Behavioral Partitioning with Exploiting Function-Level Parallelism

Yuko Hara[†]   Hiroyuki Tomiyama[†]   Shinya Honda[†]   Hiroaki Takada[†]   Katsuya Ishii[‡]

Graduate School of Information Science, Nagoya University
Furo-cho, Chikusa-ku, Nagoya, Aichi 464-8603, Japan
Email: †{hara, tomiyama, honda, hiro}@ertl.jp, ‡ishii@itc.nagoya-u.ac.jp

Abstract—**This paper proposes a method to efficiently generate hardware from a large behavioral description by behavioral synthesis. For a program with functions which are executable in parallel, this proposed method determines an optimal behavioral partitioning which fully exploits the function-level parallelism with simultaneously minimizing the area in the datapath and control path. This partitioning problem is formulated as an integer programming problem. Experimental results demonstrate the effectiveness of our proposed method.**

I. INTRODUCTION

Due to the continuous increasing size of LSIs, the LSI design is being gradually shifted from the traditional Register-Transfer Level (RTL) design with Hardware Design Languages (HDLs) to behavioral synthesis, which automatically synthesizes an RTL circuit from a behavioral description [1]. Yet, behavioral synthesis has not been widely adopted in industry because the quality of automatically synthesized circuits is inferior to that of human-designed one, especially when synthesizing from large behavioral descriptions.

In general, a large program consists of a number of functions. There are mainly two techniques to handle functions in behavioral synthesis; *function inlining* and *function-based partitioning*. With function inlining, all the callee functions are inlined into their callers, resulting in a huge main function, and then the main function is fed by a behavioral synthesis tool. This produces an inefficient circuit with a long delay and large area of control path although hardware resources can be shared effectively among functions. On the other hand, function-based partitioning produces   hardware modules (one main module and   − 1 sub modules) from a program consisting of functions. This can reduce the delay and area of control path, while the overall datapath may be increased since hardware resources cannot be shared between modules.

Our earlier work [2] has proposed an   -way behavioral partitioning based on an integer programming problem, where   denotes the number of hardware modules. [2] optimally determines functions to be implemented in a main module and ones to be in sub modules in such a way that the overall datapath is minimized while keeping the complexity of the control path lower than a certain level. Simultaneously, it optimizes the number of modules   . However, [2] does not take the function-level parallelism into account, which degrades the performance for programs where functions are executable in parallel. This paper proposes an improved method of [2], which determines an optimal partitioning which exploits the function-level parallelism.

There have been several studies on behavioral partitioning to efficiently synthesize hardware by behavioral synthesis. Vahid has extensively studied behavioral partitioning for sequential programs [3]-[6]. [3], first, decides the appropriate granularity
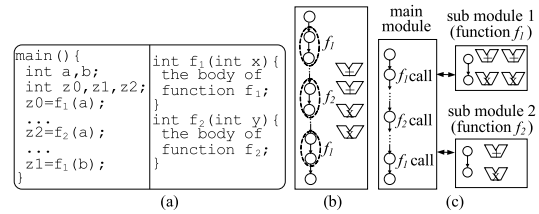

Fig. 1.   Traditional methods: (a) An example program, (b) Function inlining, (c) Function-based partitioning without clustering.

of procedures (functions) using various techniques [4]-[6], and then performs traditional   -way partitioning. However, his work does not take the parallelism among procedures (functions) into account. Takahashi et al. propose a partitioning method that considers the parallelism of processes (functions) which are explicitly specified to be executable in parallel [7]. However, both [3] and [7] assume the predetermined number of modules, which dose not always perform the optimal partitioning, while our approach can obtain a partitioing which cannot found by [3] or [7], by simultaneously optimizing the number of modules and the partitioning.

This paper is organized as follows. Section II describes fundamental techniques to handle functions in behavioral synthesis. Section III proposes a function-level partitioning method based on integer programming. Section IV shows experimental results. Finally, Section V concludes this paper.

II. FUNDAMENTAL PARTITIONING TECHNIQUES

This section presents fundamental partitioning techniques for behavioral synthesis and discusses their advantages and disadvantages.

A. Function Inlining

Function inlining replaces function calls with the bodies of the called functions. Let us consider an example program shown in Fig. 1(a). This program consists of a main function and two functions,   1 and   2, which are called from the main function. Fig. 1(b) shows the FSM of a circuit which is synthesized from Fig. 1(a) with inlining. Note that   1 is inlined twice since it is called twice from the main function.

Inlining has an advantage of minimizing the total datapath by sharing resources between functions. Assume that   1 requires two adders and two multipliers, and   2 requires one adder and one multiplier. Since these two functions are implemented in the same module, they can share the resources, that is, two adders and two multipliers are used in total. Moreover, there is no performance degradation caused by inter-module
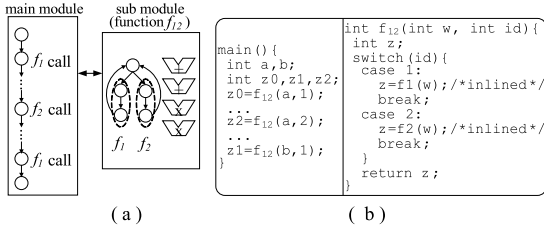
2008 ISOCC

Fig. 2.   Clustering: (a) Partitioning with clustering, (b) A refined program.

Fig. 3.   An example program where functions are executable in parallel: (a) A pseudo program, (b) An example partitioning.

communication. Also, inlining extends operation-level parallelism and the scope of optimizations such as common subexpression elimination and constant propagation. However, the large number of states may produce an inefficient circuit with a long critical path delay due to the complicated control path, or behavioral synthesis may not be completed within a practical time. These disadvantages become significant, especially for programs where large functions are called a number of times from different points of the program text.

B. Function-Based Partitioning without Clustering

Function-based partitioning without clustering is to run behavioral synthesis for each function. This approach produces hardware modules from a program consisting of functions. Let us consider the same example program in Fig. 1(a). The FSM of a circuit synthesized with this approach is shown in Fig. 1(c), where one main module and two sub modules are generated. Note that only a single module is generated for f_1 even though it is called twice.

This generates individually small modules, leading to the small area and short delay in the controller circuit. However, it has a disadvantage of increasing its total datapath area because the resources cannot be shared between modules such as sub modules 1 and 2 in Fig. 1 (c), even though they are sequentially called. Moreover, the inter-module communication overhead may degrade the performance.

C. Function-Based Partitioning with Clustering

Function-based partitioning with clustering is to cluster and implement some functions in a sub module, instead of implementing them individually. Fig. 2 (a) shows the FSM by clustering f_1 and f_2 of Fig. 1 (a). It suppresses the number of states in the main module. Also, the sub module minimizes the datapath area by sharing the resources. Moreover, clustered functions are implemented only once in the sub module, leading to the small control path area in the sub module. Thus, clustering multiple functions can minimize the overall area.

This can be achieved by transforming a program as shown in Fig. 2 (b). A function f_{12} is newly defined, which calls either f_1 or f_2 based on a parameter id. f_1 and f_2 are inlined into f_{12}, while f_{12} itself is not inlined.

When synthesizing from a program with a number of functions, however, clustering too many functions may results in the large number of states in a sub module, similar to inlining. Thus, it is important to determine the appropriate function clustering.

III. EXPLOITING FUNCTION-LEVEL PARALLELISM

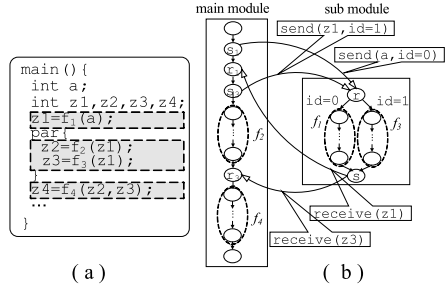This section proposes a new behavioral partitioning method exploiting the function-level parallelism.

A. Problem Description

We propose a method to optimally determine functions to be implemented in the main module and ones to be in sub modules with exploiting the function-level parallelism. This problem is formulated as an integer programming problem. Our goal is to minimize the overall datapath area (i.e., the total cost of hardware resources) with fully exploiting the function-level parallelism of an input program while keeping the complexity (i.e., the number of states) of individual modules lower than a certain level specified by a designer.

For simplicity, we assume a single hierarchy of function calls. In general, a program consists of multiple hierarchies of function calls. To deal with those programs, if a function f calls another function f', we inline either f or f' into its caller before the partitioning step. For this purpose, granularity selection techniques presented in [3] can be used.

Let us consider an example program depicted in Fig. 3(a). We assume that a pseudo statement, par, explicitly specifies the parallel execution of functions. In Fig. 3(a), the execution of function f_1 is followed by the parallel execution of functions f_2 and f_3 with taking z1 as input. Then, function f_4 is executed with taking z2 and z3 as input.

The FSM in Fig. 3(b) can be obtained by our proposed method for Fig. 3(a). In Fig. 3(b), f_2 and f_4 are inlined into a main module, while f_1 and f_3 are implemented in a same sub module. Black and white arrows in Fig. 3(b) represent the state transition and inter-module communication, respectively. The state $s_1/\bar{s}_1$ represents the state to send/receive data to/from f_1 implemented in the sub module. Note that f_2 and f_3 are implemented in different modules since the par statement in Fig. 3(a) explicitly specifies that they are executable in parallel.

B. Problem Formulation

We show the formulation of the proposed partitioning problem as an integer programming problem. Notations are defined in Table I. Here, let us explain P_l with an example of Fig. 3(a). Fig. 3(a) totally has three function call points, corresponding to gray boxes. Note that functions in one par statement are called at the same point. In Fig. 3(a), f_1 is called at the first function call point, P_0, so $P_0 = \{ f_1 \}$. Next, since f_2 and f_3 are in the same par statement, they are called at the same point P_1, thus $P_1 = \{ f_2, f_3 \}$. Finally, f_4 is called at P_2, so $P_2 = \{ f_4 \}$.

Next, a 0-1 variable $x_{i,k}$ is defined as follows:

$$x_{i,k} = \begin{cases} 1 & \text{if function } f_i \text{ is implemented in module } M_k \\ 0 & \text{otherwise} \end{cases}$$

TABLE I Definition of notations.

N_R	The number of hardware resource types
r_j	Hardware resource $(j = 0, 1, \ldots, N_R - 1)$
a_j	The area of resource r_j
N_F	The number of functions in a program
f_i	Function in a given program $(i = 0, 1, \ldots, N_F - 1)$
	Note that f_0 represents a main function.
c_i	The number of times function f_i is called in the program text
$n_{i,j}$	The number of resource r_j required by function f_i
s_i	The number of states in a module synthesized from function f_i
	Note that s_i does not include the number of states for inter-module communication.
s_{snd}	The number of states required for inter-module communication to send data
s_{rcv}	The number of states required for inter-module communication to receive data
N_M	The number of hardware modules
m_k	Hardware module $(k = 1, \ldots, N_M - 1)$
	Note that m_0 represents a main module synthesized from the main function.
S_k	The number of states in module m_k
S_{const}	The designer-specified constraint on the number of states for each module
fc_l	The l-th function call point
FC_l	A set of functions which are called at fc_l
FC_{total}	The total number of function call points which require inter-module communication
A_k	The datapath area of module m_k
A_{total}	The total datapath area
N_k^{cmp}	The number of comparator required in module m_k
a^{cmp}	The area of a comparator

where $\sum_k x_{i,k} = 1$.

Functions which are executable in parallel, i.e., functions which belong to a same FC_l, such as f_2 and f_3 in Fig. 3(a), should be implemented in different modules. Thus, the next formula should be met.

$$f_i \in FC_l \land f_{i'} \in FC_l \land i \neq i' \Rightarrow x_{i,k} \cdot x_{i',k} = 0 \ \forall k \quad (1)$$

Then, the number of states in module m_k is estimated as follows:

$$S_0 = s_0 + \sum_i c_i \cdot s_i \cdot x_{i,0} + FC_{total} \cdot (s_{snd} + s_{rcv}) \quad (2)$$

$$S_k = \sum_i s_i \cdot x_{i,k} + (s_{snd} + s_{rcv}) \quad k \neq 0 \quad (3)$$

Formula (2) represents the estimated number of states in the main module m_0. For an inlined function, its number of states s_i multiplied by its number of time being called from the main function c_i is added. $FC_{total} \cdot (s_{snd} + s_{rcv})$ denotes the total number of states for inter-module communication with sub modules. Since one pair of s_{snd} and s_{rcv} is required for one inter-module communication, FC_{total} pairs of s_{snd} and s_{rcv} are needed in total. FC_{total} is obtained by the next formula.

$$FC_{total} = \sum_l y_l \quad (4)$$

where a 0-1 variable y_l is defined as follows:

$$y_l = \begin{cases} 1 & \text{if } \prod_{i | f_i \in FC_l} x_{i,0} = 0 \\ 0 & \text{otherwise} \end{cases}$$

Formula (3) represents the estimated number of states in the sub module m_k. The last term denotes the number of states to communicate with the main module. The number of states in each module cannot exceed the limit specified by a designer. Therefore, the formula below must hold.

$$S_k \leq S_{const} \quad (5)$$

Next, the datapath area of the main module and sub modules can be estimated by formulas (6) and (7), respectively.

$$A_0 = \sum_j \{ \max_i (x_{i,0} \cdot n_{i,j}) \cdot a_j \} \quad (6)$$

$$A_k = \sum_j \{ \max_i (x_{i,k} \cdot n_{i,j}) \cdot a_j \} + N_k^{cmp} \cdot a^{cmp} \quad (7)$$

In module m_k, the required number of resource r_j is given by the maximum number among $n_{i,j}$'s for functions clustered into module m_k. A sub module which implements more than one function requires comparators to determine the function to be executed. The required number of comparators, N_k^{cmp}, is $\sum_i x_{i,k}$ when $\sum_i x_{i,k}$ is greater than one, otherwise 0.

Then, the total datapath area A_{total} can be estimated by the formula below:

$$A_{total} = \sum_k A_k \quad (8)$$

As shown above, the optimization problem on behavioral partitioning can be defined as the integer programming problem, which finds $x_{i,k}$ minimizing formula (8) with satisfying the constraints in formulas (1) and (5). By finding the optimal solution of this integer programming problem, a designer can obtain the optimal partitioning. To find the optimal solution, some commercial ILP solvers can be used, or some general algorithms such as the branch and bound method, simulated annealing, and the genetic algorithm, can be applied.

Note that our method finds the optimal number of modules as well as the optimal N_F-way partitioning simultaneously, by simply adding the following equation into the integer programming formulation.

$$N_M = N_F \quad (9)$$

This equation does not mean that the number of modules must be exactly N_F, but means that it must be equal to or less than N_F. If the optimal number of modules is less than N_F, a solver will yield $x_{i,k} = 0 \ \forall i$, for some module m_k.

IV. Experiments

We conducted two sets of experiments to demonstrate the effectiveness of our partitioning method. We used two benchmark programs, lms and fft. These programs are composed of a main function and several double-precision floating-point arithmetic functions [8]. Note that fft and lms are rather large, which consist of more than 800 lines of C code, without including comment lines or empty lines.

First, we performed behavioral synthesis for each function in order to obtain necessary parameters, such as the number of states and the types and numbers of required resources. Then, we solved the integer programming problem proposed in Section III. We developed a C program based on the depth-first search with pruning algorithm. The constraint on the number of states were given every 10 and 20 for lms and fft, respectively. Next, for the obtained partitioning, we conducted behavioral synthesis and logic synthesis. Xilinx Virtex 4 [9] was specified as a target device. eXCite [10] and Synplify-Pro [11] were used for behavioral synthesis and logic synthesis, respectively. All the synthesis processes were optimized for performance maximization. RTL simulation was performed to measure the execution cycles.

Tables II and III summarize the results with a method in [2] (Method-1) and the method proposed in this paper (Method-2) for lms and fft, respectively. The first columns denote the constraint on the number of states. Due to the space limitation, we omit the obtained partitionings in Table III and show only the number of states in individual modules. Numbers in parentheses under the results with Method-2 are normalized values whose baselines are the results with Method-1 under the same constraint. Function-based partitioning without clustering (*w/o clustering*) were performed for the comparison. Note that *w/o*

TABLE II EXPERIMENTAL RESULTS FOR LMS.

Constraint on states	Partitioning (Method-1 [2])	No. of states	Area (K gates)	Clock Period (ns)	Exec. cycles	Exec. time (μs)	Partitioning (Method-2, this paper)	No. of states	Area (K gates)	Clock Period (ns)	Exec. cycles	Exec. time (μs)
180	{main, add, sub, mul, div, addmul}	—	—	—	—	—	{main, add, sub, addmul} {mul, div}	92 51	583 (1.44)	29.7 (0.82)	1,538 (0.83)	45.7 (0.68)
100-170	{main} {add, sub, mul, div, addmul}	37 88	404	36.1	1,853	66.9						
90							{main, mul} {add, sub, div, addmul}	81 75	528 (1.31)	36.8 (1.02)	1,548 (0.84)	57.0 (0.85)
80	{main, add, sub} {mul, div, addmul}	50 72	481	40.3	1,768	71.2	{main} {add, sub, div, addmul} {mul}	28 75 15	486 (1.01)	37.0 (0.92)	1,583 (0.90)	58.5 (0.82)
60-70	{main} {add, sub, addmul} {mul, div}	37 39 51	494	29.0	1,853	53.7	{main} {add, sub, addmul} {mul, div}	28 39 51	494 (1.00)	30.7 (1.06)	1,583 (0.85)	48.6 (0.90)
50	{main} {add, mul, addmul} {sub, div}	37 44 46	511	30.2	1,853	56.0	{main} {add, sub, addmul} {mul} {div}	28 39 15 38	561 (1.13)	22.5 (0.75)	1,583 (0.85)	35.7 (0.82)
40	{main} {add, sub} {mul, addmul} {div}	37 18 36 38	496	23.5	1,853	43.6						
w/o clustering	{main} {add} {sub} {mul} {div} {addmul}	37 10 10 15 38 23	591	24.0	1,853	44.5	{main} {add} {sub} {mul} {div} {addmul}	28 10 10 15 38 23	598 (1.01)	23.3 (0.97)	1,583 (0.85)	37.0 (0.83)

TABLE III EXPERIMENTAL RESULTS FOR FFT.

Constraint on states	No. of states (Method-1 [2])	Area (K gates)	Clock period (ns)	Exec. cycles	Exec. time (μs)	No. of states (Method-2, this paper)	Area (K gates)	Clock period (ns)	Exec. cycles	Exec. time (μs)
340	—	—	—	—	—	137/ 10/ 128	1,029 (1.26)	57.0 (1.26)	6,047 (0.68)	344.5 (0.86)
220-320	—	—	—	—	—					
200	—	—	—	—	—					
180	—	—	—	—	—					
160	58/ 70/ 149	813	45.0	8,884	400.2					
140	132/ 136	933	38.2	8,816	336.5	138/ 13/ 128	1,035 (1.11)	54.5 (1.43)	6,075 (0.69)	331.1 (0.98)
120	58/ 103/ 115	992	50.6	8,860	448.5	68/ 115/ 92	1,068 (1.08)	39.1 (0.77)	6,053 (0.68)	236.5 (0.53)
100	92/ 100/ 82	1,050	39.0	8,855	345.7	48/ 82/ 87/ 51	1,068 (1.02)	38.7 (0.99)	6,101 (0.69)	236.0 (0.68)
80	58/ 70/ 71/ 79	1,087	38.2	8,860	338.6	68/ 79/ 51/ 79	1,175 (1.08)	35.8 (0.94)	6,053 (0.68)	216.8 (0.64)
w/o clustering	58 / 10 / 10 / 15 / 38 / 5 / 71 / 79	1,179	36.8	8,860	325.7	48 / 10 / 10 / 15 / 38 / 5 / 71 / 79	1,172 (0.99)	35.9 (0.98)	6,101 (0.69)	218.8 (0.67)

clustering under Method-1 is a traditional technique, and *w/o clustering* under Method-2 is its straightforward improvement. Behavioral synthesis could not be completed within 24 hours for partitionings obtained with Method-1 when the constraint is greater than or equal to 180 for both fft and lms.

Let us compare Method-1 and Method-2 under the same constraint. In lms and fft, multiple functions are called in parallel at three function call points out of six points, and four out of 11, respectively. Tables II and III show, by exploiting the function-level parallelism, that Method-2 achieves up to 17% and 32%, and on average 14.6% and 31.6% reduction on the number of execution cycles, respectively. On the other hand, Method-2 increases the area since Method-2 implements parallel functions in different modules due to the function-level parallelism, which leads to the decreased resource sharing among functions.

As the constraint becomes loose, the area decreases due to the increased resource sharing, while the execution time increases due to the long delay of control path. This means that the area-performance trade-off points are well described by our proposed method, and designers can efficiently obtain the optimal partitioning before behavioral synthesis in actual. However, some exceptions can be seen on the area and the execution time in Tables II and III. This is mainly because of logic-level optimization, which sometimes affects the area and the clock period, but that is not taken into account in our method.

V. CONCLUSIONS

In this paper, we proposed a behavioral partitioning method based on integer programming. Our method optimally determines functions to be implemented in the main module and ones to be in sub modules in such a way that the function-level parallelism of an input program is fully exploited and the overall datapath is minimized while keeping the complexity of individual modules within a manageable level. The proposed method achieved up to 32% reduction on the number of execution cycles compared with a traditional method.

REFERENCES

[1] D. Gajski, N. Dutt, A. Wu, and S. Lin, *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers, 1992.
[2] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii, "Function-level partitioning of sequential programs for efficient behavioral synthesis," *IEICE Trans. on Fundamentals* vol.E90-A, no.12, Dec. 2007.
[3] F. Vahid, "Partitioning sequential programs for CAD using a three-step approach," *ACM TODAES*, vol. 7, no. 3, July 2002.
[4] F. Vahid, "Procedure exlining: a transformation for improved system and behavioral synthesis," *ISSS*, 1995.
[5] F. Vahid, "Procedure cloning: a transformation for improved system-level functional partitioning," *ACM TODAES*, vol. 4, no. 1, Jan. 1999.
[6] F. Vahid, "Techniques for minimizing and balancing I/O during functional partitioning," *IEEE Trans. on CAD*, vol. 18, no. 1, Jan. 1999.
[7] M. Takahashi, N. Ishiura, A. Yamada, and T. Kambe, "Thread composition method for hardware compiler Bach maximizing resource sharing among processes," *IEICE Trans. on Fundamentals*, vol. E83-A, no. 12, Dec. 2000.
[8] SoftFloat, http://www.jhauser.us/arithmetic/SoftFloat.html.
[9] Xilinx, http://www.xilinx.com/.
[10] Y Explorations, Inc., http://www.yxi.com/.
[11] Synplicity, http://www.synplicity.com/.