A New HW/SW Partitioning Algorithm for Synthesizing the Highest Performance Pipelined ASIPs with Multiple Identical FUs

Nguyễn Ngọc Bình*†, Masaharu Imai*†, and Akichika Shiomi*‡

† Dept. of Information & Computer Sciences
Faculty of Engineering Science
Osaka University
1-3 Machikaneyama-cho
Toyonaka-shi, Osaka
Japan 560
E-mail: {binh,imai}@ics.es.osaka-u.ac.jp

‡ Dept. of Computer Science
Faculty of Information
Shizuoka University
3-5-1 Johoku-cho
Hamamatsu-shi, Shizuoka
Japan 432
E-mail: shiomi@cs.inf.shizuoka.ac.jp

Abstract

This paper introduces a new HW/SW partitioning algorithm for automatic synthesis of a pipelined CPU architecture with multiple identical functional units (MIFUs) of each type in designing ASIPs (Application Specific Integrated Processors). The partitioning problem is formalized as a combinatorial optimization problem that partitions the operations into hardware and software so that the performance of the designed ASIP is maximized under given gate count and power consumption constraints, regarding the optimal selection of needed FUs of each type. A branch-and-bound algorithm with proposed lower bound function is used to solve the formalized problem. The experimental results show that the proposed algorithm is found to be effective and efficient.

1 Introduction

Embedded systems [1] implement dedicated functions such as control of anti-blocking brakes, the instrumentation and control of an assembly line or compression and encoding of audio, video and data in a multi-media system. Thus, the function is well defined in advance and the embedded system is installed once. An embedded system is implemented partly in hardware (HW) and partly in software (SW) by using optimization tools for the HW design, codesign of HW and SW, and design of embedded SW. Design of embedded systems has been developed for several years, but the tight integration of HW and SW design and some particular design problems is one of the new issues. Especially, an Application Specific Integrated Processor (ASIP) is a dedicated microprocessor that is designed putting a special application field in mind. It contains a CPU core, memory (RAM, ROM), and peripheral circuits as shown in Figure 1. ASIP is used in embedded systems where the performance, hardware cost, and power consumption are important factors.

In the traditional embedded system design, system architects decide which operations will be implemented in HW or SW. In order to produce an efficient result in reasonable design time, an efficient HW/SW codesign partitioning method should be used. Many HW/SW partitioning methods have been proposed. Gupta and De Micheli [2] introduced a method that moves operations from HW to SW to meet performance constraint at minimal cost. Ernst, et al. [3] take the opposite approach moving time critical operations from SW to HW. Woo, et al. [4] introduced a codesign method that divides the operations into HW, SW and codesign groups. Then the designer manually investigates the HW/SW tradeoff by distributing the implementation of the codesign operations between HW and SW.

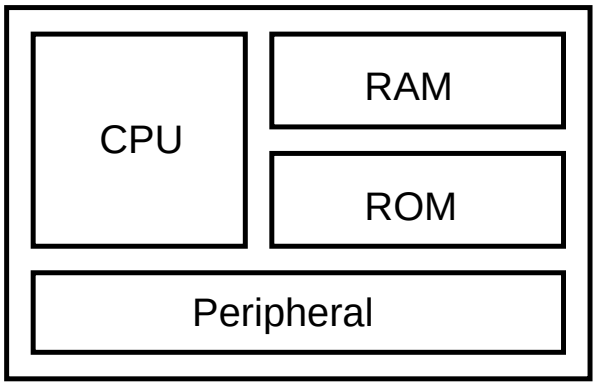

Figure 1: ASIP components.

The ASIP design optimization problems can be classified into 3 classes: (1) highest performance design, (2) least HW cost (gate count) design, and (3) lowest power consumption design. The HW/SW codesign problem addressed in this paper is related to the highest performance pipelined ASIP design with gate count and power consumption constraints. Our approach differs from the above mentioned traditional methods in automating HW/SW partitioning at operator level to get an optimal design of the ASIP.

A HW/SW codesign system PEAS-I (Practical Environment for ASIP development - type I) [5] employs a formal

* The authors were formerly with the Department of Information and Computer Sciences, Toyohashi University of Technology, Toyohashi-shi, Japan 441.

This research is supported in part by Grant-in-Aid for Scientific Research Nos. 07558038 and 07680353 from the Ministry of Education, Science and Culture, Japan.

EURO-DAC '96 with EURO-VHDL '96
0-89791-848-7/96 $4.00

method to synthesize an optimal instruction set processor by solving Instruction set implementation Method Selection Problems (IMSP) types 1, 2 and 3. IMSP-1 [6] is set up assuming no interaction among the operations, and each operation was to be implemented using a separate HW module. On the other hand, IMSP-2 [7] is an extension of IMSP-1 by taking resource sharing into account. While IMSP-1, 2 are for designing the highest performance ASIPs, IMSP-3 [8] yields the design of ASIP with the least HW cost subjecting to execution cycle and power consumption constraints.

The main part of PEAS-I is the architecture information generator, which automatically performs the optimal HW/SW partitioning to decide the best CPU core architecture under given constraints, as well as define an optimal instruction set for the selected architecture. Once the HW/SW partitioning has been performed, the HW portion will be generated in HDL, that can be accepted as input to the high level synthesis tool PARTHENON [9]. The SW portion will be compiled into the code of the selected instruction set by a C compiler and simulator generated by the application program development tool automatically. In other words, PEAS-I accepts an application program with input data and will generate the optimal ASIP (CPU core, instruction set, and software tools such as C compiler and simulator) automatically by using the IMSP solvers for HW/SW partitioning. These features make PEAS-I differ from other HW/SW codesign systems such as ASIA [10] and CASTLE [11], where the HW/SW partitioning is done manually in these systems.

2 Pipelined CPU Architecture with MIFUs

The target CPU to be generated by PEAS-I belongs to a class of Harvard architecture with separate data bus and instruction bus. The minimum part in the PEAS-I CPU core architecture is 'Kernel' which consists of an ALU, a one-bit shifter, and a register file. The CPU core may include other functional units (FUs) such as multiplier, divider, and so on. Moreover, while the CPU may contain the Kernel and different types of FUs, it is assumed that there can be multiple identical FUs (MIFUs) as shown in Fig. 2. Note that so far the PEAS CPU cores were assumed to have no more than one FU of each type.

The pipelined architecture synthesized by PEAS-I consists of four stages: (1) IF (Instruction Fetch and decode), (2) EX (EXecution), (3) MEM (MEMory access), and (4) 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 is with a RISC type of the load/store architecture and each control step corresponds to one clock cycle. The architecture has a register bypass to forward the computation results to Kernel or FUs. Each FU can be multi cycle and pipelined. For the former CPU architecture, we have developed IMSP-2P (IMSP-2 for Pipeline) [12, 13] for the high performance ASIP design, and IMSP-3P [14] for the least gate count ASIP design.

In this paper we deal with IMSP-2P for the presented CPU core with MIFUs and we call it as IMSP-2P-MIFU, which is an abbreviation of "Instruction set implementation Method Selection Problem type 2 for Pipelined architecture with Multiple Identical Functional Units of each type." In the following sections we define the IMSP-2P-MIFU HW/SW partitioning problem, its formalization with a proposed algorithm, and describe the experimental results.

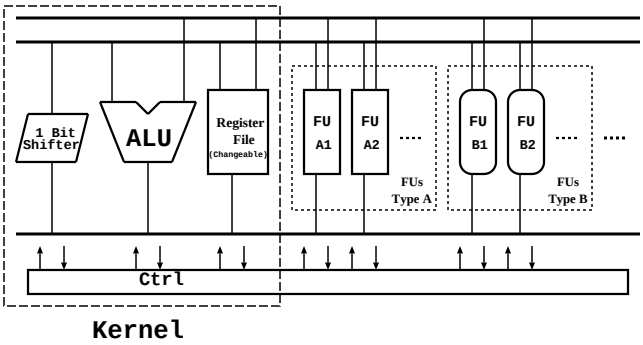

Figure 2: PEAS CPU core architecture with multiple identical FUs.

3 Partitioning Problem Formalization

The instruction set architecture of the designed ASIP is based on the GNU C Compiler (GCC) abstract machine model [15]. The reason behind that choice is as follows: (1) the generated ASIP is assumed to execute application program written in C language, and (2) GCC is a widely accepted public-domain software that generates an efficient object code. The set of operations and functions that can be generated by GCC is chosen to be the set of candidate instructions that can be included in the designed ASIP. 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. The basic operations contain other C operators and functions that are not primitive operations and can be implemented using some HW choices (such as fast or slow HW modules) or using SW subroutine that uses primitive operations. Please refer to Refs. [8, 12] for the primitive and basic operations.

The IMSP-2P-MIFU HW/SW partitioning problem is defined as follows:

"Select the implementation methods of the basic operations, among HW choices and SW, and define the needed number of identical FUs of each type so that the performance of the designed pipelined ASIP is maximized under given gate count and power consumption constraints, taking the functional module sharing into account."

We formalize this problem as a combinatorial optimization problem, which can be solved using an algorithm based on the branch-and-bound method.

3.1 Definitions and Notations

In the rest of this paper, the following definitions and notations are used:

(1) The "implementation method" refers to any of HW or SW implementations of an operation. For any operation there might be many HW implementations such as fast or slow HW modules.

(2) "n" denotes the total number of different basic operations (or number of basic operation types) to be considered.

(3) "f_i" denotes the execution frequency of operation #i in the given set of application programs, where $0 \leq i \leq n$. f_0 is for execution frequency of all primitive operations.

(4) "M" denotes the whole set of implementation methods that realize all operations.

(5) "M_i" denotes the set of implementation methods which realize operation #i, where $M_i \subseteq M$, and $1 \leq i \leq n$.

(6) "x_i" denotes an implementation method that realizes operation #i, where $x_i \in M_i$, $1 \leq i \leq n$.

(7) "$k_i(x_i)$" denotes the number of identical FUs for operation #i when implemented by method x_i, where $1 \leq i \leq n$. Then $X = (x_1, x_2, \ldots, x_n)$ and $K = (k_1(x_1), k_2(x_2), \ldots, k_n(x_n))$ represent an architecture configuration of the PEAS CPU with MIFUs. (In fact, K depends on X, i.e. $K = K(X)$.)

(8) "k_i^{max}" denotes the maximal number of identical FUs for operation #i.

(9) When $M_i \cap M_j \neq \phi$ ($i \neq j$), and if $M_i \cap M_j$ contains a functional module x, then x can be used to implement operations #i and #j simultaneously.

(10) "S" represents the set of selected implementation methods of the whole basic operations. That is, $S = \bigcup_{i=1}^{n}\{x_i\}$. Generally, $|S| \leq n$. When one or more functional module(s) is/are shared, $| S | < n$, otherwise $| S | = n$.

(11) "$t_i(x_i)$" denotes the execution cycle of operation #i when implemented by method x_i, where $1 \leq i \leq n$.

(12) "$a(x_i)$" and "$p(x_i)$" denote the area and power consumption required for implementation method x_i, respectively, where $1 \leq i \leq n$. "$a(x_0)$" and "$p(x_0)$" are for the Kernel only.

(13) "A_max" and "P_max" denote the maximum allowable gate count and the maximum allowable power consumption, respectively, for the selected functional modules in the ASIP chip.

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

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

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

(17) "c_j" denotes clock cycles needed to define control (e.g., branch delay) from block B_j to another one, where $1 \leq j \leq N$.

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

3.2 Problem Formalization

The IMSP-2P-MIFU HW/SW partitioning problem can be formalized as a combinatorial optimization problem as follows:

Find solution vectors

$$\begin{aligned} X &= (x_1,\ x_2, \cdots,\ x_n) \\ K &= (k_1(x_1),\ k_2(x_2), \cdots,\ k_n(x_n)) \end{aligned}$$

which minimize the objective function:

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

subject to:

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

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

and

$$0 \leq k_i(x_i) \ \leq \ k_i^{max} \ \ (1 \leq i \leq n). \qquad (4)$$

The key point in computing $T(X, K)$ in Eq.(1) is to obtain the value of $t(B_j, X, K)$. We have developed a HW/SW partitioning-oriented pipeline scheduling algorithm to estimate $t(B_j, X, K)$ for basic block B_j under configuration X and K, which is an extension of a scheduling algorithm [16]. Note that the algorithm in Ref. [16] has a limitation where $k_i^{max} = 1 \ \ (i = 1, \cdots, n)$.

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. We have extended the scheduler for the general case with any vectors X, K. In the solution vector K, a component $k_i(x_i) = 0$ (for some i) means none of the FUs of type i has been chosen and the basic operation of type i is implemented by SW.

Note that the problem defined by Eqs.(1)–(4) under the condition $k_i^{max} = 1 \ \ (i = 1, \cdots, n)$ is the same problem defined in Ref. [12] for IMSP-2P. That is, we here deal with a more general case.

4 IMSP-2P-MIFU Solver

4.1 Input and Output

The input to the IMSP-2P-MIFU solver includes the following items:
(i) The GCC's RTL code of the given application program,
(ii) F_j's for $j = 1, \cdots, N$,
(iii) b (#execution cycles reduced by un-taken branches),
(iv) constraint parameters A_max and P_max, and
(v) 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-MIFU solver includes the optimum implementation method of each basic operation, the number of MIFUs of each type, and pipelined schedules of BBs. 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 automatically integrates the functional module sharing among basic operations into one HW module whenever possible.

4.2 Algorithm

Because IMSP-2P-MIFU as well as IMSP-2P is NP-hard, and in order to solve it in reasonable computation time, the branch-and-bound method is used. The branch-and-bound method is known as one of the most effective methods to solve combinatorial optimization problems. The branch-and-bound algorithm is a tree searching technique, in which the process of searching an optimum instruction set is viewed by

the algorithm as finding a leaf node in a search tree. One of the most important issues in 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-MIFU solver is the same as in the IMSP-2P as follows:

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

$$u_i(x_i) = \begin{cases} 1, & \text{if } x_i \text{ is a HW implementation,} \\ t_i(x_i), & \text{if } x_i \text{ is a SW implementation,} \end{cases} \quad (6)$$

$$Stall_{fast} = T(X_{fast}, K_1) - \sum_{i=0}^{n} f_i \ , \quad (7)$$

where the parameter d represents the depth of the node under consideration, $Stall_{fast}$ is the number of pipeline stalls in executing the given application program using the hypothetical FUs of one cycle denoted by X_{fast}, and $T(X_{fast}, K_1)$ is computed by using Eq.(1), and K_1 is with $k_i^{max} = 1$ ($i = 1, \cdots, n$). Please refer to Ref. [12] for derivations of Eqs. (5)–(7). Note that the module sharing capability and heuristic reordering are the same as in the IMSP-2, IMSP-2P solvers.

4.3 Design Space Exploration

Besides designing a good lower bound function, it is necessary to have a strategy to exploit the design space efficiently. As mentioned in Ref. [12], IMSP-2P is with the design space of up to 1.5×10^8 nodes for 14 basic operation types. In the case of IMSP-2P-MIFU, the number of nodes in the design space becomes $\prod_{i=1}^{n} k_i^{max}$ times larger than those of the former design space of IMSP-2P. This is because when operation #i is considered to be implemented in HW by method x_i, the algorithm investigates the necessary number of identical FUs chosen currently by x_i on the values from 1 to k_i^{max} incorporating with other basic operations' implementation methods to find the best K. For example, when $n = 7$, $k_i^{max} = 4$ (for all i), the IMSP-2P-MIFU solver could be $4^7 = 16484$ times slower than the IMSP-2P solver. However, we can reduce the number of searches by re-defining the values of k_i^{max}'s using the information on the currently selected FUs, the design constraints, and the application program's RTL code.

Assuming that on the current depth of the search an FU with delay D cycles and latency L cycles ($L = D$ for non-pipelined FUs) for operation #i is chosen, we can note that the number of identical FUs should not exceed L (we use $L(x_i)$ to describe the latency of FU for x_i) because the operations are executed in the pipeline manner and pipeline hazards such as structural hazards must be eliminated. Moreover, it should also not be larger than the maximum number of the basic operations of type i within each basic block, i.e., less than or equal to

$$m_i = \max_{1 \le j \le N} \{\text{no. of operations \#}i \text{ in } B_j\}. \quad (8)$$

On the other hand, the maximum number of identical FUs for operation #i can be chosen when all other operations are supposed to be implemented in SW, then the sums of gate counts and power consumption of these identical FUs and those of the Kernel must be satisfied the design constraints in Eqs.(2) and (3). Note that every implementation method includes the Kernel as the minimum HW. Summarizing and denoting the number of identical FUs for operation #i for the current status (with the implementation method x_i) as κ_i^{max} we have the following estimation:

$$\kappa_i^{max} = \min\{k_i^{max},\ L(x_i),\ m_i,\ \lfloor \tfrac{A\_max - a(x_0)}{a(x_i) - a(x_0)} \rfloor,\ \lfloor \tfrac{P\_max - p(x_0)}{p(x_i) - p(x_0)} \rfloor\} \quad (9)$$

where $a(x_0)$ and $p(x_0)$ are gate count and power consumption of the Kernel, respectively. Then, the IMSP-2P-MIFU solver uses

$$0 \le k_i(x_i) \ \le \ \kappa_i^{max} \text{ (for all } i) \quad (10)$$

instead of Eq.(4). Clearly, $\kappa_i^{max} \le k_i^{max}$. In many cases κ_i^{max} becomes 1 when FU is with $D = 1$ or large gate count as well as power consumption, thus no more than one FU of that type can be chosen. At the beginning of the algorithm, the solvers check for the given design constraints to ensure their correctness (e.g. at least the design must contain the Kernel, and so on). Note that Eq.(9) is applied to the HW choice of operation #i only, i.e., when $a(x_0) < a(x_i) \le A\_max$ and $p(x_0) < p(x_i) \le P\_max$ hold. At the beginning of the algorithm, a large value (e.g. 9999) is given to k_i^{max}, then the IMSP-2P-MIFU will define the best selection of identical FUs automatically.

5 Experiments and Results

This section describes the effectiveness and efficiency of the proposed algorithm.

The experimental conditions are the same as in Ref. [12] with a HW/SW module information database of both pipelined and non-pipelined FUs as well as SW subroutines, and the following sample programs: ESS (Equation System Solver program, which solves a system of two linear equations using Cramer's rule), IMC (Inverse Matrix Calculator program that computes the inverse of a non-singular 3×3 matrix using Cramer's rule), and *diffeq* (a program for solving a second order differential equation from Ref. [17]).

These sample programs with associated input data were fed to APA (Application Program Analyzer) of the PEAS-I system to obtain the execution frequencies of basic operation, basic blocks, etc. The analyzed results are shown in Refs. [12, 13]. The code optimization was performed by the GNU C Compiler [15], the pipeline scheduling was performed by the scheduler described in Ref. [16].

5.1 Effectiveness

Using the analyzed information from the given application program, the IMSP-2P-MIFU algorithm accordingly selected the optimum partitioning for different values of A_max and P_max. This capability is the same as of the IMSP-2P [12]. The power consumption was ignored to simplify the experimental cases by giving a large value to P_max. Because there are one-cycle FUs (a multiplier **mul_csa**, an extender **extend**, and a shifter **b_alsft**) in the database (i.e. $D = L = 1$), the solver never selects more than one of such FUs because of Eq.(9). The number of selected identical dividers varies depending on the constraints. Table 1 represents part

Table 1: Selection of identical FUs by IMSP-2P-MIFU with k_i^{max} = 4 (for all i) for IMC.

A_max (Kgates)	Gate Count	Exe. Cycles	(# identical FUs) Module name
150	110939	65702	(3)div_2seq_p3
110	85554	65732	(4)div_2seq_p6
85	81812	65762	(2)div_2seq_p3
81	70055	65792	(3)div_2seq_p6
70	66534	65822	(4)div_2seq_p9
52	45046	66242	(2)div_2seq_p9
45	40982	67033	(3)div_2seq
40	39353	67813	(3)div_2seq
39	35174	67873	(2)div_2seq
35	33545	68473	(2)div_2seq
27	26405	84879	(2)mul_seq
26	25764	85209	(2)div_seq

Table 2: Selection of identical FUs by IMSP-2P-MIFU with k_i^{max} = 4 (for all i) and without multipliers of $D = L < 4$ in HW database for IMC.

A_max (Kgates)	Gate Count	(# identical FUs) Mudule name
150	116911	(4)div_2seq_p6 (2)mul_bpr_p2
116	104512	(2)div_2seq_p3 (3)mul_bpr_p4
104	89234	(4)div_2seq_p9 (3)mul_bpr_p4
89	78490	(3)div_2seq_p9 (3)mul_bpr_p4
75	68409	(2)div_2seq_p6 (4)mul_bpr_p8
65	64243	(3)div_2seq_p9 (3)mul_bpr_p8
64	63009	(2)div_2seq_p6 (3)mul_bpr_p8
56	53499	(2)div_2seq_p9 (3)mul_bpr_p8
52	48099	(2)div_2seq_p9 (2)mul_bpr_p8
46	44035	(3)div_2seq (2)mul_bpr_p8
44	43627	(2)div_2seq (3)mul_bpr_p8
42	38227	(2)div_2seq (2)mul_bpr_p8
37	36910	(2)div_2seq (3)mul_bpr

of the experimental results for IMC with k_i^{max} = 4 (for all i). To show the effectiveness of the IMSP-2P-MIFU for other type of FUs, we removed multipliers with $D = L < 4$ from the database, then ran IMSP-2P and IMSP-2P-MIFU with k_i^{max} = 4 (for all i). Some of the results are shown in Tab. 2. Note that FUs with the number of identical FUs equal to 1 are not shown in Tabs. 1 and 2. Please refer to Ref. [12] for the names and characteristics of the modules. Figures 3 – 5 show the area (gate count) vs. execution cycle tradeoff as well as the performance improvement by IMSP-2P-MIFU in comparing to IMSP-2P for ESS, IMC, and *diffeq*, respectively. Using the IMSP-2P-MIFU, we can select the pipelined ASIPs with MIFUs with higher performance of about 7% compared to IMSP-2P for these sample programs. It was found that there are at most 6 operations of the same type (multiplication/division) in each basic block for these sample programs. For application programs with more operations of the same type the better improvement can be expected.

5.2 Efficiency

The IMSP-2P-MIFU with the proposed algorithm is so efficient that the optimal pipelined ASIP as well as its instruction set can be selected within several seconds on a PC with Pentium P54-120 (UNIX FreeBSD 2.0.5R) for any constraint. The average number of visited nodes in the search tree is about 248, 404, and 332 with the average time of 3 secs, 7 secs, and 4 secs for ESS, IMC, and *diffeq*, respectively.

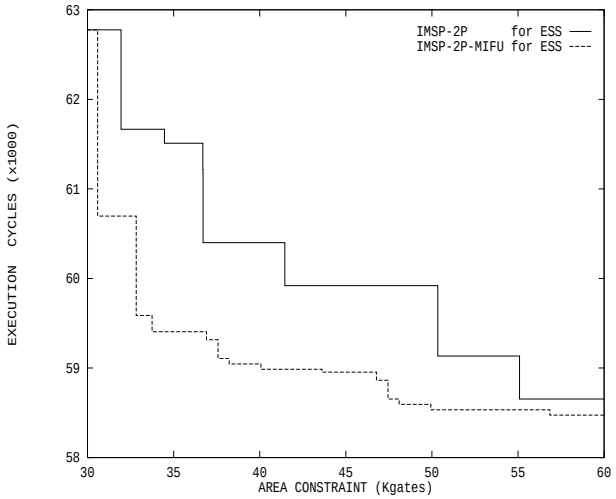

Figure 3: Area vs. execution cycle tradeoff for ESS.

6 Conclusion and Future Work

In this paper an effective and efficient HW/SW codesign partitioning algorithm for designing the highest performance pipelined ASIPs with MIFUs of each type under given gate count and power consumption constraints has been introduced. A branch-and-bound algorithm was used to solve the presented partitioning problem and was implemented in C language. A good lower bound function for the algorithm has been presented. According to the experimental results, the proposed algorithm is found to be able to solve a considerable size partitioning problem in a reasonable computation time (typically within several seconds on a PC with Pentium P54-120). The effectiveness and efficiency of the algorithm have been demonstrated through performing a set of sample programs. The proposed algorithm is able to improve the performance of pipelined ASIPs, especially for application programs with many operations of the same type in a basic block.

However, there are following limitations in this work: (1) the cost of other components of CPU such as RAM, ROM, register file, peripheral circuits (ASICs) was not addressed; (2) data and instruction bus width was not addressed; (3) the power consumption is still a crude approximation. Switching activity of the HW modules and the power consumption in case of the SW implementations should be taken into account, e.g., by using the technique proposed by V. Tiwari [18]; (4) the sample application programs are mainly computational and small. Larger application programs, e.g. DSP programs, should be fed to the PEAS-I system; etc.

Our further research needs to investigate these issues. The development of an IMSP-3P-MIFU HW/SW partitioning algorithm (as an extension of the IMSP-3P [14]) for designing

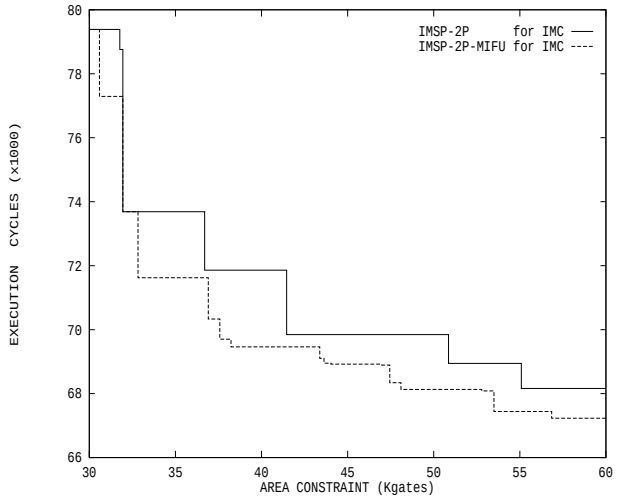

Figure 4: Area vs. execution cycle tradeoff for IMC.

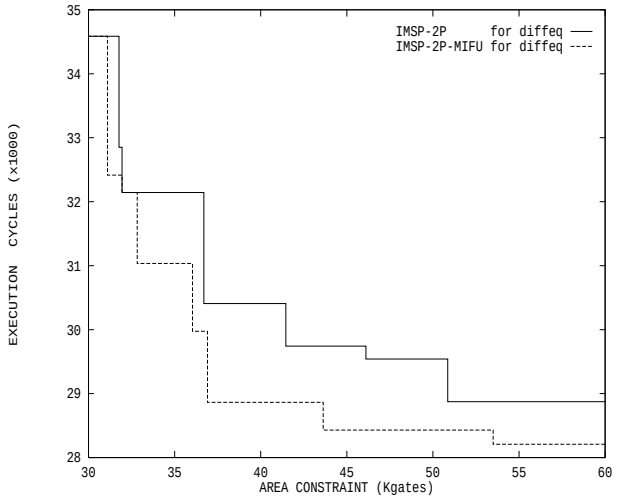

Figure 5: Area vs. execution cycle tradeoff for *diffeq*.

a least gate count pipelined ASIP with MIFUs under power consumption and execution cycle constraints is also planned.

References

[1] R. Camposano and J. Wilberg: "Embedded System Design," *in Design Automation for Embedded System*, R. Camposano and W. Wolf, eds, vol. 1, nos.1–2, pp. 5 – 50, Kluwer Academic Publishers, Jan. 1996.

[2] R. Gupta and G. De Micheli: "Hardware-Software Cosynthesis for Digital Systems," *IEEE Design & Test*, pp. 29 – 41, Sep. 1993.

[3] R. Ernst, J. Henkel, and T. Benner: "Hardware-Software Cosynthesis for Microcontroller," *IEEE Design & Test*, pp. 64 – 75, Sep. 1993.

[4] N. Woo, A. Dunlop, and W. Wolf: "Codesign from Cospecification," *Computer*, pp. 42 – 47, Jan. 1994.

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

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

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

[8] A. Alomary, T. Nakata, Y. Honma, A. Shiomi, M. Imai, and N. Hikichi: "An ASIP Instruction Set Optimization Algorithm with Execution Cycle Constraint," *Proc. of the 4th Synthesis And SImulation Meeting and international Interchange (SASIMI'93)*, pp. 34 – 43, Nara, Japan, Oct. 1993.

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

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

[11] J. Wilberg, et al.: "Design Flow for Hardware/Software Cosynthesis of a Video Compression System," *Proc. of Codes/CASHE '94*, Grenoble, France, 1994.

[12] N.N. Binh, M. Imai, A. Shiomi, and N. Hikichi: "A Hardware/Software Partitioning Algorithm for Pipelined Instruction Set Processor," *Proc. of EURO-DAC'95*, pp. 176 – 181, Brighton, U.K., Sep. 1995.

[13] N.N. Binh, M. Imai, A. Shiomi, and N. Hikichi: "A Hardware/Software Codesign Method for Pipelined Instruction Set Processor Using Adaptive Database," *Proc. of ASP-DAC'95*, pp. 81 – 86, Chiba, Japan, Aug. 1995.

[14] N.N. Bình, M. Imai, A. Shiomi, and N. Hikichi: "A Hardware/Software Partitioning Algorithm for Designing Pipelined ASIPs with Least Gate Counts," *Proc. of DAC'96*, pp. 527 – 532, Las Vegas, USA, Jun. 1996.

[15] R. Stallman: *Using and Porting GNU CC*, Free Software Foundation, Version 1.40, 1991.

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

[17] P.G. Paulin, J.P. Knight, and E.F. Girczyc: "HAL: A Multiparadigm Approach to Automatic Data Path Synthesis," *Proc. of DAC'86*, pp. 263 – 270, 1986.

[18] V. Tiwari, S. Malik, and A. Wolfe: "Power Analysis of Embedded Software: A First Step Towards Software Power Minimization," *IEEE Trans. VLSI*, vol.2, no.4, pp. 437 – 445, Dec. 1994.