関数レベル並列性を活用した動作記述分割手法

原　祐子[†]　冨山　宏之[†]　本田　晋也[†]　高田　広章[†]　石井　克哉[††]

[†] 名古屋大学大学院情報科学研究科
[††] 名古屋大学情報連携基盤センター
〒464-8603 愛知県名古屋市千種区不老町
E-mail: [†]{hara,tomiyama,honda,hiro}@ertl.jp, [††]ishii@itc.nagoya-u.ac.jp

あらまし　本論文では，大規模動作記述プログラムからハードウェアを効率良く生成する動作合成手法を提案する．本提案手法は，いくつかの並列動作可能な関数から構成されるプログラムを入力とし，関数レベルの並列性を最大限に活用しつつ，全体のデータパス面積及びコントロールパス面積を抑えるような動作記述の分割を決定する．この分割問題を整数計画問題として定式化する．実験により，本手法の有効性を示す．
キーワード　動作合成，関数レベル並列性，整数計画問題

Partitioning Behavioral Descriptions Exploiting Function-Level Parallelism

Yuko HARA[†], Hiroyuki TOMIYAMA[†], Shinya HONDA[†], Hiroaki TAKADA[†], and Katsuya ISHII[††]

[†] Graduate School of Information Science, Nagoya University
[††] Information Technology Center, Nagoya University
Furo-cho, Chikusa-ku, Nagoya, Aichi 464-8603, Japan
E-mail: [†]{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 consisting of functions which are executable in parallel, this proposed method determines an optimal behavioral-level 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.
Key words　Behavioral Synthesis, Function-Level Parallelism, Integer Programming Problem

1. Introduction

In these years, a continuous increase in the size of LSIs has brought the design productivity crisis. As a result, 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]. Although behavioral synthesis is one of the most prospective technologies to improve the design productivity, behavioral synthesis has not been widely adopted in industry yet. This is because the quality of automatically generated circuits by behavioral synthesis is inferior to that of human-designed one, which is especially serious for the synthesis 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, which results 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 n hardware modules (one main module and $n-1$ sub modules) from a program consisting of n functions. This can reduce the delay and the area of control path in individual modules, while the overall datapath may be increased since hardware resources cannot be shared between modules.

In our earlier work [2], we have proposed an N-way partitioning method based on integer programming, where N denotes the number of hardware modules. Our method in [2] optimally determines functions to be inlined into a main module and ones to be synthesized into 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 N. However, our method in [2] does not explicitly take the function-level parallelism into account, which can degrade the performance when applied to a program with functions which are executable in parallel. This paper proposes an improved method of [2] for behavioral-level partitioning. Our proposed method in this paper determines an optimal behavioral-level partitioning which exploits the function-level parallelism with simultaneously minimizing the area in the datapath and control path.

There have been several studies on behavioral-level partitioning to efficiently synthesize hardware by behavioral synthesis. In the last decade, Vahid has extensively studied behavioral-level partitioning for sequential programs [3]~[6]. The partitioning approach presented in [3] consists of three steps; procedure determination, pre-clustering and N-way partitioning. The first two steps decide the appropriate granularity of procedures (functions) using various techniques [4]~[6]. After that, traditional N-way partitioning is performed. The first two steps in [3] are useful to determine the appropriate granularity for N-way partitioning. However, his work does not take the parallelism among procedures (functions) into account. Thus, it may not exploit the parallelism of an input program where functions are executable in parallel, leading to the performance degradation of a circuit to be synthesized. Takahashi et al. have proposed a partitioning method for behavioral descriptions with processes (functions) [7]. Their work can utilize the parallelism by considering processes (functions) which are explicitly specified to be executable in parallel. However, both [3] and [7] assume that the number of modules N for functions to be partitioned into is fixed, which may not perform the optimal partitioning. As stated in [7], in many cases, it is efficient to partition functions into the smaller number of modules since as smaller the number of modules, as larger the number of hardware resources shared among functions. However, there are some cases where better circuits can be generated with the larger number of modules, which will be seen in experiments with our previous method [2] in Section 4. This partitioning cannot be obtained by [3] nor [7].

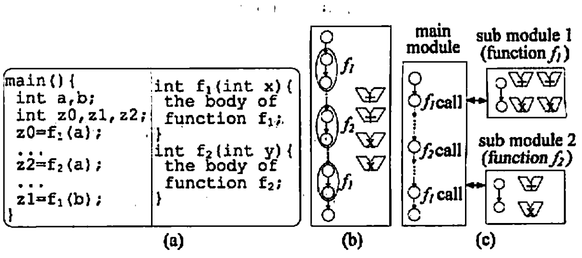
The rest of this paper is organized as follows. Section 2 describes fundamental techniques to handle functions in behavioral synthesis. Section 3 proposes a function-level partitioning method based on integer programming. Section 4 shows experimental results to demonstrate the effectiveness of the proposed method. Finally, Section 5 concludes this paper with a summary.

2. Fundamental Techniques for Behavioral Synthesis

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

2.1 Function Inlining

Function inlining replaces function calls with the bodies of the called functions. Let us consider an example program



```
main(){
 int a,b;
 int z0,z1,z2;
 z0=f1(a);
 ...
 z2=f2(a);
 ...
 z1=f1(b);
}
```

```
int f1(int x){
 the body of
 function f1;
}
int f2(int y){
 the body of
 function f2;
}
```

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

shown in Fig. 1(a). This program consists of a main function and two functions, f_1 and f_2, which are called from the main function. Fig. 1(b) shows the FSM of a circuit which is synthesized from the program in Fig. 1(a) with inlining. Note that f_1 is inlined twice since it is called twice from the main function.

Inlining has an advantage of minimizing the total datapath by sharing the resources between different functions. Assume that f_1 requires one adder and one multiplier, and f_2 requires two adders and two multipliers. Since these two functions are implemented in the same hardware module, they can share the hardware resources, that is, two adders and two multipliers are used in total. Moreover, there is no performance degradation caused by inter-module communication. Also, inlining extends operation-level parallelism and the scope of optimizations such as common sub-expression elimination, constant propagation, copy propagation, dead-code elimination and so on. However, the large number of states in the main module 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 with large functions which are called a number of times from different points of the program text.

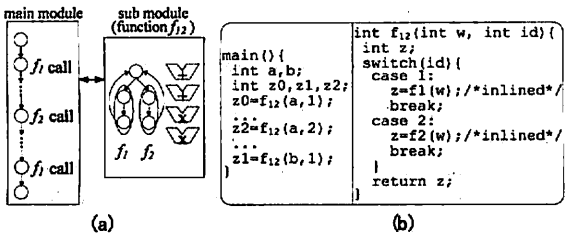
2.2 Function-Based Partitioning without Clustering

Function-based partitioning without clustering is to run behavioral synthesis for each function. This approach produces N hardware modules from a program consisting of N functions. Let us consider the same example program in Fig. 1(a). The FSM of the 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 their being sequentially called. Moreover, the inter-module communication overhead may degrade the performance.

2.3 Function-Based Partitioning with Clustering

Function-based partitioning with clustering is a method to



```
main(){
 int a,b;
 int z0,z1,z2;
 z0=f12(a,1);
 ...
 z2=f12(a,2);
 ...
 z1=f12(b,1);
}
```

```
int f12(int w, int id){
 int z;
 switch(id){
  case 1:
    z=f1(w);/*inlined*/
    break;
  case 2:
    z=f2(w);/*inlined*/
    break;
  }
  return z;
}
```

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

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 in 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 between functions. 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 circuit 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.

If too many functions are clustered when synthesizing from a program with a number of functions, however, a sub module may have the large number of states, similar to inlining. Thus, it is important to appropriately determine functions to be implemented in the main module and sub modules.

3. Partitioning Exploiting Function-Level Parallelism

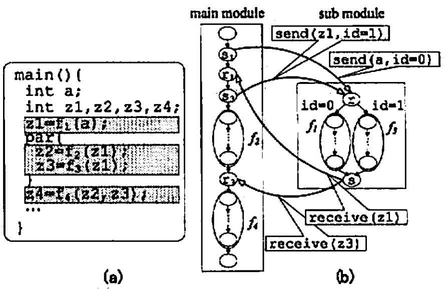
This section proposes a new behavioral-level partitioning method exploiting the function-level parallelism.

3.1 Problem Description

This section proposes an improved method of [2]. This new method optimally determines functions to be implemented in the main module and sub modules with exploiting the function-level parallelism. This problem is formulated as an integer programming. 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 the input program while keeping the complexity (i.e., the number of states) of individual modules lower than a certain level specified by a designer.

In our problem definition, the delay and area of the control path, which sometimes affect the overall critical path delay and chip area, are not explicitly taken into account. Instead, they are implicitly managed by means of the constraint on the number of states since it is known that the number of states largely affects the delay and area of the control path [8].

For simplicity, we assume that no function except the main function calls other functions and there is no global variables used in functions which are executable in parallel. Our future work will relax these assumptions. At present, for the first assumption, if a function f calls another function f', either f or f' needs to be inlined into its caller before the partitioning step. For this purpose, granularity selection techniques presented in [3] can be used.



```
main(){
 int a;
 int z1,z2,z3,z4;
 z1=f1(a);
 par{
  z2=f2(z1);
  z3=f3(z1);
 }
 z4=f4(z2,z3);
 ...
}
```

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

Let us consider an example program depicted in Fig. 3(a). This pseudo program describes function calls from a main function. We assume that a pseudo statement, par, explicitly specifies the parallel execution of functions. Namely, 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 the result of f_1 (z1) as input to f_2 and f_3. After the completion of f_2 and f_3, the results of f_2 and f_3 are written to z2 and z3, respectively. Then, function f_4 is executed by taking z2 and z3 as input.

For example, the FSM in Fig. 3(b) can be obtained by our proposed method for the program in 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 (r_1) represents the state to send (receive) data to (from) f_1 implemented in the sub module. Also, s_3 and r_3 are the same for f_3. 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. In Fig. 3(b), first, z1 and id=0, which is to select f_1, are sent from the main module at s_1 to the sub module. Then, the sub module receives data at r from the main module, and f_1 in the sub module are executed. After the completion of f_1, at r_1, the main module receives the result z1, which is sent from the sub module at s. Next, z1 and id=1, which is to select f_3, are sent from the main module at s_3 to the sub module, followed by the parallel execution of f_2 and f_3. After that, the main module receives the result z3 from the sub module at r_3 after the completion of f_2 and f_3. Finally, f_4 is executed in the main module.

3.2 Problem Formulation

This section shows the formulation of the proposed partitioning problem as an integer programming problem. First, the following notations are defined:

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 individually
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

It should be noted that s_{snd} and s_{rcv} are one in most cases, but may be more than one depending on the numbers and sizes of data to be transferred. Therefore, for generality, s_{snd} and s_{rcv} are left as parameters rather than one.

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

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

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

Next, let us explain FC_l with the 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), function f_1 is called at the first function call point, fc_0, so $FC_0 = \{f_1\}$. Next, since functions f_2 and f_3 are in a par statement, they are called at the same point fc_1, thus $FC_1 = \{f_2, f_3\}$. Finally, function f_4 is called at fc_2, so $FC_2 = \{f_4\}$.

As explained in the previous section, 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 shoud be met.

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

Then, with the notations defined above, the number of states in module m_k is estimated as follows:

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

$$S_k = \sum_i s_i \cdot x_{i,k} + (s_{snd} + s_{rcv}) \ (k = 1, \ldots, N_M - 1) \quad (3)$$

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 to the number of states in the main module. The last term $FC_{total} \cdot (s_{snd} + s_{rcv})$ denotes the total number of states for inter-module communication between the main module and sub modules. Since the main module requires one pair of s_{snd} and s_{rcv} for inter-module communication at one time, 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 follow:

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

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.

$$S_k \leqq S_{const}, (k = 0, 1, \ldots, N_M - 1) \quad (5)$$

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

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

$$A_k = \sum_j \{\max_i(x_{i,k} \cdot n_{i,j}) \cdot a_j\} + N_k^{cmp} \cdot a^{cmp}, (k = 1, \ldots, N_M - 1) \quad (7)$$

In module m_k, the required number of resource r_j is given by the maximum number among $n_{i,j}$'s for functions which are 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 given by $\sum_i x_{i,k}$ when it 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-level partitioning can be defined as an 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 the integer programming problem, a designer can obtain the optimal partitioning. To find the optimal solution, some commercial solvers for ILP can be used, or some general algorithms such as the branch and bound

Table 1 Characteristics of functions in `fft`

Hardware resources	main	add	sub	mul	div	int_to_double	sin	cos
No. of states	26	8	8	13	36	3	69	77
No. of function calls	—	3	3	5	1	2	1	1
64-bit adders		7	7	3	2	2	8	8
64-bit subtracters		1	1		1		1	1
64-bit multipliers				4	4		4	4
64-bit dividers					1		1	1
64-bit comparators	2	9	9	5	5		12	12
64-bit shifters		3	3	3	3	1	4	4
32-bit adders	3	2	2	3	3	2	4	4
32-bit subtracters		4	4	2	3	1	4	7
32-bit dividers	1							
32-bit incrementers		1	1				2	2
32-bit comparators	1	10	10	6	6	2	15	18

method, simulated annealing, the genetic algorithm, and so on, can be applied. As seen later, a simple exhaustive search algorithm was used in our experiments. Still, it yields an optimal solution within one second for realistic benchmark programs. For larger programs, however, it may be necessary to develop more efficient algorithms, which remains as one of our future works.

It should be noted that our method finds the optimal number of hardware modules as well as the optimal N-way partitioning simultaneously, by simply adding the following equation into the integer programming formulation.

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

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.

4. Experiments

We conducted a set of experiments to demonstrate the effectiveness of our partitioning approach. We used a benchmark programs `fft` from [9], which performs an Fast Fourier Transform for a matrix of double-precision floating-point numbers. This program is composed of a main function and several double-precision floating-point arithmetic functions [10]. Note that this program is rather large, which consists of approximately 900 lines of C code without including comment lines or empty lines. Characteristics of functions in `fft`, such as the number of function calls and the types and the number of hardware resources required by each function, are shown in Table 1, where double_ is omitted for some functions due to the limited space. Next, we performed behavioral-level partitioning proposed in this paper. Also, the method proposed in [2] was performed for the comparison. Then, we gave the constraint on the number of states every 20. We developed a C program to find the optimal solution for the integer programming problem defined in the previous section. Our solver is based on exhaustive search, but it took less than one second to find the optimal partitioning for each constraint.

Then, we conducted behavioral synthesis and logic synthesis to evaluate the area and clock period of the design. Xilinx Virtex 4 [11] was specified as a target device. YXI eXCite [12] and Synplicity Synplify-Pro [13] were used for behavioral synthesis and logic synthesis, respectively. All of these synthesis processes were optimized for performance maximization. Register-transfer level simulation was also performed to measure the execution cycles.

Table 2 summarizes the synthesis and simulation results with our previous method in [2] (Method-1) and the method proposed in this paper (Method-2). The first column denotes the constraint on the number of states. The results with Method-1 are described from the second to fifth columns. Also, the results with Method-2 are described from the sixth to ninth columns. Numbers in parentheses from the sixth to ninth columns are normalized values whose baseline are the results from the second to fifth columns under the same constraint. By Method-1 and Method-2, nine and five partitioning solutions were totally obtained, respectively. Function-based partitioning without and with exploiting parallelism were performed for the comparison. Note that function-based partitioning without parallelism is a technique explained in Section 2, and function-based partitioning with exploiting parallelism is its improvement. Although Method-1 obtained nine partitioning solutions, behavioral synthesis could not be completed within 24 hours when the constraint is greater than or equal to 180. The reason for the difference in the total number of partitioning solutions obtained by Method-1 and Method-2 is as follow. When it is possible to execute functions in parallel, such as double_add and double_sub in `fft`, those functions are implemented in different modules by Method-2. Thus, when formula (1) gives the constraint on the number of states greater than or equal to 160, Method-2 obtained a same solution, which minimizes formula (8) with meeting formula (5) in the previous section.

First, let us compare the results in Table 2 when the same constraint on the number of states is given to Method-1 and Method-2. In `fft`, multiple functions are called at four function call points out of 11 points. Then, by exploiting the function-level parallelism, Method-2 achieves up to 47% reduction of the execution time when the constraint is 120, and on average 26.2% reduction compared with Method-1. On the other hand, the area with Method-2 is larger than that with Methods-1 under the same constraint. This is because Method-2 implements parallel functions in different modules in order to exploit the function-level parallelism. This leads to the decreased resources sharing among functions.

Next, for results with each of Method-1 and Method-2, as the constraint becomes loose, the area tends to decrease due to the increased resource sharing, while the execution time tends to increase due to the long delay of control path. Thus, area-performance trade-off points are well described for Method-1 and Method-2. However, some exceptions can be seen on the area when the constraint is 140 with Method-

Table 2 Partitioning obtained by the method in [2] and the method proposed in this paper

Constraint on states	The method proposed in [2] (Method-1)				The method proposed in this paper (Method-2)			
	Partitioning	No. of states	Gate count	Exec. time (μs)	Partitioning	No. of states	Gate count	Exec. time (μs)

340	{main, add, sub, mul, div, int_to_double, sin, cos}	—	—	—	{main, add, int_to_double, sin}
					{sub}
					{mul, div, cos}
						137
						10
						128
							1,028,924
							(1.26)
								344.50
								(0.86)

220-320	{main} {add, sub, mul, div, int_to_double, sin, cos}	—	—	—				
200	{main, add, sub, int_to_double} {mul, div, sin, cos}	—	—	—				
180	{main, div} {add, sub, mul, int_to_double, sin, cos}	—	—	—				
160	{main} {add, sub, mul, div, int_to_double} {sin, cos}	58 70 149	813,489	400.17				

140	{main, int_to_double, cos}
	{add, sub, mul, div, sin}
		132
		136
			933,168	336.46	{main, add, sin}
					{sub, int_to_double}
					{mul, div, cos}
						138
						13
						128
							1,034,673
							(1.11)
								331.06
								(0.98)

120	{main}
	{add, sub, mul, int_to_double, sin}
	{div, cos}
		58
		103
		115
			991,809	448.54	{main, add, int_to_double}
					{sub, div, sin}
					{mul, cos}
						68
						115
						92
							1,067,732
							(1.08)
								236.51
								(0.53)

100	{main, div}
	{add, sub, mul, sin}
	{int_to_double, cos}
		92
		100
		82
			1,049,861	345.68	{main}
					{add, int_to_double, sin}
					{sub, cos}
					{mul, div}
						48
						82
						87
						51
							1,067,716
							(1.02)
								235.97
								(0.68)

80	{main}
	{add, sub, mul, div, int_to_double}
	{sin}
	{cos}
		58
		70
		71
		79
			1,087,182	338.64	{main, add, int_to_double}
					{sub, sin}
					{mul, div}
					{cos}
						68
						79
						51
						79
							1,174,695
							(1.08)
								216.81
								(0.64)

Constraint on states	Function-based partitioning without exploiting parallelism				Function-based partitioning with exploiting parallelism			
	Partitioning	No. of states	Gate count	Exec. time (μs)	Partitioning	No. of states	Gate count	Exec. time (μs)

—	{main}
	{add}
	{sub}
	{mul}
	{div}
	{int_to_double}
	{sin}
	{cos}
		58
		10
		10
		15
		38
		5
		71
		79
			1,178,844	325.68	{main}
					{add}
					{sub}
					{mul}
					{div}
					{int_to_double}
					{sin}
					{cos}
						48
						10
						10
						15
						38
						5
						71
						79
							1,172,333
							(0.99)
								218,78
								(0.67)

1 and 100 with Method-2, and on the execution time when the constraint is 120 and 140 with Method-1. One reason for the exceptions is logic-level optimization, which sometimes affects the area and the clock period, but that is not taken into account in our proposed methods.

The experimental results in Table 2 demonstrate the effectiveness of the partitioning method proposed in this paper. This method more efficiently determines the optimal partitioning which fully exploits the function-level parallelism with simultaneously minimizing the overall area compared with our previous method in [2].

5. Conclusions

In this paper, we have proposed a behavioral-level partitioning method based on integer programming. Our method optimally determines functions to be inlined into the main module and ones to be synthesized into 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. Experimental results demonstrate that the proposed partitioning method enables efficient behavioral synthesis from a large behavioral description.

Acknowledgements

This work is in part supported by KAKENHI 19700040.

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, pp.2853-2862, Dec. 2007.

[3] F. Vahid, "Partitioning Sequential Programs for CAD Using a Three-Step Approach," *ACM Trans. on Design Automation of Electronic Systems*, vol. 7, no. 3, pp. 413 - 429, July 2002.

[4] F. Vahid, "Procedure Exlining: A Transformation for Improved System and Behavioral Synthesis," *International Symposium on System Synthesis*, 1995.

[5] F. Vahid, "Procedure Cloning: A Transformation for Improved System-Level Functional Partitioning," *ACM Trans. on Design Automation of Electronic Systems*, vol. 4, no. 1, pp. 70 - 96, Jan. 1999.

[6] F. Vahid, "Techniques for Minimizing and Balancing I/O during Functional Partitioning," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 1, pp. 69 - 75, 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, pp.2456 - 2463, Dec. 2000.

[8] G. R. Gupta, M. Gupta, and P. R. Panda, "Rapid Estimation of Control Delay from High-Level Specifications", *Design Automation Conference*, pp. 455 - 458, 2006.

[9] SNU Real-time Benchmarks, http://archi.snu.ac.kr/realtime/benchmark/.

[10] SoftFloat, http://www.jhauser.us/arithmetic/SoftFloat.html.

[11] Xilinx, http://www.xilinx.com/.

[12] Y Explorations, Inc., http://www.yxi.com/.

[13] Synplicity, http://www.synplicity.com/.