IEICE TRANS. FUNDAMENTALS, VOL.E93–A, NO.2 FEBRUARY 2010

PAPER

Partitioning of Behavioral Descriptions with Exploiting Function-Level Parallelism

Yuko HARA[†*a)], ***Nonmember***, **Hiroyuki TOMIYAMA**[†], ***Member***, **Shinya HONDA**[†], ***Nonmember***, ***and*** **Hiroaki TAKADA**[†], ***Member***

SUMMARY A novel method to efficiently synthesize hardware from a large behavioral description in behavioral synthesis is proposed. For a program with functions executable in parallel, this proposed method determines a behavioral partitioning which simultaneously minimizes the overall datapath area and the complexity of the controller while maximizing performance of a synthesized circuit by fully exploiting function-level parallelism of a behavioral description. This method is formulated as an integer programming problem. Experimental results demonstrate that this method leads to a shift of the explorable design space so that superior solutions which could not be explored by earlier work are included, showing the effectiveness of our proposed method.

key words: *behavioral synthesis, function-level partitioning, integer programming problem*

1. Introduction

Behavioral synthesis is a technology to automatically synthesize an Register-Transfer Level (RTL) circuit from a behavioral description written in a high-level programming language such as C. These years, behavioral synthesis has been expected as a promising LSI design alternative to the traditional RTL design with Hardware Design Languages (HDLs) [1]. So far, a number of behavioral synthesis tools have been developed in both academia and industry. At present, nevertheless, many LSI designers hesitate to roll over to behavioral synthesis and still start the LSI design with HDLs. This is because the quality of automatically generated circuits in behavioral synthesis is inferior to that of human-designed ones, which is especially serious for the synthesis from large behavioral descriptions.

Behavioral partitioning, which is a prior step to behavioral synthesis, is one of solutions for the above problem. Partitioning an input behavioral description into multiple smaller segments and running behavioral synthesis for each segment can yield various advantages such as area reduction, performance improvement, synthesis runtime reduction, packaging or I/O constraints satisfaction, and power/energy consumption reduction [2]. Many studies on behavioral partitioning have been presented in the last two decades. Most of earlier studies focused on multi-chip partitioning techniques for mapping a single system on multiple chips [3]–[7]. As the capacity of a chip grows, however, behavioral partitioning for improving performance, area, power/energy, and so on has become a main concern of designers and has been studied in various work these days [2], [8]–[11].

Large behavioral descriptions written in C[**] generally consist of a number of functions. Thus, if designers need to partition such behavioral descriptions, it is natural to partition them at the function level granularity. Actually, many C-based behavioral synthesis tools such as SPARK [12], CCAP [13], eXcite [14], and Cyber [15] have a synthesis option to partition an input program at the function level. It generates one hardware module from one function. That is to say, if applied to all the functions, n hardware modules are produced from a program consisting of n functions. This technique can reduce the delay and the area of the controller in individual modules, while the overall datapath area may become large since hardware resources cannot be shared between modules. On the other hand, functions to which the above technique is not applied are inlined into their callers. Inlining can reduce the datapath area by resource sharing among functions, while the delay and the area of the controller might be increased due to the complicated controller. This disadvantage becomes critical especially in case of a program where large functions are called a number of times from different points of the program text. Another technique employed by some behavioral synthesis tools such as Bach [8] clusters some functions in a same sub module after partitioning the input programs at the function level. This technique can reduce the datapath area by resource sharing among functions clustered in the same module, and can also reduce the delay and the area of the controller in individual modules by diminishing the complexity of the controller. If too many functions are clustered in a same sub module when synthesizing from a program with a number of functions, however, the controller of the sub module may become complex, similar to inlining. As explained above, each of the three techniques has both advantages and disadvantages. Therefore, it is crucial to utilize the features of the three techniques and determine an appropriate technique for each function.

Manuscript received June 23, 2009.
Manuscript revised September 25, 2009.

[†]The authors are with the Graduate School of Information Science, Nagoya University, Nagoya-shi, 464-8603 Japan.
[*]The author is a research fellow of the Japan Society for the Promotion of Science.
a) E-mail: hara@ertl.jp

DOI: 10.1587/transfun.E93.A.488

[**]This paper assumes the C language as an input language to behavioral synthesis because nowadays C-based behavioral synthesis has become the most popular.

In our earlier work [10], we have proposed a function-level partitioning method based on integer programming. The method employs the three techniques and determines a function-level partitioning in such a way that the overall datapath is minimized while diminishing the complexity of the controller in individual modules. However, function-level parallelism is not considered in the method. Even when the method is applied to a program with functions executable in parallel, these functions are sequentially executed. This misses chances to improve performance.

This paper proposes an improved method for function-level partitioning. This work significantly improves our previous work [10] in explicitly taking function-level parallelism of a program into account. This extension leads to a shift of the explorable design space so that superior solutions which could not be explored by our previous work [10] are included.

This method determines a function-level partitioning in such a way that the overall datapath area and the complexity of the controller in individual modules are simultaneously minimized while maximizing performance of a synthesized circuit by fully exploiting function-level parallelism specified by designers. Although various work also studied similar behavioral partitioning methods such as in [8] and [9], our proposed method outperforms them in exploring the better area-performance trade-off points since our method employs the three techniques to handle functions, while to the best of our knowledge no work integrates the three. This is the first behavioral partitioning work considering function-level parallelism and intelligently utilizing the three techniques to handle functions.

The rest of this paper is organized as follows. First, Sect. 2 explains fundamental techniques to handle functions in behavioral synthesis and discusses related work. Next, Sect. 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, Sect. 5 concludes this paper with a summary and future work.

2. Existing Techniques

Large C programs, which generally consist of a number of functions, are naturally partitioned at the function-level granularity for behavioral partitioning. There are three fundamental techniques to handle functions in behavioral synthesis; *function inlining*, *function-based partitioning without clustering*, and *function-based partitioning with clustering*.

This section, first, explains these three techniques and shows their advantages and disadvantages. Next, related work on behavioral partitioning is discussed.

2.1 Function Inlining

Function inlining is a well-known compiler optimization technique which replaces function calls with the bodies of the callee functions. Inlining is also widely adopted for behavioral synthesis.

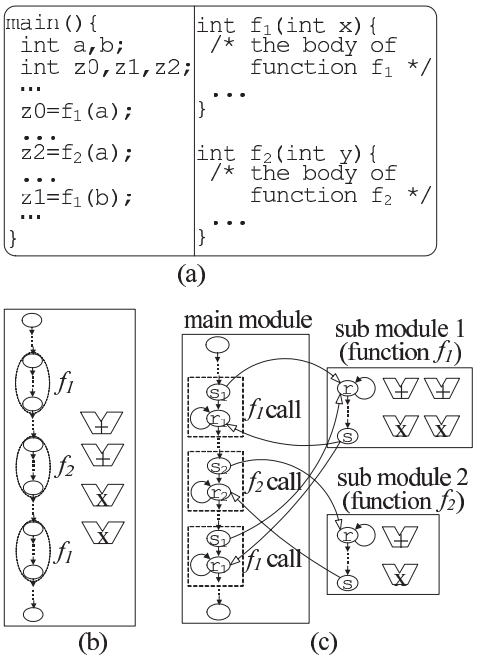

Fig. 1 Traditional techniques: (a) An example program, (b) Function inlining, (c) Function-based partitioning without clustering.

Let us consider an example program 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. Figure 1(b) shows the FSM of a circuit synthesized from the program in Fig. 1(a) by inlining f_1 and f_2. Note that f_1 is inlined twice since it is called from two different points in the main function on the program text.

Inlining has several advantages. First, inlining enables resource sharing among multiple functions, resulting in the small datapath area. Assume that f_1 requires two adders and two multipliers, and f_2 requires one adder and one multiplier. Since the two functions are implemented in the same hardware module, they can share the functional units, that is, two adders and two multipliers are required in total. Second, no overhead of inter-module communication is necessary. Furthermore, inlining extends the scope of optimizations such as common sub-expression elimination, constant propagation, copy propagation, dead-code elimination and so on. Finally, inlining also extends operation-level parallelism.

Inlining, however, increases both the delay and the area of the controller since the number of states in the main module becomes large[†]. Also, inlining may increase the datapath delay due to multiplexers inserted at the inputs of functional units and registers for sharing. At worst, if the main function after inlining the callee functions becomes too large, behavioral synthesis may not be completed within a practical time. These disadvantages become critical especially in

[†]The number of states in a module has a correlation with the complexity of its controller [16].

case of programs where large functions are called a number of times from different points on the program text.

2.2 Function-Based Partitioning without Clustering

Function-based partitioning without clustering is a technique to generate one hardware module from one function. If applied to all the functions, N hardware modules (one main module and $N-1$ sub modules) are produced from a program consisting of N functions. Let us consider the same example program in Fig. 1(a). The FSM of a circuit synthesized with this technique 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 on the program text.

The main advantage of this technique is that it can reduce the complexity of the controller in individual modules, leading to the short delay and the small area of the controller. On the other hand, this technique has a disadvantage of increasing the overall datapath area since resources are not shared between multiple functions even when the same types of resources are required by the functions. For example, in Fig. 1(c), three adders and three multipliers are used in total even though f_1 and f_2 are sequentially executed. Another disadvantage of this technique is that inter-module communication may degrade performance.

2.3 Function-Based Partitioning with Clustering

Function-based partitioning with clustering is a technique to cluster some functions in one sub module instead of individually synthesizing sub modules. Figure 2(a) shows the FSM of a circuit generated by clustering f_1 and f_2 of Fig. 1(a) in a single sub module. It suppresses the number of states in the main module. Also, the sub module minimizes the datapath area by resource sharing between the 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 and the complexity of the controller in individual modules.

Function clustering can be achieved by transforming the original program shown in Fig. 1(a) into the one in Fig. 2(b). A function f_{12} is newly defined, which calls either f_1 or f_2 depending on a parameter *id*. When synthesizing from f_{12}, f_1 and f_2 are both inlined into f_{12}, while f_{12} itself is not inlined into the main function.

As shown above, with clustering, the total datapath area can be reduced by resource sharing between multiple functions, and the control path area can be also reduced by suppressing the complexity of the controller in individual modules. If too many functions are clustered in a same sub module when synthesizing from a program with a number of functions, however, the controller of the sub module may become complex, similar to inlining. Thus, it is crucial to appropriately determine functions to be implemented in the main module and ones to be in sub modules.

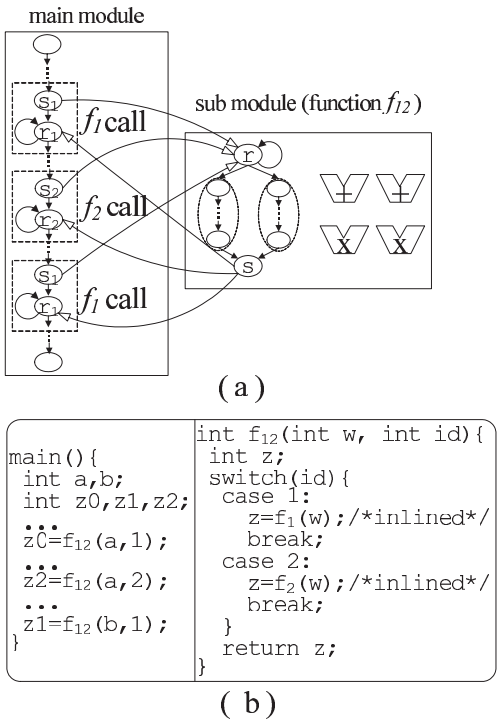

Fig. 2 Clustering: (a) Partitioning with clustering, (b) A refined program for clustering.

2.4 Related Work on Behavioral Partitioning

This section discusses various work on behavioral partitioning methods [2], [8]–[11].

Takahashi et al. studied a function-level partitioning method which minimizes the overall datapath area while suppressing the complexity of the controller [8]. Two function-based partitioning techniques (i.e., function-based partitioning without clustering and function-based partitioning with clustering) are employed. Their work considers an input program's function-level parallelism, which is explicitly specified on the program text by designers. Jasrotia et al. proposed a similar method focusing on loop-level partitioning but not function-level partitioning [9]. For sequential programs, the method employs two techniques to handle loops; *inlining* and *exlining*. If a loop is decided to be inlined, the body of the loop is expanded in a main module, while if decided to be exlined, the loop is extracted as a sub module. The goal of the method is to reduce the complexity of the controller in individual modules in order to implement a circuit with the low power controller. Also, we presented a method which minimizes the overall datapath area while diminishing the complexity of the controller in individual modules [10]. With utilizing the three aforementioned techniques (i.e., function inlining, function-based partitioning without clustering, and function-based partitioning with clustering), the method simultaneously optimizes the partitioning and the number of modules[†].

[†]In Sect. 2.5, we will discuss the method in more detail.

However, these studies might be unable to find a good partitioning for the following three restrictions. First, [8] and [9] employ only two techniques to handle functions or loops out of the three, i.e., inlining and clustering are not employed in [8] and [9], respectively. They cannot exploit the advantages of the unemployed technique. This narrows the design space to be explored. Next, the number of modules is predetermined in [8] and [9]. The number of modules and the partitioning should be simultaneously determined. They also narrows the design space. Finally, [9] and [10] do not consider the parallelism among loops and functions, respectively. Even when applied to a program with the parallelism among loops or functions, [9] and [10] miss chances to improve performance. Our proposed method in this paper can explore the wider design space than these studies because our method simultaneously optimizes the number of modules and the partitioning by employing the three techniques and fully exploiting function-level parallelism.

Vahid extensively studied behavioral partitioning for large sequential programs [2], [17]–[19]. [2], first, decides the appropriate granularity of procedures (functions) using various techniques in [17]–[19], and then performs a traditional N-way partitioning. Uchida et al. developed a partitioning method for low power behavioral synthesis [11]. After threads (modules) are generated according with a traditional N-way partitioning, their proposed two-way partitioning is conducted for those threads. This two-way partitioning divides a thread into two; one has a local register file (RF), and the other does not have an RF. Then, a gated clock is applied to each thread for minimizing the total power consumption. These methods are orthogonal to our proposed method. [2] and [11] can be integrated with our proposed partitioning method as the preprocessing and postprocessing methods, respectively.

2.5 Our Previous Work

We have presented a function-level partitioning method for large sequential programs [10]. With employing the three fundamental techniques explained above, our previous work optimally determines functions to be inlined into the 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 controller in individual modules lower than a certain level. This is formulated as an integer programming problem.

Let us consider an example program depicted in Fig. 3(a). This program describes function calls from a main function. Assume that the method [10] obtained a partitioning where function f_1 is inlined into the main module, functions f_2 and f_3 are clustered in a sub module, and function f_4 is implemented in a different sub module from the one where f_2 and f_3 are clustered. Figure 3(b) shows the FSM of a synthesized circuit based on the obtained partitioning. Black and white arrows in Fig. 3(b) represent state transition and inter-module communication, respectively. Function clustering can be implemented by transforming the program in Fig. 3(a) into the one shown in Fig. 3(c), where a newly defined function f_{23} calls either f_2 or f_3 depending on a parameter `id`. f_2 and f_3 are inlined into f_{23}, while f_{23} itself is not inlined into the main function.

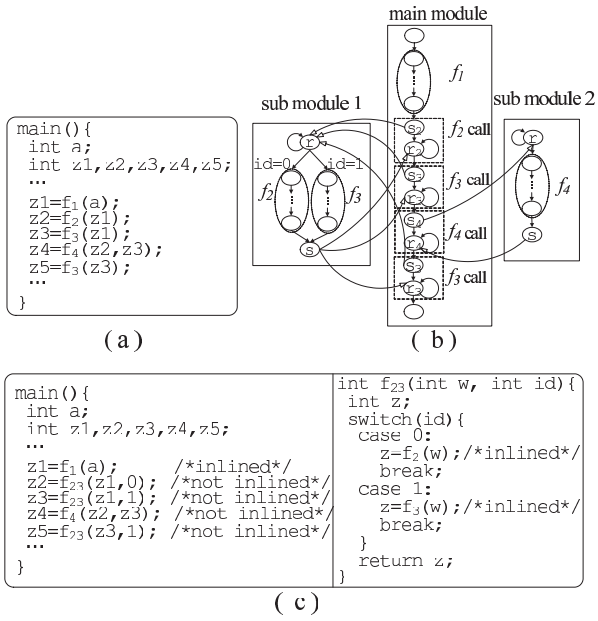

Fig. 3 N-way partitioning for sequential programs: (a) An example program, (b) A partitioning example, (c) A refined program for the partitioning example described in Fig. 3(b).

With the method [10], designers can explore the performance-area trade-off points before conducting behavioral synthesis in actual. However, the method does not consider function-level parallelism of input behavioral descriptions, which misses chances to improve performance. For example, in Fig. 3(a), functions f_2 and f_3 have no dependency each other, that is, they are executable in parallel. Since they are implemented in the same sub module in Fig. 3(b), however, they are sequentially executed. In case of the program in Fig. 3(a), f_2 and f_3 should be implemented in different modules so that they can be executed in parallel for achieving better performance. Therefore, function-level parallelism should be explicitly taken into account for maximizing performance of the circuit.

3. Partitioning Exploiting Function-Level Parallelism

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

3.1 Problem Description

We propose a function-level partitioning method to determine functions to be implemented in the main module and ones to be in sub modules with fully exploiting function-level parallelism of an input behavioral description. This method is formulated as an integer programming problem. Our goal is to minimize the overall datapath area (i.e., the

total cost of functional units) while keeping the complexity of the controller (i.e., the number of states) in individual modules lower than a certain level specified by designers and minimizing the number of execution cycles.

Our problem definition is based on an observation in [16] that the number of states often largely affects the overall critical path delay (i.e., the clock period). An increase in the number of states generally leads to the longer delay of controller, causing to a longer period, that is, the clock period is positively-correlated with the number of states†.

In addition, we try to minimize the number of execution cycles for any partitioning solutions by fully exploiting potential parallelism of the input program (i.e., both function-level parallelism and operation-level parallelism) as follows; Function-level parallelism is maximized by executing in parallel functions which have no dependency with each other; Operation-level parallelism within a function is maximized by giving the sufficient numbers of functional units. Thus, the number of execution cycles does not significantly differ between the partitioning solutions.

In general, performance of a circuit is determined by the clock period and the number of execution cycles. In our approach, depending on the partitioning solutions, the clock period largely differs, while the number of execution cycles does not significantly differ. Consequently, in our method, the number of states is used as a measure of performance and is given as a constraint by designers.

For simplicity, in this method, we assume that functions executable in parallel have no dependency with each other. Such functions are specified by designers, as well as done in [8]. Also, we assume that no function except the main function calls other function. At present, if a function f calls another function f', either f or f' needs to be inlined into its caller before our proposed partitioning. For this purpose, the granularity selection techniques presented in [2] can be used. Our future work will relax these assumptions.

Let us consider an example program depicted in Fig. 4(a). This program describes function calls from a main function. We assume that a pseudo statement, `par`, explicitly specifies the parallel execution of functions in the same `par` statement. In Fig. 4(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 (i.e., `z1`) as input. 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.

Assume that for the program in Fig. 4(a), our proposed method obtained a partitioning solution under a certain constraint, where f_2 and f_4 are in the main module, and f_1 and f_3 are in a same sub module. Figure 4(b) shows the FSM of a synthesized circuit based on the obtained partitioning. In Fig. 4(b), f_2 and f_4 are inlined into the main module, while f_1 and f_3 are clustered in the sub module. Black and white arrows in Fig. 4(b) represent state transition and inter-module communication, respectively. State s_1 (or r_1) represents a state to send (or receive) data to (or from) f_1 implemented in the sub module. State s_3 (or r_3) is also the same for f_3. Note that f_2 and f_3 are implemented in different modules since the `par` statement in Fig. 4(a) explicitly specifies that they are executable in parallel. In Fig. 4(b), first, `a` and `id=0` (a parameter to select f_1) are sent at s_1 from the main module to the sub module. Then, the sub module receives those data at r, and f_1 in the sub module is executed. After the completion of f_1, the main module receives the result of f_1 (i.e., `z1`) from the sub module. Next, similarly, `z1` and `id=1` (a parameter to select f_3) are sent at s_3 from the main module to the sub module, followed by the parallel execution of f_2 in the main module and f_3 in the sub module. After the completion of both f_2 and f_3, at r_3, the main module receives the result of f_3 (i.e., `z3`) from the sub module. Finally, f_4 is executed in the main module.

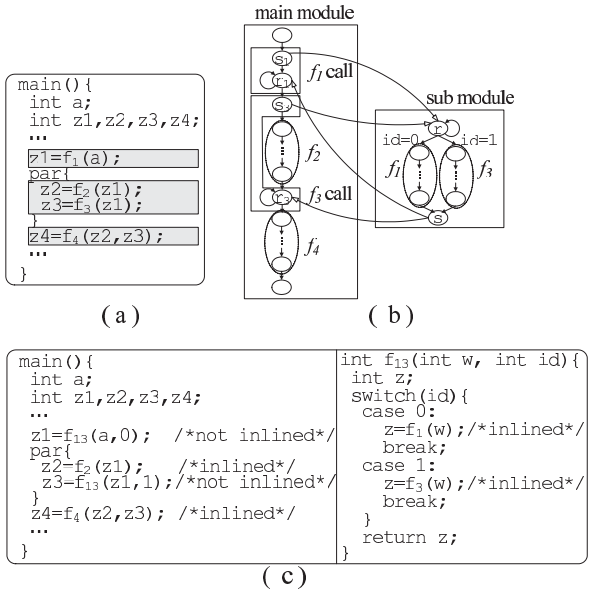

Fig. 4 An example program with functions which are executable in parallel: (a) An example program, (b) A partitioning example, (c) A refined program for the partitioning example described in Fig 4(b).

The circuit in Fig. 4(b) is synthesized from a program in Fig. 4(c), which is transformed from the one in Fig. 4(a). A function f_{13} is newly defined, which calls either f_1 or f_3 depending on a parameter *id*. Note that in Fig. 4(c) f_2 and f_4 are inlined into the main function, while f_1 and f_3 are clustered in f_{13}, which itself is not inlined into the main function.

3.2 Problem Formulation

We formulate the proposed partitioning problem as an integer programming problem. Notations are defined in Table 1. First, let us explain FC_l with the example program in Fig. 4(a). In case of the program in Fig. 4(a), there are totally three function call points, which are described as gray boxes in Fig. 4(a). Note that all the functions belonging to

†The number of operations is another key factor to influence on the complexity of the controller. By a simple extension of our method proposed in this paper, not only the number of states but also the number of operations can be given as the constraint, as well as done in [10].

Table 1 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 a 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 that function f_i is called in the program text Note that c_0 is 1.
$n_{i,j}$	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 for sending data in inter-module communication
s_{rcv}	The number of states for receiving data in inter-module communication
N_M	The number of hardware modules
m_k	Hardware module ($k = 0, 1, \ldots, N_M - 1$) 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 in 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 comparators required in module m_k
a^{cmp}	The area of a comparator

a same `par` statement are called at the same function call point. In Fig. 4(a), f_1 is called at the first function call point, fc_0, so $FC_0 = \{f_1\}$. Next, since f_2 and f_3 are in a same `par` statement, they are both called at the same point fc_1, thus $FC_1 = \{f_2, f_3\}$. Finally, f_4 is called at fc_2, so $FC_2 = \{f_4\}$.

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

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

where $\sum_k x_{i,k} = 1$.
Note that $x_{0,0}$ is 1 since the main function f_0 is implemented in the main module m_0.

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. 4(a), should be implemented in different modules. Thus, the next formula should 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, the number of states in module m_k is estimated as follows:

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

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

Formula (2) estimates the number of states in the main module m_0. The first term represents that for an inlined function f_i, its number of states s_i multiplied by its number of time being called from the main function c_i is added. The second term, $FC_{total} \cdot (s_{snd} + s_{rcv})$, denotes the total number of states for inter-module communications with sub modules. Since one pair of the states for sending and receiving data (i.e., $s_{snd} + s_{rcv}$ states) is necessary for one inter-module communication on the program text, FC_{total} pairs of them (i.e., $FC_{total} \cdot (s_{snd} + s_{rcv})$ states) are necessary 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:

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

Namely, y_l is 1 when the main module has inter-module communication with at least one sub module at the function call point FC_l, otherwise 0.

Formula (3) estimates the number of states in sub module m_k. The last term ($s_{snd} + s_{rcv}$) denotes the number of states for inter-module communication with the main module.

Here, the number of states in each module cannot exceed the limit specified by designers. 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.

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

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

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 in module m_k. A sub module which implements more than one function requires comparators to determine the function to be executed depending on a parameter *id*. In module m_k, 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 an integer programming, which finds $x_{i,k}$ minimizing formula (8) with meeting the

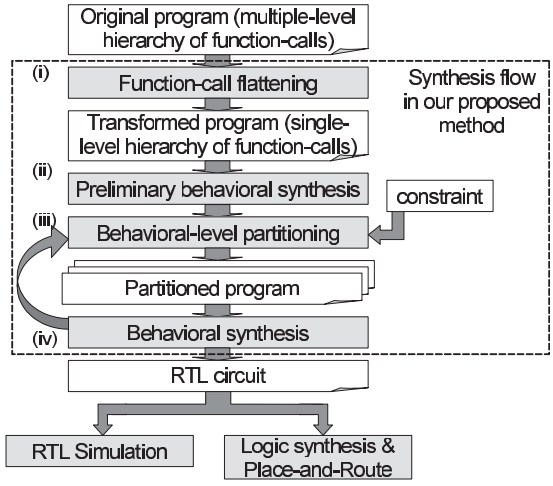

Fig. 5 Overall synthesis flow.

constraints in formulas (1) and (5). By finding such $x_{i,k}$s, designers can obtain a partitioning solution. For solving this problem, 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.

Here, it should be noted that our method simultaneously optimizes the number of modules and the N-way partitioning by simply adding the following equation to the above integer programming formulation.

$$N_M = N_F \tag{9}$$

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_ks.

3.3 Overall Synthesis Flow

Figure 5 displays the steps for obtaining a partitioning solution with using our proposed method. Our proposed method consists of four steps; function-call flattening, preliminary behavioral synthesis, behavioral partitioning, and behavioral synthesis of the entire design.

Step (i): As mentioned in Sect. 3.1, the multiple-level hierarchy of function calls is flattened into a single level by function inlining. Thus, if a function f calls another function f', either f or f' needs to be inlined into its caller. For this purpose, the granularity selection techniques presented in [2] can be used.

Step (ii): For each function f_i in a transformed program at Step (i), behavioral synthesis is executed in order to obtain the numbers and types of hardware resources (i.e., $n_{i,j}$) and the number of states (i.e., s_i). Also, function-level parallelism is examined.

Step (iii): Designers give the constraints on function-level parallelism and the number of states (i.e., S_{const}) to the integer programming problem presented in Sect. 3.2. Then, it obtains a partitioning solution, based on which the program is partitioned.

Step (iv): Behavioral synthesis is conducted for each cluster obtained at Step (iii), and an RTL circuit of the entire design is realized. This RTL circuit is input to the RTL simulation and lower-level synthesis (i.e., logic synthesis and place-and-route) in order to measure the execution cycles and the circuit performance (the area and clock period), respectively.

One problem at Step (iii) is how the designers should decide the constraint on the number of states. The number of states is affected by several factors such as the ability of the behavioral and logic synthesis tools, and the clock frequency constraint, so finding the solution at once is not an easy problem. Therefore, the synthesis Steps (iii) and (iv) need to be repeated with varying the constraints on the number of states. The key point is that our proposed method in this paper can be used as a tool which significantly efficiently explores the design space. For each constraint, our method finds a partitioning solution without actually synthesizing the circuits, so the designers do not have to run behavioral/logic synthesis for all the possible partitionings.

3.4 Limitations

It should be noted that our proposed method does not obtain an optimal solution in an exactly mathematical sense. The integer programming formulation in Sect. 3.2 does not take into account the influences of optimizations in behavioral and logic synthesis, which may cause the difference between the estimations of the number of states and area and the actual results. Some examples of such possible influences are shown as follows. With function inlining, the actual number of states may differ from the estimation by formula (2) because of various optimizations in behavioral synthesis beyond the boundary of functions. Similarly, with function clustering, the estimation by formula (3) may be inaccurate since some states can be shared between multiple functions clustered in the same sub module. Also, such optimizations can increase or decrease functional unit requirements, which causes the inaccurate estimation of the total area of functional units by formulas (6) and (7). Although the number of execution cycles is not assumed to differ between the partitioning solutions, it may slightly differ, e.g., because of the above-mentioned influences of optimizations on the formulas (2) and (3). Moreover, even if the estimation were accurate, the area and clock period might vary due to optimizations in logic synthesis and place-and-route. These influences should be carefully considered in our future work.

Even though our integer programming formulation does not necessarily obtain an optimal solution, our proposed method is useful because, to the best of our knowledge, this is the first work which intelligently applies the three techniques (i.e., function inlining, function based-partitioning without clustering, and function-based parti-

tioning with clustering) to functions. Thus, our proposed method presents a significant contribution by exploring superior solutions which could not be obtained by earlier work, which is demonstrated through experiments in Sect. 4.

4. Experiments

In this section, we show the effectiveness of our proposed method through experiments.

4.1 Experimental Setup

We conducted three sets of experiments to demonstrate the effectiveness of our partitioning method. We used three benchmark programs [20], `fft`, which performs a Fast Fourier Transform for a matrix of double-precision floating-point numbers, `lms`, which performs a Least Mean Squares adaptive signal enhancement for a given matrix of double-precision floating-point numbers, and `gaussian`, which performs a Gaussian filter for a set of given double-precision floating-point numbers. These programs are composed of a main function and several double-precision floating-point arithmetic functions [21], [22]. Note that `fft`, `lms`, and `gaussian` are rather large, which consist of more than 800, 700, and 600 lines of C code, respectively, without including comment or empty lines.

We followed the synthesis flow described in Sect. 3.3. The numbers of functions in `fft`, `lms`, and `gaussian` after function-call flattening at Step (i) in Fig. 5 are eight, six, and six, respectively. Also, `fft`, `lms`, and `gaussian` have three, four, and four function-call points where multiple functions are executable in parallel out of six, 11, and seven, respectively. We implemented a C program to solve the algorithm described in Sect. 3.3 based on the exhaustive search. The constraint on the number of states was given every 25. Although the solution space exponentially increases with increasing the number of functions, a partitioning solution was obtained for each constraint within several seconds for `fft` and within 1 second for `lms` and `gaussian` in CPU time. Xilinx Virtex-4 XC4VLX200 [23] was specified as a target device. Its package and speed grade were set FF1513 and -11, respectively. eXCite [14] was used for behavioral synthesis. The following synthesis options were given to eXCite; No resource constraint was specified to minimize the number of execution cycles; Each functional unit completes in a single clock cycle. Synplify-Pro [24] and XST [23] were used for logic synthesis and place-and-route, respectively. All the synthesis processes were optimized for performance maximization. RTL simulation was also performed to measure the execution cycles.

For the comparison, we conducted experiments under the following five synthesis methods including our proposed method;

INLINE: Function inlining
FBPAR: Function-based partitioning without clustering

Table 2 Traditional synthesis methods for `fft`.

Synthesis method	No. of modules	No. of states	Exec. cycles	Clock period (ns)	Exec. time (μs)	Area (LUTs)
INLINE	1	—	—	—	—	—
FBPAR	8	59 / 32 / 32 / 25 / 55 / 10 / 130 / 159	10,930	33.42	365.28	150,186
FBPAR-PE	8	49 / 32 / 32 / 25 / 55 / 10 / 130 / 159	7,441	31.84	236.88	151,417

Table 3 Earlier partitioning method (FBPAR-CLUS) for `fft`.

Constraint on states	No. of modules	No. of states		Exec. cycles	Clock period (ns)	Exec. time (μs)	Area (LUTs)	CPU time (s)
700-	1	Est.	676	—	—	—	—	8.89
		Act.	—					
450-675	2	Est.	59 / 431	—	—	—	—	9.33
		Act.	—					
375-425	2	Est.	239 / 363	—	—	—	—	9.97
		Act.	—					
350	2	Est.	227 / 347	10,754	39.65	426.44	111,790	10.92
		Act.	199 / 347					
325	2	Est.	290 / 310	10,832	40.77	441.60	120,953	11.17
		Act.	291 / 310					
300	3	Est.	59 / 146 / 287	10,930	39.98	436.98	104,802	11.00
		Act.	59 / 146 / 287					
250-275	2	Est.	248 / 242	10,880	32.39	352.40	111,386	10.10
		Act.	241 / 242					
225	3	Est.	59 / 219 / 214	10,930	33.80	369.40	116,199	9.66
		Act.	59 / 219 / 214					
200	4	Est.	59 / 70 / 182 / 183	10,930	27.80	303.89	126,231	8.89
		Act.	59 / 70 / 182 / 183					
175	4	Est.	59 / 146 / 130 / 159	10,930	31.27	341.80	133,306	6.92
		Act.	59 / 146 / 130 / 159					

Table 4 Our proposed method (FBPAR-CLUS-PE) for `fft`.

Constraint on states	No. of modules	No. of states		Exec. cycles	Clock period (ns)	Exec. time (μs)	Area (LUTs)	CPU time (s)
350-	3	Est.	328 / 32 / 182	7,385	36.72	271.14	128,099	8.23
		Act.	321 / 32 / 182					
300-325	3	Est.	277 / 32 / 235	7,392	35.95	265.74	129,175	8.11
		Act.	273 / 32 / 235					
225-275	3	Est.	149 / 189 / 206	7,398	35.85	265.21	137,936	7.48
		Act.	145 / 189 / 206					
200	4	Est.	149 / 32 / 182 / 183	7,398	35.60	263.37	140,790	6.92
		Act.	145 / 32 / 182 / 183					
175	4	Est.	149 / 160 / 78 / 159	7,398	34.44	254.81	147,678	5.22
		Act.	145 / 160 / 78 / 159					

Table 5 Traditional synthesis methods for `lms`.

Synthesis method	No. of modules	No. of states	Exec. cycles	Clock period (ns)	Exec. time (μs)	Area (LUTs)
INLINE	1	—	—	—	—	—
FBPAR	6	37 / 32 / 32 / 25 / 55 / 55	2,539	19.31	49.02	79,125
FBPAR-PE	6	29 / 32 / 32 / 25 / 35 / 55	2,092	19.91	41.66	78,590

FBPAR-PE: Function-based partitioning without clustering which is integrated with the parallel execution of functions

FBPAR-CLUS: A partitioning method employing function-based partitioning with clustering [10] (also shown in Sect. 2.5)

FBPAR-CLUS-PE: Our proposed partitioning meth-od fully exploiting function-level parallelism

4.2 Experimental Results

For `fft`, Tables 2, 3, and 4 summarize experimental results of traditional synthesis methods (INLINE, FBPAR, and FBPAR-PE), FBPAR-CLUS [10], and FBPAR-CLUS-PE (i.e., our proposed method), respectively. In Table 2, applied traditional synthesis methods are shown in the first columns. In Tables 3 and 4, given constraints on the number of states are specified in the first columns. For each method/solution, the number of modules, the numbers of states in modules, the number of execution cycles, the clock period (ns), the execution time (μs), and area (LUTs) are described in from the second to seventh columns of Tables 2, 3, and 4. In the third columns, the numbers of states in the main modules are shown in the leftmost. Also, the third columns in Tables 3 and 4 describe the estimated and actual numbers of states in modules in the upper (Est.) and lower (Act.) rows, respectively. CPU time to solve the integer programming problem is shown in the last columns of Tables 3 and 4. Similarly, Tables 5–9, and 10 summarize experimental results of the three traditional synthesis methods for `lms`, FBPAR-CLUS for `lms`, FBPAR-CLUS-PE for `lms`, the three traditional synthesis methods for `gaussian`, FBPAR-CLUS for `gaussian`, and FBPAR-CLUS-PE for `gaussian`, respectively.

With INLINE, behavioral synthesis could not be completed within 24 hours for `fft` and `lms`. For `gaussian`, although INLINE completed behavioral synthesis, performance is not good despite the smallest number of execution cycles among all the results of `gaussian`. This is because inlining all the functions generated a large circuit with the complicated controller, which caused the very long clock period. These results mean that INLINE is not practical for large designs. FBPAR-CLUS with the constraint equal to or larger than 375 also could not complete behavioral synthesis within 24 hours for `fft` and `lms`.

As mentioned in Sect. 3.4, some results in Tables 4, 7, and 10 are slightly different from the estimations in Sect. 3.2. For some partitioning solutions, the actual numbers of states in the main modules are smaller than the estimated ones since inlining functions allows various optimizations in behavioral synthesis beyond the boundary of functions as well as expands the operation-level parallelism, which results in the smaller numbers of states than the estimations. This also causes the variance of execution cycles among the partitioning solutions contrary to our assumption that the number of execution cycles does not differ. However, this variance has the negligibly small influence on performance compared with that of the clock period. Thus, our approach to manage performance by the number of states is still useful. Similary, influences of various optimizations in behavioral synthesis slightly vary the functional unit requirements from the estimation. Among the three sets of experiments, the estimation error of the total area of the required functional units was at worst 1.9%. Such underestimation and overestimation should be improved in our future work.

Next, let us look at Figs. 6, 7, and 8, which depict the performance-area trade-off points for `fft`, `lms`, and `gaussian`, respectively. For FBPAR-CLUS and FBPAR-CLUS-PE, as the constraint becomes loose, the area decreases due to the increased resource sharing, while the execution time increases due to the long delay resulted from the complicated controller. Each of these methods well de-

Table 6 Earlier partitioning method (FBPAR-CLUS) for `lms`.

Constraint on states	No. of modules	No. of states		Exec. cycles	Clock period (ns)	Exec. time (μs)	Area (LUTs)	CPU time (s)
375-	—	Est.	358	—	—	—	—	0.09
		Act.	—					
200-350	2	Est.	37 / 191	2,544	24.14	61.40	45,456	0.08
		Act.	38 / 191					
150-175	2	Est.	121 / 131	2,459	24.31	59.78	55,925	0.08
		Act.	107 / 131					
125	3	Est.	37 / 115 / 78	2,455	23.58	57.89	56,005	0.08
		Act.	38 / 115 / 78					
100	3	Est.	65 / 85 / 78	2,524	22.03	55.61	63,742	0.08
		Act.	66 / 85 / 78					
75	5	Est.	37 / 62 / 25 / 55 / 55	2,544	20.27	51.56	72,971	0.08
		Act.	38 / 62 / 25 / 55 / 55					

Table 7 Our proposed method (FBPAR-CLUS-PE) for `lms`.

Constraint on states	No. of modules	No. of states		Exec. cycles	Clock period (ns)	Exec. time (μs)	Area (LUTs)	CPU time (s)
275-	2	Est.	274 / 25	2,012	34.07	68.55	69,174	0.06
		Act.	259 / 25					
175-250	2	Est.	121 / 168	2,052	24.48	50.24	65,353	0.06
		Act.	121 / 168					
125-150	3	Est.	31 / 115 / 78	2,092	23.11	48.35	57,706	0.06
		Act.	29 / 115 / 78					
100	3	Est.	59 / 85 / 78	2,092	20.86	43.63	63,792	0.06
		Act.	59 / 85 / 78					
75	5	Est.	31 / 62 / 25 / 55 / 55	2,092	20.37	42.62	73,742	0.05
		Act.	29 / 62 / 25 / 55 / 55					

Table 8 Traditional synthesis methods for `gaussian`.

Synthesis method	No. of modules	No. of states	Exec. cycles	Clock period (ns)	Exec. time (μs)	Area (LUTs)
INLINE	1	176	208	27.41	5.70	51,783
FBPAR	6	37 / 32 / 32 / 25 / 55 / 55	331	17.37	5.75	53,091
FBPAR-PE	6	29 / 32 / 32 / 25 / 35 / 55	229	17.26	3.95	53,197

Table 9 Earlier partitioning method (FBPAR-CLUS) for `gaussian`.

Constraint on states	No. of modules	No. of states		Exec. cycles	Clock period (ns)	Exec. time (μs)	Area (LUTs)	CPU time (s)
275- (INLINE)	1	Est.	275	208	27.41	5.70	51,783	0.08
		Act.	176					
125-250	2	Est.	27 / 119	331	19.82	6.56	32,480	0.06
		Act.	27 / 119					
75-105	3	Est.	27 / 62 / 69	331	19.96	6.61	35,590	0.06
		Act.	27 / 62 / 69					
50	4	Est.	27 / 43 / 32 / 48	331	17.01	5.63	40,897	0.05
		Act.	27 / 43 / 32 / 48					

Table 10 Our proposed method (FBPAR-CLUS-PE) for `gaussian`.

Constraint on states	No. of modules	No. of states		Exec. cycles	Clock period (ns)	Exec. time (μs)	Area (LUTs)	CPU time (s)
175-	2	Est.	174 / 36	220	23.65	5.20	50,193	0.08
		Act.	171 / 36					
125-150	2	Est.	108 / 85	220	20.21	4.45	46,373	0.08
		Act.	106 / 85					
100	3	Est.	19 / 85 / 36	229	21.16	4.85	45,423	0.06
		Act.	19 / 85 / 36					
75	3	Est.	19 / 66 / 55	229	18.94	4.34	50,219	0.05
		Act.	19 / 66 / 55					
50	4	Est.	47 / 32 / 36 / 25	224	17.20	3.85	52,168	0.05
		Act.	46 / 32 / 36 / 25					

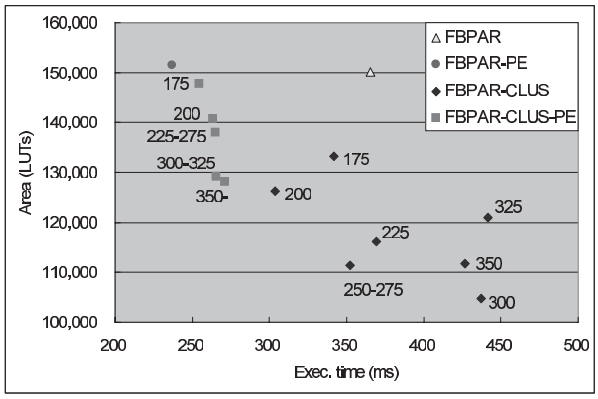

Fig. 6 Performance-area trade-off for `fft`.

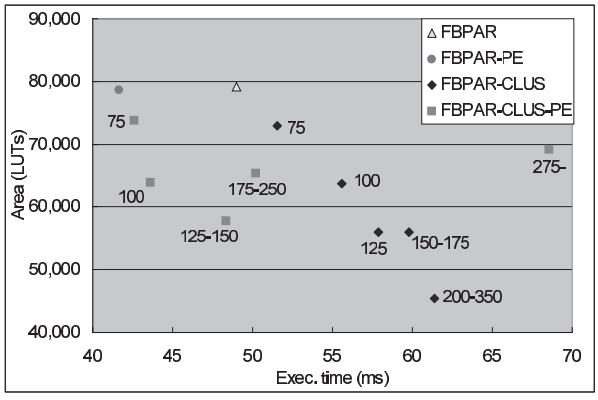

Fig. 7 Performance-area trade-off for `lms`.

scribes the area-performance trade-off points. Furthermore, FBPAR-CLUS-PE leads to a shift of the explorable design space so that superior solutions which could not be explored by FBPAR-CLUS are included. Compared with FBPAR-CLUS, FBPAR-CLUS-PE can find high performance solutions with still almost the same area. In Fig. 6, "300-325" and "350-" by FBPAR-CLUS-PE achieve more than 10% higher performance than "200" by FBPAR-CLUS. Similarly, in Fig. 7 "75" by FBPAR-CLUS-PE achieves 17% higher performance than "75" by FBPAR-CLUS, and in Fig. 8 "50" by FBPAR-CLUS-PE achieves 48% higher performance than "275-" by FBPAR-CLUS. Also, in Fig. 7, "100," "125-150," and "175-250" by FBPAR-CLUS-PE are better in both performance and the area than "75" by FBPAR-CLUS, which has the highest performance of solutions by FBPAR-CLUS. These facts indicate that FBPAR-CLUS-PE can efficiently explore the better performance-area trade-off points and is more effective than FBPAR-CLUS[†].

Note that FBPAR-CLUS-PE and FBPAR-CLUS complement each other. By utilizing FBPAR-CLUS-PE together with FBPAR-CLUS, designers can explore the wider design space. Similarly, FBPAR-CLUS-PE can be utilized with FBPAR-PE for the same purpose. Therefore, it is useful to utilize either of these three methods depending on the designers' need on the quality of circuits. It would be better to use FBPAR-CLUS if designers emphasize area more than performance, while FBPAR-PE if designers emphasize performance more than area. And, FBPAR-CLUS-PE is efficient if designers would like to explore the performance-area trade-off points between them.

[†]The performance of "275-" by FBPAR-CLUS-PE is degraded from that of solutions by FBPAR-CLUS. This is because the clock period in "275-" became long due to the large number of states in a module compared with that in solutions by FBPAR-CLUS.

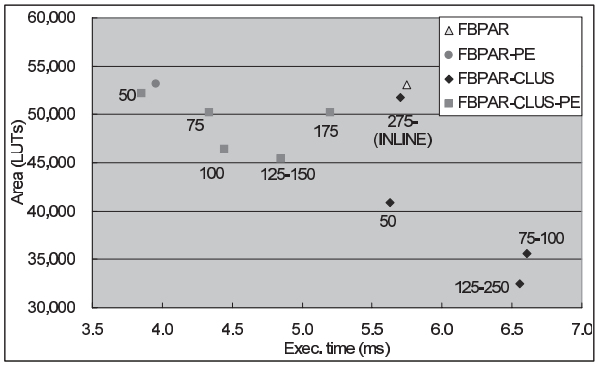

Fig. 8 Performance-area trade-off for `gaussian`.

5. Conclusions

In this paper, we have proposed a behavioral partitioning method based on integer programming. Our method determines functions to be inlined into the main module and ones to be implemented in sub modules in such a way that simultaneously the overall datapath area and the complexity of the controller is minimized while maximizing performance of a synthesized circuit by fully exploiting function-level parallelism of a behavioral description. Experimental results demonstrated that our proposed method led to a shift of the explorable design space so that superior solutions which could not be explored by earlier work are included, showing the effectiveness of our proposed method.

The current partitioning method assumes a single-level hierarchy of function calls. Also, it is assumed that functions executable in parallel have no communication with each other. These assumptions should be relaxed in future.

Acknowledgments

This work is in part supported by KAKENHI 19700040.

References

[1] D.D. Gajski, N.D. Dutt, A.C.-H. Wu, and S.Y.-L. Lin, High-Level Synthesis: Introduction to Chip and System Design, Kluwer Academic Publishers, 1992.

[2] F. Vahid, "Partitioning sequential programs for CAD using a three-step approach," ACM Trans. Des. Autom. Electron. Syst., vol.7, no.3, pp.413–429, 2002.

[3] R. Gupta and G. De Micheli, "Partitioning of functional models of synchronous digital systems," International Conference on Computer-Aided Design, pp.216–219, 1990.

[4] P. Lakshmikanthan, S. Govindarajan, V. Srinivasan, and R. Vemuri, "Behavioral partitioning with synthesis for Multi-FPGA architectures under interconnect, area, and latency constraints," International Parallel & Distributed Processing Symposium, pp.924–931, 2000.
[5] V. Srinivasa, S. Govindarajan, and R. Vemuri, "Fine-grained and coarse-grained behavioral partitioning with effective utilization of memory and design space exploration for multi-FPGA architectures," IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol.9, no.1, pp.140–158, 2001.
[6] W.-J. Fang and A.C.-H. Wu, "Integrating HDL synthesis and partitioning for multi-FPGA designs," IEEE Trans. Design Test, vol.15, no.2, pp.65–72, 1998.
[7] Y. Fei and N.K. Jha, "Functional partitioning for low power distributed systems of systems-on-a-chip," International Conference on VLSI Design, pp.274–281, 2002.
[8] M. Takahashi, N. Ishiura, A. Yamada, and T. Kambe, "Thread composition method for hardware compiler bach maximizing resource sharing among processes," IEICE Trans. Fundamentals, vol.E83-A, no.12, pp.2456–2463, Dec. 2000.
[9] K. Jasrotia and J. Zhu, "Stacked FSMD: A power efficient microarchitecture for high level synthesis," International Symposium on Quality Electronic Design, pp.425–430, 2004.
[10] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii, "Function-level partitioning of sequential programs for efficient behavioral synthesis," IEICE Trans. Fundamentals, vol.E90-A, no.12, pp.2853–2862, Dec. 2007.
[11] J. Uchida, N. Togawa, M. Yanagisawa, and T. Ohtsuki, "A thread partitioning algorithm in low power high-level synthesis," Asia and South Pacific Design Automation Conference, pp.74–79, 2004.
[12] S. Gupta, R.K. Gupta, N.D. Dutt, and A. Nicolau, "Coordinated parallelizing compiler optimizations and high-level synthesis," ACM Trans. Des. Autom. Electron. Syst., vol.9, no.4, pp.441–470, 2004.
[13] M. Nishimura, N. Ishiura, Y. Ishimori, H. Kanbara, and H. Tomiyama, "High-level synthesis of software function calls," IEICE Trans. Fundamentals, vol.E91-A, no.12, pp.3556–3558, Dec. 2008.
[14] Y Explorations, Inc., http://www.yxi.com/
[15] NEC System Technologies, Ltd., http://www.necst.co.jp/english/index.htm
[16] 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.
[17] F. Vahid, "Procedure exlining: A transformation for improved system and behavioral synthesis," International Symposium on System Synthesis, pp.84–89, 1995.
[18] F. Vahid, "Procedure cloning: A transformation for improved system-level functional partitioning," ACM Trans. Des. Autom. Eletron. Syst., vol.4, no.1, pp.70–96, 1999.
[19] F. Vahid, "Techniques for minimizing and balancing I/O during functional partitioning," IEEE Trans. Comput.-Aided Des. Integrated Circuits Syst., vol.18, no.1, pp.69–75, 1999.
[20] SNU Real-time Benchmarks, http://archi.snu.ac.kr/realtime/benchmark/
[21] Y. Hara, H. Tomiyama, S. Honda, and H. Takada, "Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis," IPSJ Journal of Information Processing, vol.17, pp.242–254, Oct. 2009.
[22] SoftFloat, http://www.jhauser.us/arithmetic/SoftFloat.html
[23] Xilinx Inc., http://www.xilinx.com/
[24] Synplicity, http://www.synplicity.com/

Yuko Hara received her B.E. in Information Engineering and her M.E. in Graduate School of Information Science from Nagoya University in 2006 and 2008, respectively. Currently she is a Ph.D. candidate at Graduate School of Information Science, Nagoya University. Her research interests include behavioral synthesis and embedded systems. She is a member of ACM and IPSJ.

Hiroyuki Tomiyama received his Ph.D. degree in computer science from Kyushu University in 1999. From 1999 to 2001, he was a visiting postdoctoral researcher with the Center of Embedded Computer Systems, University of California, Irvine. From 2001 to 2003, he was a researcher at the Institute of Systems & Information Technologies/KYUSHU. In 2003, he joined the Graduate School of Information Science, Nagoya University, as an assistant professor, where he is now an associate professor. His research interests include design automation, architectures and compilers for embedded systems and systems-on-chip. He currently serves as an editorial board member of IPSJ Transactions on SLDM, IEEE Embedded Systems Letters, and International Journal on Embedded Systems. He has also served on the organizing and program committees of several premier conferences including ICCAD, ASP-DAC, DATE, CODES+ISSS, and so on. He is a member of ACM, IEEE, and IPSJ.

Shinya Honda received his Ph.D. degree in the Department of Electronic and Information Engineering, Toyohashi University of Technology in 2005. From 2004 to 2006, he was a researcher at the Nagoya University Extension Course for Embedded Software Specialists. In 2006, he joined the Center for Embedded Computing Systems, Nagoya University, as an assistant professor. His research interests include system-level design automation and real-time operating systems. He received the best paper award from IPSJ in 2003. He is a member of IPSJ.

Hiroaki Takada is a professor at the Department of Information Engineering, the Graduate School of Information Science, Nagoya University. He is also the executive director of the Center for Embedded Computing Systems (NCES). He received his Ph.D. degree in Information Science from the University of Tokyo in 1996. He was a research associate at the University of Tokyo from 1989 to 1997, and was a lecturer and then an associate professor at Toyohashi University of Technology from 1997 to 2003. His research interests include real-time operating systems, real-time scheduling theory, and embedded system design. He is a member of ACM, IEEE, IPSJ, and JSSST.