of the operations may be implemented in SW using an ALU only. During the execution of these SW operations, other ALU operations must be stalled. However, FUs may operate simultaneously as long as no more than one instruction can be issued at each clock cycle and there are no two instructions which complete their execution at the same time. That is, the method must handle all types of data hazards and structural hazards.

The rest of the paper is organized as follows. Section 2 reviews related work and describes an architecture model to be considered in this paper. Section 3 gives definitions and notations. Section 4 describes a new problem formalization and the proposed algorithm. The effectiveness and efficiency of the proposed algorithm are shown in Section 5 through design samples. Section 6 gives the conclusion and future work.

2 Related Work

A HW/SW codesign system PEAS-I (Practical Environment for ASIP development - type I) [2] has been developed to synthesize an optimal instruction set processor by solving Instruction set implementation Method Selection Problems (IMSP) type 1 and 2. IMSP-1 [3] is set up assuming no interaction among the operations, and each operation was to be implemented using a separate HW module. However, IMSP-2 [4] is an extension of IMSP-1 by taking resource sharing into account.

The target CPU to be generated by PEAS-I belongs to a class of Harvard architecture with separate data bus and instruction bus. The PEAS-I CPU core architecture is shown in Figure 1, where 'Kernel' consists of an ALU, a one-bit shifter, and a register file. The CPU core may include other FUs such as multiplier, divider, and so on. The pipelined architecture synthesized by PEAS-I consists of four stages: IF (Instruction Fetch and decode), EX (EXecution), MEM (MEMory access) and WR (Write back to Register file), respectively. While each of IF, MEM, and WR stages takes only one cycle, EX stage takes one or more cycles. The PEAS-I CPU has a RISC type load/store (register-register) architecture and each control step corresponds to one clock cycle. While the CPU may contain the Kernel and different types of FUs, it is assumed that there are no identical FUs. The architecture has a register bypass to forward the computation results to Kernel or FUs. Each FU can be multi cycle and pipelined. However, in the IMSP-1 and 2 formalizations, the pipeline was not taken into account, in particular, pipeline hazards were not addressed and left for future work [2]. Therefore, we need to develop a method to estimate the effect of pipeline hazards as accurately as possible because they affect the performance of the CPU. Moreover, the method should estimate the execution cycles of the application program, i.e. the performance of the ASIP, as accurately as possible.

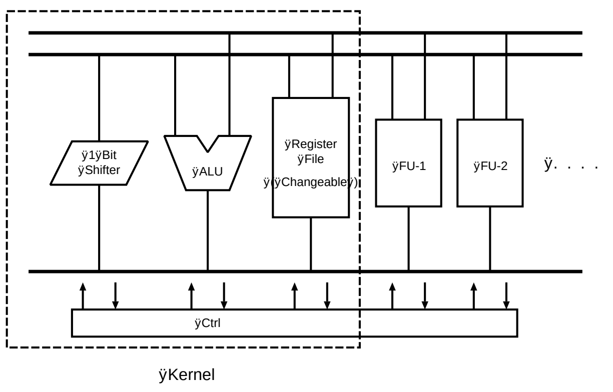

Figure 1: PEAS-I CPU core architecture.

Huang and Despain [5] propose a systematic approach to synthesize an instruction set that the given application SW can be efficiently mapped to a parameterized, pipelined microarchitecture. While their work is similar to our work in terms of the inputs and part of outputs, it is different from our method in terms of approach and efficiency. First, Huang and Despain assume that the designers are required to specify the number of HW resources (in parlicular, the number of each type of FU), which will take several iterations to find the best allocation; whereas in our method, an optimal pipelined architecture can be selected and generated automatically. Second, Huang and Despain synthesize an instruction set from the assembly code by grouping some instructions into a new one; whereas we generate the optimal instruction set from a super set regarding the optimal HW/SW partitioning.

3 Definitions and Notations

The architecture of an ASIP synthesized by the PEAS-I system is based on the GNU C Compiler (GCC) abstract machine model [6]. The GCC Register-Transfer Language(RTL) operations are divided into **primitive** and **basic** operations. The primitive operations contain the minimum operations that can be included in the ASIP chip so that it can execute any C program. The primitive operations should be implemented in HW as the Kernel. The basic operations contain other C operators that are not included in the primitive operations. A basic operation can be implemented using some HW choices (such as fast or slow HW modules) or using a SW subroutine (runtime routine) that uses primitive operations and some other basic operations.

The HW/SW partitioning problem in the current version of PEAS-I is defined as follows [4]:

For a whole set of all candidate instructions representing a given application domain, select a set of implementation methods which maximizes the performance of the CPU under the constraints of chip area and power consumption, taking into account the functional module sharing relation among instructions.

In order to formalize the IMSP-2P the following definitions and notations are used in the remainder of this paper.

(1) "n" denotes the total number of basic operations to be considered.

(2) "f_i" denotes the execution frequency count of basic operation $\#i$ in the given set of application programs, where $1 \leq i \leq n$. We denote frequency count of all primitive operations as f_0.

(3) "x_i" denotes an implementation method that realizes operation $\#i$, where x_i may be HW choice or SW, $0 \leq i \leq n$. Then $X = (x_0, x_1, ..., x_n)$ is a combination of implementation methods to be considered.

(4) "$t_i(x_i)$" denotes the execution cycles of operation $\#i$ when implemented by method x_i, where $0 \leq i \leq n$.

(5) "$a(x_i)$" and "$p(x_i)$" denote the area and power consumption required for implementation method x_i respectively, where $0 \leq i \leq n$.

(6) "A_max" and "P_max" denote the available chip area and the maximum power consumption allowable for the computing module in the ASIP chip.

(7) "N" denotes the total number of basic blocks in the application program's GCC RTL code.

(8) "$t(B_j, X)$" denotes the execution cycles needed to execute basic block B_j using a combination of implementation methods X, where $1 \leq j \leq N$.

(9) "F_j" denotes the execution frequency count of basic block B_j in the given set of application programs, where $1 \leq j \leq N$.

(10) "c_j" denotes clock cycles needed to define control (e.g., branch delay) from block B_j to another one, where $1 \leq j \leq N$. Here, it is assumed that all branches are taken and delay slot scheduling is not performed.

(11) "b" denotes execution cycles reduced by un-taken branches in execution of the given application program.

Note that f_i, F_j, c_j, and b are computed from the application program and associated input data by using the application program analyzer (APA) of the PEAS-I system.

4 Proposed Method

The problem addressed in this paper for designing an optimal pipelined instruction set processor in PEAS-I is called IMSP-2P ('P' stands for Pipeline) and can be considered as an extension of IMSP-2 toward the pipelined architecture. Our goal is to detect and resolve pipeline hazards and to increase the performance of pipelined ASIP to be designed as much as possible in the new problem formalization.

4.1 IMSP-2P Formalization

Find a solution vector

$$\mathrm{X} \;=\; (x_0,\; x_1, \cdots,\; x_n)$$

which minimizes the objective function:

$$T(X) = \sum_{j=1}^{N} (F_j \times (t(B_j, X) + c_j)) \;\; - \;\; b \;\;, \qquad (1)$$

subject to the constraints:

$$\sum_{x_i \in S} a(x_i) \leq A\_max, \qquad (2)$$

$$\sum_{x_i \in S} p(x_i) \leq P\_max, \qquad (3)$$

where

$$S \;\;=\;\; \bigcup_{i=0}^{n} \{x_i\} \qquad (4)$$

4.2 Consideration

We have developed a HW/SW partitioning-oriented pipeline scheduling algorithm [7] to estimate $t(B_j, X)$ for basic block B_j for given HW resources X. The pipeline control hazards are addressed in introducing the coefficients c_j. Note that the number of clock cycles due to control hazards is equal to $\sum_{j=1}^{N}(F_j \times c_j) \;\; - b$. The pipeline scheduling algorithm detects and resolves all types of data hazards and structural hazards by ensuring that no more than one instruction can be issued or completed at each control step. Therefore, the IMSP-2P solver estimates the pipeline execution cycles accurately if the code optimization is performed by the same strategy taken in the scheduling algorithm.

4.3 IMSP-2P Solver

4.3.1 Input and Output

The input to the IMSP-2P solver includes the following items:

(1) the GCC's RTL code of the given application program,

(2) F_j's for $j = 1, \cdots, N$,

(3) b (# clock cycles reduced by un-taken branches),

(4) area and power consumption constraints A_max and P_max, and

(5) the **module information database**, which includes execution cycle count, latency, area, and power consumption of each implementation method of all operations.

The output of the IMSP-2P solver includes the optimum implementation method of each basic operation and pipelined schedules of basic blocks. The instruction set of the designed ASIP will include the primitive operations as default and those basic operations that are selected to be implemented in HW. The algorithm tries to automatically integrate the functional modules, which share basic operations, into one HW module whenever possible.

4.3.2 Algorithm

The IMSP-2P can also be solved using the branch-and-bound method as IMSP-2 can. The key to solving problems efficiently by this method is to find a tight lower-bound function to prune as many non-optimum solutions as early as possible. The lower-bound function used in the IMSP-2P solver is as follows:

$$\begin{aligned} Lower\_bound = (f_0 + Stall_{fast}) \\ + \sum_{i=1}^{d-1} (f_i \times u_i(x_i)) \; + \; \sum_{i=d}^{n} f_i \;\;, \qquad (5) \end{aligned}$$

$$u_i(x_i) = \begin{cases} 1, & \text{if } x_i \text{ is a HW implementation ,} \\ t_i(x_i), & \text{if } x_i \text{ is a SW implementation ,} \end{cases}$$

where the parameter d represents the depth of the node under consideration. The first term in Eq.(5) represents the value independent of X, where $Stall_{fast}$ is the number of stalls in executing the given application program using the hypothetical FUs of one cycle denoted by X_{fast} and is computed as follows:

$$Stall_{fast} = T(X_{fast}) - \sum_{i=0}^{n} f_i \;\;, \qquad (6)$$

where $T(X_{fast})$ is computed by using Eq.(1).

The second term in the lower-bound in Eq.(5) represents the value less than determined cost of the already searched path, while the third term represents the minimum cost of the remaining path. Note that instructions are executed in a pipeline manner and can be overlapped. Therefore, in the best case, instructions may be overlapped maximally and there are no idle clock cycles. In this case, instructions will be executed as if they were executed in one cycle each. However, when operations are implemented in SW, they cannot be overlapped since only the Kernel is in charge of their execution. This principle also explains why we used the hypothetical FUs X_{fast} in defining the lower-bound function. Also, note that execution frequency counts f_i's ($i = 0, 1, ..., n$) are computed by using the inputs (1) and (2). Please note that the module sharing capability and heuristic reordering are the same as in the IMSP-2 solver.

5 Experiments and Results

The IMSP-2P algorithm has been implemented in C and examined on a workstation. A set of sample programs has been performed to evaluate the effectiveness and efficiency of the algorithm.

5.1 Module Library

We use a module library with both non-pipelined FUs and pipelined FUs such as multipliers and dividers generated using a high-level synthesis system called PARTHENON [9] and cell library VSC470.lib (0.8μmCMOS) from VLSI Technology, Inc. A 16 MHz clock was assumed in the design of HW modules. The database contains 14 basic operations, each of them has different implementation methods ranging from 2 to 11. The number of leaf nodes in the search tree is of $11^2 \times 7^4 \times 2^8 = 74,373,376$. The whole search tree ranges from 8.2×10^7 to 1.5×10^8 nodes depending on the order of variables (x_i's) to be examined. Therefore, it is necessary to have an efficient strategy to explore the search space to get an optimal solution in a reasonable time.

Table 1: Part of Module Database with Pipelined Multipliers and Dividers

Module Name	*Gate Count*	*Power**	*L*	*D*	*Implied Operations*
kernel**	14918	18062.3	1	1	(primitive)
b_alsft	756	876.3	1	1	ashl, ashr, lshl, lshr
extend	137	172.8	1	1	extendhi, extendqi, z_extendhi, z_extendqi
mul_csa	7747	11106.9	1	1	mul, umul
mul_3clk	6118	8008.5	3	3	
mul_bpr	3161	3643.0	17	17	
mul_seq	2393	2777.1	32	32	
mul_seq_p4	14567	17138.4	4	32	
mul_seq_p8	7586	8989.1	8	32	
mul_seq_p16	4052	4829.7	16	32	
mul_bpr_p2	19552	23103.2	2	16	
mul_bpr_p4	10149	12029.9	4	16	
mul_bpr_p8	5400	6497.5	8	16	
div_2seq	5808	6910.4	19	19	div, udiv, mod, umod
div_seq	3396	3931.4	35	35	
div_seq_p17	5458	6628.6	17	34	
div_2seq_p3	29127	34804.9	3	18	
div_2seq_p6	15499	18362.8	6	18	
div_2seq_p9	10744	12713.5	9	18	

* Unit: μWatt/MHz ** with 8 registers
L: Latency (cycles) D: Delay (cycles)

Part of the module information database used in the experiments is shown in Table 1. In this table, 'kernel' represents the minimal HW components. The module 'b_alsft' denotes a barrel shifter that performs both arithmetic and logical shift operations such as **ashl**, **ashr**, **lshl**, and **lshr**. The module 'extend' performs extension operations such as **extendhi**, **extendqi**, **z_extendhi**, and **z_extendqi**. The modules 'mul_csa', 'mul_3clk', 'mul_bpr', and 'mul_seq' denote multipliers that execute a 32-bit $\times$ 32-bit multiplication such as **mul** or **umul** in 1, 3, 17, and 32 clock cycles, respectively. The modules 'div_2seq' and 'div_seq' represent dividers that execute 32 bit division operations such as **div**, **udiv**, **mod**, and **umod** in 19 and 35 clock cycles, respectively. These modules are non-pipelined, therefore, each delay and latency are the same. The pipelined multipliers are denoted as 'mul_seq_p4', $\cdots$, 'mul_bpr_p8', and dividers as 'div_seq_p17', $\cdots$, 'div_2seq_p9'.

Table 2: Expected execution cycles of SW implemented operations

Basic Operation	*#Cycles*	*Basic Operation*	*#Cycles*
div	216	mul	96
udiv	202	umul	91
mod	214	trunchi	2
umod	201	truncqi	1
ashl	31	extendhi	10
ashr	31	extendqi	9
lshl	31	z_extendhi	2
lshr	31	z_extendqi	1

Another part of the module database describes the expected execution cycles for each basic operation to be implemented in SW by using the 'kernel' only. This part is shown in Table 2, where **trunchi** and **truncqi** represent the truncation operations.

5.2 Sample Programs

The sample programs used in the experiments are as follows:
(1) ESS : Equation System Solver program, which solves a system of two linear equations using Cramer's rule.
(2) IMC : Inverse Matrix Calculator program that computes the inverse of a non-singular 3×3 matrix using Cramer's rule.
(3) *diffeq* : A program for solving a second order differential equation from Ref. [8].

These sample programs were fed to APA of the PEAS-I system. The code optimization was performed by the Gnu C Compiler [6].

5.3 Algorithm Efficiency

Experimental results show that the proposed algorithm with the lower bound function in Eq.(5) is very efficient. Selecting the optimum architecture out of 7.44×10^7 combinations is not an easy task, even to be solved by using the integer linear programming (ILP) approach.

An analyzer has been developed to reduce the search space and to get necessary information for the proposed algorithm, such as GCC RTL code, execution frequency (F_j) of each basic block (BB), data dependencies between instructions in BB, and the number of basic operation types in each BB. A basic block is said to be *dependent* on X if it contains instruction(s) with the basic operation(s), otherwise it is called *independent* of X. Note that a combination of implementation methods X is determined when a leaf node in the search tree has been reached. Then, using the pipeline scheduler [7] the IMSP-2P algorithm computes the values $t(B_j, X)$ for BBs only when these BBs are dependent on X, for other BBs (i.e., independent of X) they are computed only once before exploring the search tree.

It has been found that the number of BBs dependent on X (denoted as N') is usually much smaller than the total number of BBs of the given application program. The number of basic operation types is six for these sample programs. The reduced search space contained 4935 nodes. The analyzed results are summarized in Table 3. The number of performed experiments is shown in Table 3 (#Cases) for each sample program. Note that while the total number of nodes in each reduced search tree is about 5000, the average numbers of visited nodes are 75, 114, and 112 for the ESS, IMC and *diffeq* examples, respectively. That means Eq.(5) employs a good lower bound function. Throughout the experiment, the algorithm gave the optimum solution within a few seconds on a SPARCstation 10 workstation, including the time for analyzing the input data to get necessary information such as data flow graph, the number of basic operations, reduced search space, and so on. For a given area constraint, the IMSP-2P solver has taken CPU time of 2.6s, 2.3s, and 2.1s in average for IMC, ESS, and *diffeq*, respectively.

Table 3: Analyzed results and statistics

Specification	*ESS*	*IMC*	*diffeq*
N	66	76	50
N'	7	7	7
b	1673	2470	447
# Basic Operation Types	6	6	6
# Nodes of Reduced Tree	4935	4935	4935
# Cases	26	31	29
# Visited Nodes in Average	75	114	112

5.4 Algorithm Effectiveness

The effectiveness of the IMSP-2P algorithm has been evaluated by using it to select the implementation methods of the basic operations. The IMSP-2P selected the optimum partitioning for different values of area constraint. The power consumption was ignored to simplify the experimental cases.

FUs needed to implement basic operations for these sample programs are a multiplier, a divider, a barrel-arithmetic shifter, an extender and so on, where the multiplier and the divider can be pipelined or non-pipelined.

Figures 2 and 3 show estimation errors by IMSP-2 and IMSP-2P, respectively. Note that IMSP-2P can estimate the execution cycles much more accurately than IMSP-2. That is, estimation errors are below 1.3% for designs with gate count constraints exceeding 22 Kgates, where all operations are implemented in HW. On the other hand, the estimation errors by IMSP-2 range from 5% to 20% because of not taking into account the pipeline hazards.

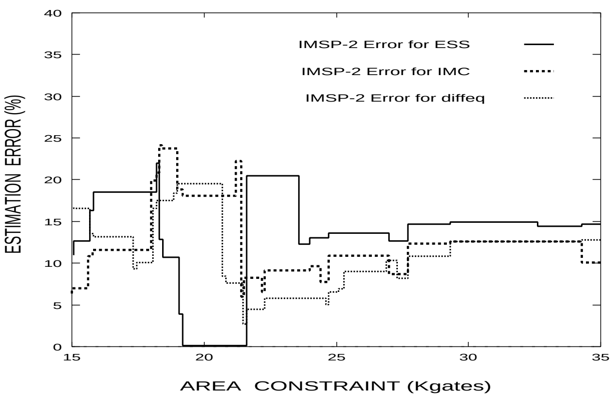

Figure 2: Estimation errors by IMSP-2

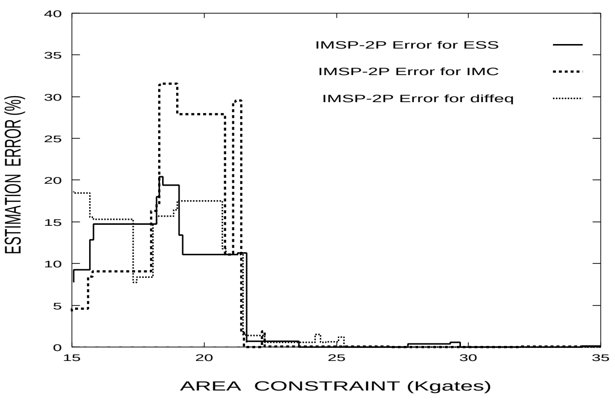

Figure 3: Estimation errors by IMSP-2P

Please note that in the scheduling process it was assumed that each SW implemented basic operation is executed in the expected fixed number of clock cycles as shown in Table 2. In practice, these SW implemented basic operations will take different execution cycles depending on the input data. In the measurement of the execution cycles, the actual behavior of these SW implementated basic operations were simulated. As a result, the execution cycle estimation reported by IMSP- 2P will contain some error as in the case of IMSP-2 due to the execution cycles dependent on input data (up to 25% and 32% as shown in Figures 2 and 3, respectively.)

Some of the design results for IMC are shown in Table 4. In this table, the second column represents area constraints A_max in Kgates. The third column represents the estimated execution cycles ($T(X)$) by the IMSP-2P solver. The fourth column shows the execution cycle errors (Err) measured by using the simulator generated by the PEAS-I system, assuming that the HW interlock is used. Note that estimation errors are almost 0% for designs with all HW implemented operations. The last column shows the HW modules that implement basic operations to be implemented by HW. The optimum instruction set is then defined on the selected HW modules. For example, in design #7 with *A_max* = 20 Kgates only the modules 'b_alsft' and 'div_seq' have been chosen together with the kernel, therefore the instruction set of the designed ASIP contains the primitive operations and those basic operations belonging to 'b_alsft' and 'div_seq', i.e. **ashr**, **ashl**, **lshr**, **lshl**, **div**, **udiv**, **mod**, and **umod**. The remaining basic operations such as **mul**, **extendhi**, and so on are implemented in SW using the Kernel. For any given area constraint, the shown partitioning represents the optimum one. Other combinations were not selected by the algorithm because of their inferiority. The design results for the ESS and *diffeq* programs are not shown here due to lack of space.

Table 4 : Estimated execution cycles, estimation errors and selected HW modules for IMC by IMSP-2P

#	*A_max Kgates*	*T(X) Cycles*	*Err (%)*	*Selected HW Modules (with kernel)*
1	55	60689	0.5	b_alsft extend mul_csa div_2seq_p3
2	40	63068	0.1	b_alsft extend mul_3clk div_2seq_p6
3	35	64817	0.1	b_alsft extend mul_3clk div_2seq_p9
4	30	71400	0.0	b_alsft extend div_2seq mul_csa
5	25	84329	0.0	b_alsft extend div_2seq mul_bpr
6	22	112396	1.7	b_alsft div_seq mul_seq
7	20	164296	27.9	b_alsft div_seq
8	19	205096	31.6	extend div_seq
9	18	344448	9.1	b_alsft extend
10	16	353088	8.4	b_alsft
11	15	402528	4.1	(kernel *only*)

Another major feature of IMSP-2P is that it can reduce the pipeline execution cycles by optimally selecting the pipelined FUs in comparing to the IMSP-2 solver. In experiments, the performance (via the execution cycles of the application program) of the designed ASIPs was measured by using the PEAS-I simulator. It is found that up to 4.2%, 8.4%, and 6.7% execution cycle reduction rates due to selecting the pipelined FUs by IMSP-2P compared to IMSP-2 were achieved for the ESS, IMC, and *diffeq* examples, respectively.

6 Conclusion and Future Work

We have proposed an efficient method to design an optimal pipelined instruction set processor in the PEAS-I system. The method uses the pipeline scheduler which is capable of detecting and resolving pipeline hazards and handling multicycle operations simultaneously. The method with IMSP-2P is introduced as a HW/SW partitioning problem and selects the implementation method of the operations that implement a pipelined ASIP instruction set so that the performance is maximized under the given design constraints. The effectiveness of the IMSP-2P algorithm was demonstrated through design examples. The method estimates the performance of a designed pipelined ASIP accurately for most designs. Considerable execution cycle reduction rates due to selecting pipelined FUs have been obtained. The algorithm is so efficient that all the optimal solutions performed in the experiments in partitioning process were obtained within a few seconds on a conventional workstation.

The estimation error reduction for SW implemented operations is left for future work. Our future work also includes the development of a HW/SW partitioning algorithm for pipelined ASIP design with the least gate count under a given power consumption and execution cycle constraints. Moreover, the design with the lowest power consumption under gate count and execution cycle constraints is also planned.

Acknowledgments

Authors would like to express their thanks to NTT Communication Science Laboratories, VLSI Technology, Inc., Science Create, Co. Ltd., Japan, for their kind help and assistance. Part of this research is supported in part by a research grant from the Ministry of Education, Science and Culture of Japan under contracts No's. 07558038 and 07680353.

References

[1] Hennessy, J.L., and Patterson, D.A., *Computer Architecture: A Quantitative Approach,* Morgan Kaufmann Publishers, 1990.

[2] Alomary, A., Nakata, T., Honma, Y., Sato, J., Hikichi, N., and Imai, M., "PEAS-I: A Hardware/Software Codesign System for ASIPs," *Proc. of EURO-DAC'93*, pp. 2 – 7, 1993.

[3] Imai, M., Alomary, A., Sato, J., and Hikichi, N., "An Integer Programming Approach to Instruction Implementation Method Selection Problem," *Proc. of EURO-DAC'92,* pp. 106 – 111, 1992.

[4] Alomary, A., Nakata, T., Honma, Y., Imai, M., and Hikichi, N., " An ASIP Instruction set Optimization Algorithm with Functional Module Sharing Constraint," *Proc. of ICCAD-93*, pp. 526 – 532, 1993.

[5] Huang, I-J., and Despain, A.M., "Synthesis of Instruction Sets for Pipelined Microprocessors,"*Proc. of DAC'94*, pp.5 – 11, 1994.

[6] Stallman, R., *Using and Porting GNU C Compiler,* Free Software Foundation, Version 1.40, 1991.

[7] Binh, N.N., Imai, M., Shiomi, A., Sato, J., and Hikichi, N., "A Pipeline Scheduling Algorithm for Instruction Set Processor Design Optimization," *Proc. of APCHDL'94*, Toyohashi, Japan, pp. 59 – 66, Oct. 1994.

[8] Paulin, P.G., Knight, J.P., and Girczyc, E.F., "HAL: A Multi-paradigm Approach to Automatic Data Path Synthesis," *Proc. of DAC'23*, pp. 263 – 270, 1986.

[9] Nakamura, Y., Oguri, K., and Nagoya, A., "Synthesis from Pure Behavioral Descriptions," *in High-Level VLSI Synthesis*, Camposano, R., and Wolf, W., eds, pp. 205 – 229, Kluwer Academic Publishers, 1991.