A Hardware/Software Partitioning Algorithm for Designing Pipelined ASIPs with Least Gate Counts

Nguyen Ngoc Bình*, Masaharu Imai*, Akichika Shiomi†, and Nobuyuki Hikichi‡

Dept. of Information & Computer Sciences
Toyohashi University of Technology
Toyohashi, 441 Japan

‡ Dept. of Software Technology
Software Research Associates, Inc.
Tokyo, 170 Japan

Abstract — This paper introduces a new HW/SW partitioning algorithm used in automating the instruction set processor design for pipelined ASIP (Application Specific Integrated Processor). The partitioning problem is formalized as a combinatorial optimization problem that partitions the operations into hardware and software so that the HW cost (gate count) of the designed pipelined ASIP is minimized under given execution cycle and power consumption constraints. A branch-and-bound algorithm with proposed lower bound functions is used to solve the presented formalization in the PEAS-I system. The experimental results show that the proposed method is found to be effective and efficient.

1 Introduction

As a type of embedded systems, 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 (ROM, RAM), and peripheral circuits. In the traditional ASIP design, system architects decide which operations will be implemented in hardware (HW) or software (SW). In order to produce an efficient design in reasonable time, an efficient HW/SW codesign partitioning method should be used. Many HW/SW partitioning methods have been proposed. Gupta and De Micheli [1] introduced a method that moves operations from hardware to software to meet performance deadline at minimal cost. Ernst et al. [2] take the opposite approach moving time critical operations from software to hardware. Woo et al. [3] introduced a codesign method that divides the operations into hardware, software and codesign groups. Then designer manually investigates the HW/SW tradeoff by distributing the implementation of the codesign operations between hardware and software. In these conventional codesign methods, there is no formal method used to obtain the optimal design. Moreover, the design process still largely depends on the designer's skill, and the design is achieved manually after investigating many design candidates.

In the ASIP design, there are three important factors: the performance, HW resources, and power consumption. Therefore, problems of the ASIP design can be classified into 3 classes: highest performance design, least HW cost (gate count) design, and lowest power consumption design. The HW/SW codesign problem addressed in this paper relates to the least gate count design with execution cycle and power consumption constraints. A formal HW/SW partitioning method for ASIP design is introduced. The method is based on a combinatorial optimization technique that decides which operations to be implemented in hardware or in software so that the ASIP gate counts are minimized under given execution cycle and power consumption constraints. An algorithm based on the branch-and-bound method is used to solve the presented optimization problem.

Besides the mentioned works [1, 2, 3], there are also other HW/SW codesign systems such as CASTLE [4] and ASIA [5] for the ASIP development. However, the HW/SW partitioning is also done manually in these systems. Performing the automatic HW/SW partitioning to find an optimal CPU core is one of the distinguished features of the method proposed in this paper compared to the above HW/SW codesign systems.

In the following sections, we introduce the HW/SW partitioning problem and its formalization, then show its implementation with the experimental results in the PEAS-I system [6].

2 HW/SW Partitioning Problem

In the ASIP design, we are given an application program with associated input data, and tools of a design environment such as an analyzer, C compiler, and so on. Assuming that the application program with its input data is analyzed to recognize the utilized operations with their execution frequencies. An ASIP must contain a minimum HW, called 'Kernel', which can make it to work with a minimum set of so-called **Primitive** operations. Other operations are called **Basic** operations (shortly, operations), which will be implemented in HW or SW. Also, assuming that there are a hardware module database and an implementation method database in the design environment. An implementation method of an operation can be a software module (run-time subroutine) or a set of HW modules, which can be shared among operations. We are to design a least gate count pipelined ASIP under given execution cycle and power consumption constraints. The HW/SW partitioning problem in this case is as follows:

*The authors are currently with the Department of Information and Computer Sciences, Faculty of Engineering Science, Osaka University, Toyonaka-shi, Osaka, 560 Japan.

†The author is currently with the Department of Computer Science, Faculty of Information, Shizuoka University, Hamamatsu-shi, 432 Japan.

33rd Design Automation Conference ®

Permission to make digital/hard copy of all or part of this work for personal or class-room use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DAC 96 - 06/96 Las Vegas, NV, USA
©1996 ACM, Inc. 0-89791-833-9/96/0006.. $3.50

"Select the implementation methods of operations, among hardware choices and software, so that the gate count of the designed pipelined ASIP is minimized under given execution cycle and power consumption constraints, taking HW resource sharing among operations."

2.1 Definitions and Notations

In order to formalize the above problem, following definitions and notations are used:
(1) The "implementation method" refers to any of hardware, or software implementations of an operation. For any operation there might be many hardware implementations such as fast or slow hardware modules.
(2) "n" denotes the total number of different operations to be considered.
(3) "f_i" denotes the execution frequency of operation #i in the given application program, where $1 \leq i \leq n$. f_0 is for the sum of execution frequencies 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 $x_i \in M_i$, $1 \leq i \leq n$. x_0 denotes the Kernel as minimum HW including an ALU, a 1-bit shifter, and a register file. Then $X = (x_1, x_2, \ldots, x_n)$ is as an architecture configuration.
(7) 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.
(8) "S" represents the set of selected implementation methods of the whole basic operations, and defines as $S = \bigcup_{i=1}^{n} \{x_i\}$. Note that $|S| \leq n$. When one or more functional module(s) is/are shared, $\mid S \mid < n$, otherwise $\mid S \mid = n$.
(9) "$t_i(x_i)$" denotes the execution cycles of operation #i when implemented by method x_i, where $1 \leq i \leq n$.
(10) "$a(x_i)$" and "$p(x_i)$" denote the area and power consumption required for implementation method x_i, respectively, where $1 \leq i \leq n$.
(11) "T_max" and "P_max" denote the maximum allowable execution cycles and the maximum allowable power consumption, respectively, for the selected functional modules in the ASIP chip.
(12) "N" denotes the total number of basic blocks in the application program's code.
(13) "$t(B_j, X)$" denotes the execution cycles needed to execute basic block B_j using a combination of implementation methods X, where $1 \leq j \leq N$.
(14) "F_j" denotes the execution frequency of basic block B_j in the given application program, where $1 \leq j \leq N$.
(15) "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.
(16) "b" denotes execution cycles reduced by un-taken branches in execution of the given application program.

2.2 Problem Formalization

The hardware/software partitioning problem can be formalized as a combinatorial optimization problem as follows: Find a solution vector

$$X = (x_1, x_2, \cdots, x_n)$$

which minimizes the objective function:

$$A(X) = \sum_{x_i \in S} a(x_i) \quad (1)$$

subject to:

$$T(X) = \sum_{j=1}^{N} \{F_j \times \big(t(B_j, X) + c_j\big)\} - b \leq T_max \,, \quad (2)$$

and

$$P(X) = \sum_{x_i \in S} p(x_i) \leq P_max. \quad (3)$$

The key point in computing $T(X)$ in Eq.(2) is to obtain the value of $t(B_j, X)$. It can be computed by using a HW/SW partitioning-oriented pipeline scheduling algorithm [7] to estimate $t(B_j, X)$ for basic block B_j under configuration X, where the pipeline architecture is given (e.g., the number of pipeline stages, whether there is a bypass or not, and so on). The pipeline scheduler must detect and solve all types of pipeline data hazards, and should produce pipelined schedules as if without HW interlock to estimate the total execution cycles of the basic block under given configuration. The pipeline control hazards are addressed in introducing the coefficients c_j's, which can also be defined by the pipeline. Note that the number of clock cycles due to control hazards is equal to $\sum_{j=1}^{N}(F_j \times c_j) - b$. Also, the pipeline scheduling algorithm detects and resolves all structural hazards.

3 Proposed Algorithm

The problem described in the previous section is NP-hard. In order to solve it in reasonable computation time an algorithm based on the branch-and-bound method is used. The algorithm characteristics are described in the following subsections.

3.1 Input and Output

The inputs to the algorithm are the execution frequencies (f_i's) of basic operations, execution frequencies of basic blocks (F_j's), b, execution cycle constraint (T_max), power consumption constraint (P_max), the module information database which includes area (gate count), power consumption, and the database of the HW/SW implementation methods of each operation with the number of execution cycles. The output of the algorithm is the set of optimal implementation methods of these operations, and a pipelined schedule of the application program's code under selected optimal pipeline architecture.

3.2 Variable Reordering

In order to reduce the computation time of the algorithm, the operations to be considered are sorted in an ascending order according to a heuristic function before applying the branch-and-bound method. Many heuristic functions can be tested to find the best one. The following heuristic was found to be the best:

$$h(i) = \min_{x_i \in M_i} a(x_i). \quad (4)$$

This heuristic tries to implement first such an operation by hardware that needs less area. Inside each M_i (set of implementation methods for operation #i by x_i, $i = 1, \ldots, n$) the elements are sorted in an ascending order of $t_i(x_i)$, where ties are broken arbitrary.

3.3 Lower Bound Function

The key to solving an optimization problem efficiently by the branch-and-bound method is to design a good lower bound function to prune as many non-optimum solutions as

possible. The lower bound function used in the algorithm is as follows:

$$Lower_bound = \sum_{x_i \in S_d^{'}} a(x_i) + \sum_{x_i \in S_d^{''} \setminus S_d^{'}} a(x_i) , \quad (5)$$

where d represents the depth of the search tree node to be investigated and associated with x_i, and '\' represents set subtraction. $S_d^{'}$ and $S_d^{''}$ are defined as follows:

$$S_d^{'} = \bigcup_{i=1}^{d-1} \{x_i\} , \quad S_d^{''} = \bigcup_{i=d}^{n} \{x_i^{min}\} , \quad (6)$$

where x_i^{min} is defined as follows. Assume that

$$M_i^{min} = \{x \in M_i \mid a(x) = \min_{x_i \in M_i} a(x_i)\}. \quad (7)$$

If $|M_i^{min}| = 1$, let x_i^{min} be the element in M_i^{min}. If $|M_i^{min}| > 1$, there are more than one implementation methods with the same minimum area for operation #i. In this case x_i^{min} is determined as follows:

$$x_i^{min} \in S_d^{'} \cap M_i^{min}, \quad \text{if } S_d^{'} \cap M_i^{min} \neq \phi , \quad (8)$$

$$x_i^{min} \in M_i^{min}, \quad \text{otherwise.} \quad (9)$$

In Eqs.(8) and (9), ties are broken arbitrary. In Eq.(8), the shown x_i^{min} is chosen so that the solution optimality is guaranteed.

During the branch-and-bound search, each node in the search tree corresponds to an (incomplete) solution, and its cost is of the (incomplete) path. The cost of the node plus the minimum cost along the remaining path represents a lower bound for the nodes of its subtree. If the lower bound of a node is greater than or equal to the cost of any previously found feasible solution, the subtree of the node can be pruned.

3.4 Estimation of Lower Bound for Execution Cycles

Besides the lower bound in Eq.(5) for the objective function of Eq.(1), another important fact to quickly find an optimum solution by the branch-and-bound method is to ensure the given constraints to be satisfied at the current node in the search tree. In this paper we focus on finding the estimation of execution cycles at a current node as the lower bound of $T(X)$ being estimated fully when a leaf node is reached. Difficulty in designing such a lower bound is how to accurately predict effects of the pipeline characteristics of selected functional units (FUs) at the current moment as well as to predict those of FUs to be selected later. Such a desired lower bound function denoted as $T_{gain}(d, x_i)$ for execution cycles $T(X)$ at a node (for x_i) of depth d can be as follows:

$$T_{gain}(d, x_i) = T_{gain}(d-1, x_{i-1}) + (f_i \times u_i(x_i)) , \quad (10)$$

where

$$u_i(x_i) = \begin{cases} 1, & \text{if } x_i \text{ is a HW implementation,} \\ t_i(x_i), & \text{if } x_i \text{ is a SW implementation.} \end{cases} \quad (11)$$

At the beginning of the algorithm we set

$$T_{gain}(0, x_0) = Stall_{fast} + f_0 = T(X_{fast}) - \sum_{i=1}^{n} f_i , \quad (12)$$

where $T(X_{fast})$ is computed by using Eq.(2) with the hypothetical one-cycle FUs denoted by X_{fast}, and $Stall_{fast} = T(X_{fast}) - \sum_{i=0}^{n} f_i$, i.e. $T_{gain}(0, x_0) - f_0$ as well, is the number of pipeline stalls (pipeline data hazards, control hazards and structural hazards) in executing the given application program. Please refer to Ref. [14] for the reason of these considerations. We have

$$T_{gain}(d, x_i) < T_{gain}(d', x_{i'}) \leq T(X) , \quad 0 \leq d < d' \leq n. \quad (13)$$

Note that $T_{gain}(d, x_i)$ depends on x_i (i.e., on X) as shown in Eq.(10). We can consider that $d = i$ if x_i's are sorted by using Eq.(4) and renamed before applying the algorithm. This lower bound allows us to prune as soon as possible all paths in the search tree which do not satisfy the execution cycle constraint by finding that $T_{gain}(d, x_i) > T_max$.

3.5 Functional Module Sharing

The functional module sharing relation is to be considered when the cost is calculated and the lower bound is estimated. The functional module sharing can be solved as follows:

"For a given path, if a hardware module which can implement more than one operations is selected to implement an operation for the first time, its cost (gate count or power consumption) is accumulated. If the same module is then selected for the second time or later to implement other operation, its cost is not accumulated."

3.6 Power Consumption

The power consumption estimation $P(X)$ in Eq.(3) is still a crude approximation by simply adding average power figures of the different modules without accounting for switching activity.

4 Implementation in PEAS-I

In the previous section we have generally defined the HW/SW partitioning problem for ASIP design, and proposed the algorithm using the branch-and-bound method. In the rest of this paper we evaluate the proposed method under the circumstance of PEAS-I as an example.

4.1 PEAS-I System

A HW/SW codesign system PEAS-I (Practical Environment for ASIP development - type I) [6] employs a formal method to synthesize an optimal instruction set processor by solving Instruction set implementation Method Selection Problems (IMSP) types 1, 2 and 3. IMSP-1 [8] is set up assuming no interaction among the operations, and each operation was to be implemented using a separate HW module. However, IMSP-2 [9] 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 [10] yields the design of ASIP with the least HW cost subjecting to execution cycle and power consumption constraints. Applying the proposed method, we deal with **IMSP-3P** for designing a **pipelined** ASIP with the least HW cost subjecting to execution cycle and power consumption constraints.

4.2 Pipelined Architecture

The minimum part in the PEAS-I CPU core architecture is the '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. 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 the CPU has no identical FUs. The architecture has a register bypass to forward computation results to Kernel or FUs. When a load instruction loads a value from memory needed by the next immediate instruction we have to stall the latter one cycle. Each FU can be multi cycle and pipelined. These characteristics of the pipeline are used by the pipeline scheduler [7] to estimate the value of $t(B_j, X)$ in Eq.(2).

The instruction set architecture of the designed ASIP is based on the GNU C Compiler (GCC) abstract machine model [11]. 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 executes any C program. The primitive operations are implemented in hardware by Kernel. The basic operations contain other C operators and functions that are not primitive operations and can be implemented using some hardware choices (such as fast or slow hardware modules) or using a software subroutine that uses primitive operations. Some of the basic operations are shown in the last column of Tab.1.

4.3 Module Database

A module database of non-pipelined FUs synthesized by using the high-level synthesis tools of PARTHENON [12] with cell library VTI.lib from VLSI Technology, Inc. has been given in Ref. [10]. Another HW/SW module information database of both pipelined and non-pipelined FUs as well as SW subroutines has been described in Ref. [13]. As shown in Ref. [10], the former database is with a search space of above 1.8×10^4 nodes. The number of nodes in the search tree generated by the latter database is over 8.2×10^7 (please refer to Refs. [13, 14]). We use the latter database shown in Tab.1 to perform the experiments for both IMSP-3 and IMSP-3P solvers. Note that in Tab.1 the multiplication has ten HW implementation methods, the division has six, while the extension and shift have only one for each. Moreover, each operation has one or more SW implementation methods (run-time subroutines), whose execution cycles depend on the application program with its input data.

5 Experimental Results

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

5.1 Sample Programs

The sample programs used in the experiments are as follows [13, 14]: 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. [15].)

These sample programs with associated input data were fed to the Application Program Analyzer of the PEAS-I system to obtain the execution frequencies of basic operations, of basic blocks, etc. The analyzed results have been shown in Refs. [13, 14]. The code optimization was performed by the GCC, the pipeline scheduling was performed by the scheduler described in Ref. [7].

Using the adaptive database generator [13] we got the execution cycle estimation of SW implementations of the basic operations met in these sample programs as shown in Tab.2.

Table 1: Part of module database with pipelined and non-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)**

Table 2: Part of adaptive database with # execution cycles of SW modules for ESS, IMC, and *diffeq*.

i	Basic Operation	# Execution cycles		
		ESS	IMC	*diffeq*
1	**mul**	**52**	**52**	**48**
2	**div**	**280**	**279**	**283**
3	**mod**	**278**	**278**	**278**
4	**ashl**	**21**	**21**	**21**
5	**extendqi**	**8**	**8**	**8**
6	**z_extendqi**	**1**	**1**	**1**

5.2 Hardware/Software Partitioning

Using the analyzed information from the given application program, the IMSP-3P algorithm accordingly selected the optimum partitioning for different values of execution cycle constraint. The power consumption was ignored to simplify the experimental cases. For instance, the results for the ESS sample program are partly shown in Tab.3. In this table the second column represents the execution cycles constraints T_max in cycles. The third and fourth columns represent the predicted execution cycles and predicted areas by the algorithm. The last column shows the selected HW modules that implement the corresponding basic operations (other operations are implemented in SW.) For any given execution cycle constraint, the shown partitioning represents the optimum one.

5.3 Execution Cycle - Area Tradeoff

Figure 1 shows the execution cycle - area tradeoff by IMSP-3 and IMSP-3P for ESS. Note that IMSP-3P yields

Table 3: Predicted execution cycle, area, and HW/SW partitioning by IMSP-3P for ESS program.

#	T_max	$T(X)$ (cycles)	$A(X)$ (gates)	Selected HW modules (with Kernel)
1	400000	353729	14918	(Kernel only)
2	353700	341066	15055	extend
3	341000	324371	15674	b_alsft
4	324300	311708	15811	b_alsft extend
5	311700	307988	18204	b_alsft extend mul_seq
6	307900	130863	18314	div_seq
7	130800	118200	18451	extend div_seq
8	118100	101505	19070	b_alsft div_seq
9	101500	88842	19207	div_seq
10	88800	80254	21269	div_seq_p17
11	80200	74378	21619	div_2seq
12	74300	70148	24012	div_2seq mul_seq
13	70100	67448	24780	div_2seq mul_bpr
14	67400	66458	27019	div_2seq mul_bpr_p8
15	66400	65708	27737	div_2seq mul_3clk
16	65700	65476	28948	mul_seq div_2seq_p9
17	65400	62776	29716	mul_bpr div_2seq_p9
18	62700	61696	31955	mul_bpr_p8 div_2seq_p9
19	61600	60736	32673	mul_3clk div_2seq_p9
20	60700	60586	34302	mul_csa div_2seq_p9

Note: Selected HW modules in #9 – #20 contain 'b_alsft' and 'extend'.

better designs than IMSP-3 due to considering the pipeline characteristics. Note that for the same execution cycle constraint, IMSP-3P can give the ASIPs of up to 8.3%, 13.2%, and 10.7% of gate counts less than IMSP-3 for ESS, IMC, and *diffeq*, respectively. Also, the performance of the ASIPs designed by IMSP-3P can be improved by 4.2%, 8.4%, and 6.7% compared to IMSP-3 for ESS, IMC, and *diffeq*, respectively. Especially, IMSP-3P can handle tight execution cycle constraints (i.e., high performance) of below 60000, 70000, and 25000 cycles for ESS, IMC, and *diffeq*, respectively, whereas IMSP-3 could not.

5.4 Area and Execution Cycle Prediction Errors

Note that IMSP-3P can accurately estimate the area and execution cycles of the designed ASIP without having to synthesize the design. Our approach enables designers to predict the main part of the execution cycle vs. area tradeoff of their design. The measured area values were obtained by synthesizing the design using PARTHENON while the measured execution cycles were obtained using the PEAS-I system simulator. It is found that the area prediction values were fairly good with an average gate count error of about 1.7% as the same as shown in Ref. [10], and the execution cycle estimation errors were below 2% (with the adaptive database approach as in Ref. [14].) Especially, the execution cycle estimation errors were almost 0% in cases of all HW implemented operations.

5.5 Constraint Satisfiability

We have found that all designs by IMSP-3P were satisfiable (in terms of that the execution cycle constraints have been satisfied) due the accurate estimation of execution cycles and the pipeline hazards, whereas most of the designs by IMSP-3 were with exceeded execution cycles (i.e. did not satisfy the execution cycle constraints) after the first trial pass. Therefore, it is necessary to iterate some times the process 'reduce the execution cycle constraint – re-perform the IMSP-3 solver – estimate the design by simulation' until the design is satisfiable.

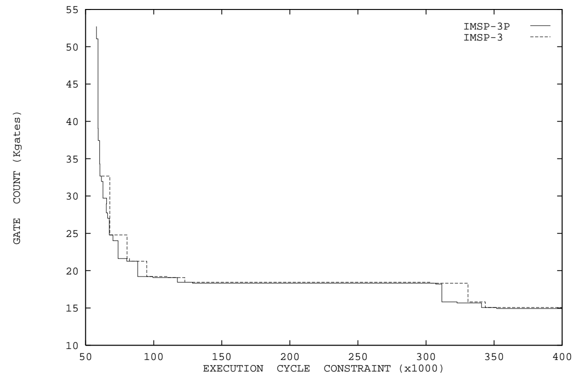

Figure 1: Execution cycle - area tradeoff for ESS.

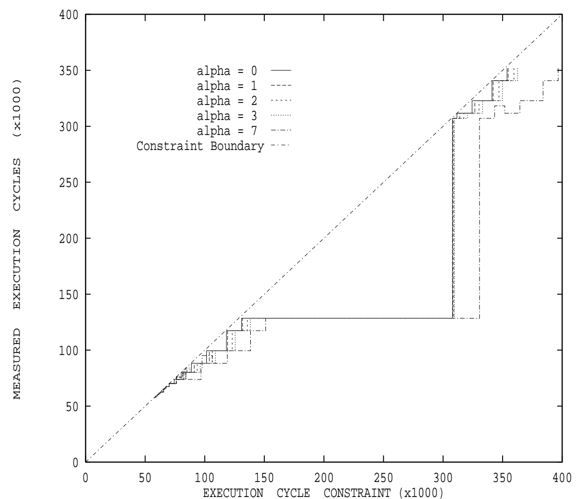

Figure 2: Execution cycle constraint satisfiability for ESS using the safe factor.

The execution cycle constraint satisfaction by IMSP-3P without iterating the design process is due to the accurate estimation of execution cycles under any architecture configuration and due to taking the pipeline characteristics into account by the proposed method.

5.6 Effect of Input Data Variation

Note that IMSP-3P can produce the satisfiable pipelined ASIPs for the given application program with *the given associated input data*. However, for other data the constraint might not be satisfied because the execution cycles of the SW module depend on data fed to operands. But we can enhance the satisfiability of the design for some class of input data to the given application program by introducing the *safe factor* α together with the *standard deviation* σ_i of basic operation $\#i$ as follows: for the SW implemented basic operation $\#i$ the execution cycles will be

$$t_i(x_i) \;=\; \overline{\tau}(\#i) \;+\; \alpha \times \sigma_i \;, \tag{14}$$

where $\overline{\tau}(\#i)$ is the average execution cycles of operation $\#i$ by SW with the given input data from profiling the given ap-

plication program with associated input data [14]. So far we have set $\alpha = 0$ in running the IMSP-3 and IMSP-3P solvers. The designer can drive the design process by setting α to any suitable value to enhance the satisfiability of the design. The experimental results for α=0, 1, 2, 3, and 7 for ESS are shown in Fig. 2 (with $\overline{\tau}_{mul}$ = 52, σ_{mul}=13.97; $\overline{\tau}_{div}$ = 280, σ_{div}=1.44; $\overline{\tau}_{ashl}$ = 21, σ_{ashl}=0.40; etc.) In this figure, for design constraints of below 74000 cycles the measured execution cycles are very close to the given execution cycle constraints for any α because all basic operations were implemented in HW. Note that the larger the α, the more the validation of design is guaranteed. For example, for α = 1, 2, 3, and 7 we have the reserved execution cycles (for data variation of software operations) of 4.4%, 7.3%, 10.7%, and 20.37%, respectively. However, the designs might suffer from a supplementary HW cost.

5.7 Algorithm Efficiency

We have shown the effectiveness of the IMSP-3P. We have also found that IMSP-3P with the proposed algorithm, especially with the described lower bounds of area and execution cycle estimation, is so efficient that the optimal pipelined ASIP as well as its instruction set can be selected within 2 seconds on SPARCstation 10 (SS-10) for any execution cycle constraint. The average number of visited nodes in the search tree is 124, 168, and 176 for ESS, IMC, and *diffeq*, respectively. The time for profiling an application program is about 20 seconds as shown in Ref. [14].

6 Conclusion and Future Work

In this paper an effective and efficient partitioning method for designing a pipelined ASIP with least gate count under given execution cycle and power consumption constraints has been introduced. The method is based on a combinatorial optimization formalization that selects the implementation methods of the basic operations. A branch-and-bound algorithm was used to solve the presented partitioning problem and was implemented in C language. Designing the good lower bounds for the algorithm has been presented. The effectiveness and efficiency of the algorithm have been demonstrated through performing a set of sample programs. The area and the execution cycles of the designed ASIPs chips that execute sample programs were predicted for different partitioning combinations. The predicted gate count was found to have an average error rate of 1.7% while the execution cycle estimation was found to have an error rate of below 2%. As an implementation, IMSP-3P is a full automatic HW/SW partitioning algorithm able to give an optimal solution in reasonable computation time. This feature makes our approach differ from existing ones.

However, there are the following limitations in this work: (1) the cost of other components of CPU such as bus width, RAM, ROM and so on was not addressed; (2) the power consumption is still a crude approximation. Switching activity and the power consumption in case of the software implementations should be taken into account, for example, by using the technique proposed by V. Tiwari [16]; (3) the satisfiability of the design is still not fully guaranteed; (4) the 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 a HW/SW partitioning algorithm for designing a pipelined ASIP with lowest power consumption under execution cycle and gate count constraints is also planned.

Acknowledgments

The authors would like to express their thanks to NTT Communication Science Laboratories, VLSI Technology, Inc., Science Create, Co. Ltd., Japan, for their kind assistance. 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.

References

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

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

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

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

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

[6] J. Sato, A. Alomary, Y. Honma, T. Nakata, A. Shiomi, N. Hikichi, and M. Imai, "PEAS-I: A Hardware/Software Codesign System for ASIP Development," *IEICE Trans. Fundamentals*, vol.E77-A, no.3, pp. 483 – 491, Mar. 1994.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.