PAPER

Function-Level Partitioning of Sequential Programs for Efficient Behavioral Synthesis

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

SUMMARY This paper proposes a behavioral level partitioning method for efficient behavioral synthesis from a large sequential program consisting of a set of functions. Our method optimally determines functions to be inlined into the main module and the other functions to be synthesized into sub modules in such a way that the overall datapath is minimized while the complexity of individual modules is lower than a certain level. The partitioning problem is formulated as an integer programming problem. Experimental results show the effectiveness of the proposed method.

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

1. Introduction

Due to the continuously increasing size of LSIs, the traditional Register-Transfer Level (RTL) design with Hardware Design Languages (HDLs) is approaching to its limit on design productivity. Behavioral synthesis (or high-level synthesis), which automatically synthesizes an RTL circuit from a behavioral description, is one of the most promising solutions to improve the design productivity [1]. At present, however, behavioral synthesis tools have not been widely used in industry yet since the quality of automatically generated circuits is still inferior to that of human-designed ones, especially, in the synthesis from large behavioral descriptions.

This paper studies behavioral level partitioning which divides a large behavioral description into a set of smaller ones. Behavioral level partitioning yields a number of advantages such as reduced synthesis runtime, improved performance, satisfaction of packaging or I/O constraints, and so on [2]. In fact, most of the earlier studies on partitioning [1] aim at satisfying physical design constraints such as packaging and I/O pins, but recently, improvement of performance and synthesis runtime has become a more important concern of designers.

In general, a large sequential description written in a high-level programming language such as C[*] consists of a set of functions (or procedures). Without behavioral level partitioning, 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 critical path delay due to the complicated control path as well as multiplexers in the datapath, or behavioral synthesis may not be completed within a practical time as we will see in our experiments. Specifically, inlining is ineffective for large functions which are called a number of times from different points of the program text.

A straightforward approach to behavioral level partitioning is to run behavioral synthesis for each function. This function-based partitioning approach produces N hardware modules (one main module and $N-1$ sub modules) from a program consisting of N functions. Some behavioral synthesis tools which have been developed recently such as SPARK [3] and CCAP [4] employ the function-based approach. However, this approach suffers from a large datapath area because hardware resources (e.g., functional units, registers, memories, etc.) cannot be shared among functions even when only a single function is active at a time.

In our earlier work [5], we have proposed a 2-way partitioning method based on integer programming. The method optimally determines functions to be inlined into a main module and ones to be synthesized into a sub module in such a way that the overall datapath is minimized while keeping the complexity of the control path lower than a certain level. The method enables resource sharing among multiple functions by merging them into one function. However, the method assumes that there exists only a single sub module in addition to a main module.

This paper proposes an improved method for behavioral level partitioning. This work significantly improves our previous method [5] in two ways. First, this work performs N-way partitioning, where N denotes the number of hardware modules[**], while the previous work does 2-way partitioning. In addition, this work optimizes N, simultaneously with N-way partitioning.

In the last decade, Vahid has extensively studied behavioral level partitioning for sequential programs [2], [6]–[8]. The partitioning approach presented in [2] 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 such as inlining, exlining [6], cloning [7], port calling [8], and so on. After that, traditional N-way partitioning is performed. Our method presented in this paper is different from

Manuscript received May 31, 2007.
Manuscript revised August 28, 2007.
Final manuscript received September 6, 2007.

[†]The authors are with the Department of Information Engineering, the Graduate School of Information Science, Nagoya University, Nagoya-shi, 464-8603 Japan.
[††]The author is with the Information Technology Center, Nagoya University, Nagoya-shi, 464-8603 Japan.
a) E-mail: hara@ertl.jp

DOI: 10.1093/ietfec/e90–a.12.2853

[*]This paper assumes the C language as an input language for behavioral synthesis.

[**]N can be smaller than the number of functions in the input program.

the work in [2] in several ways. First, our method simultaneously performs function inlining and N-way partitioning, while the work in [2] achieves them in separate steps. Next, we optimize the number of hardware modules at the same time. Finally, we formally define the partitioning problem using integer linear programming (ILP) in order that the optimal partitioning can be obtained.

The rest of this paper is organized as follows. Section 2 describes two traditional approaches for behavioral level partitioning and their comprehensive comparisons in the preliminary experiments. Section 3 presents an improved method for function-based partitioning. Also, the preliminary experiments to compare the traditional approaches and the improved method are shown. Section 4 proposes a function-level partitioning method based on integer programming. Section 5 shows experimental results to demonstrate the effectiveness of the proposed method. Section 6 concludes this paper with a summary and future work.

2. Traditional Methods

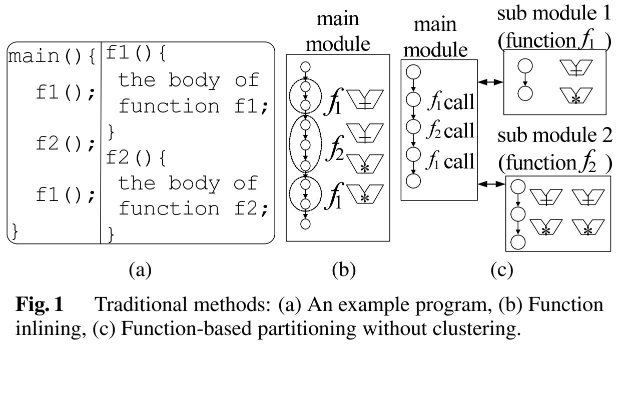
This section presents fundamental techniques for behavioral synthesis and discusses their advantages and disadvantages.

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 used for behavioral synthesis.

Let us consider an example program shown in Fig. 1(a). This program consists of a main function and two functions, `f1` and `f2`, which are called from the main function. Figure 1(b) shows the FSM of a circuit which is synthesized from the program in Fig. 1(a) with inlining. Note that function `f1` is inlined twice since it is called twice from the main function.

Inlining has several advantages. First, inlining enables resource sharing among different functions, resulting in small datapath area. Assume that function `f1` requires one adder and one multiplier, and function `f2` requires two adders and two multipliers. 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 used in total. Second, no overhead of inter-module communication is necessary. Next, inlining extends the scope of optimizations such as common sub-expression elimination,



```
main(){  f1(){
  f1();    the body of
           function f1;
  f2();  }
         f2(){
  f1();    the body of
           function f2;
}        }
```

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

constant propagation, copy propagation, dead-code elimination and so on. Finally, inlining also extends operation-level parallelism.

Inlining, however, increases both the area and delay of control path because the number of states in the main module becomes large. Also, inlining may increase the datapath delay due to multiplexers between functional units and registers. At worst, if the main function after inlining becomes too large, behavioral synthesis may not be completed within a practical time. These disadvantages become critical especially in the case of large functions are called a number of times from different points of the program text.

2.2 Function-Based Partitioning without Clustering

A straightforward approach to behavioral level partitioning is to run behavioral synthesis for each function. This function-based partitioning approach produces N hardware modules (one main module and $N-1$ sub 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 function `f1` even though it is called twice.

The main advantage of function-based partitioning is that it can reduce the complexity of individual modules. Thus, the area and delay of the control path can be reduced. On the other hand, the function-based partitioning increases the overall datapath area because hardware resources cannot be shared among functions even when the same resources are required by multiple functions. For example, in Fig. 1(c), three adders and three multipliers are allocated in total even though the two functions are sequentially executed. Another disadvantage of this approach may degrade the performance by inter-module communication.

2.3 Preliminary Comparisons between Function Inlining and Function-Based Partitioning without Clustering

We preliminarily made comparisons to evaluate the advantages and disadvantages of inlining and function-based partitioning without clustering. We used five benchmark programs consisting of two functions: a main function and a double-precision floating-point addition function which is called from the main function. The addition function is implemented by using integers [9]. Fundamental structures of these programs are almost the same except the number of times the addition function is called. We compared circuit area (equivalent gate count), clock period, the number of states, the number of execution cycles, execution time and behavioral synthesis time for the five programs. In the experiments, we used commercial tools YXI eXCite 3.0 [10] for behavioral synthesis and Xilinx ISE [11] for logic synthesis. In these comparisons, we specified the constraint on clock period as 50 MHz (20 ns) and Xilinx Virtex 2 as a target device.

The results of the preliminary experiments are shown in Table 1. The first column represents the number of times the addition function is called from the main function. In

Table 1 Comparisons between function inlining and function-based partitioning without clustering.

No. of times the addition function is called	applied technique	gate count	clock period (ns)	No. of states		exec. cycles	exec. time (ns)	beh. syn. time (s)
				main module	sub module			
1	partitioned	68,636	36.310	8	61	26	944	248
	inlined	62,775	36.310	61	—	21	763	237
2	partitioned	69,898	36.310	12	61	40	1,452	250
	inlined	80,146	36.497	118	—	30	1,095	774
4	partitioned	69,236	35.937	20	61	68	2,444	252
	inlined	101,312	37.082	232	—	48	1,780	4,661
8	partitioned	70,240	36.310	36	61	124	4,502	258
	inlined	—	—	—	—	—	—	—
16	partitioned	70,764	36.310	68	61	236	8,569	268
	inlined	—	—	—	—	—	—	—

the second column, the applied technique for the addition function is described. When the addition function is called once, inlining generates better circuit for both datapath area and control path area (the number of states). However, as increasing the number of times the addition function is called, inlining leads to the increasing number of states in the main module, which results in the larger control path and longer synthesis time. Besides, when the addition function is called eight times or more, behavioral synthesis does not finish since the main function after inlining became too large.

Comprehensive comparisons between function inlining and function-based partitioning without clustering show that inlining results in a long critical path delay due to the complicated control path as well as multiplexers in the datapath. On the other hand, function-based partitioning leads to an increase in the number of execution cycles due to the inter-module communication, and the execution time also becomes longer.

Based on the above results, it is not appropriate to inline functions which are called a number of times. On the other hand, all the functions should not individually be partitioned because sharing no hardware resources makes the circuit area larger. We will show the more shortcomings of function-based partitioning without clustering in Sect. 3.

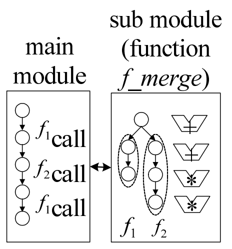
3. Sharing of Hardware Resources by Function-Based Partitioning with Clustering

In this section, we explain a method called "function-based partitioning with clustering," which is an improvement of both function inlining and function-based partitioning without clustering.

3.1 Function-Based Partitioning with Clustering

As shown in Sect. 2.2, function-based partitioning is not efficient in terms of the datapath area. This drawback can be suppressed by clustering functions.

Figure 2(a) shows the FSM of the circuit synthesized from the same program in Fig. 1(a) with clustering. In Fig. 2(a), two functions `f1` and `f2` are clustered into the same sub module in order that they can share the hardware resources. The number of states in the sub module is almost the same as the sum of those of individual modules. Note that the number of states for `f1` is not doubled though `f1` is called twice. Thus, clustering can keep the datapath area small while suppressing the complexity of the control path.



(a)

```
main(){

 f_merge(1);


 f_merge(2);


 f_merge(1);

}
```

```
f_merge(int id){
 switch (id){
 case 1:
  /*to be inlined*/
  f1();
  break;
 case 2:
  /*to be inlined*/
  f2();
  break;
 }
}
```

(b)

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

Function clustering can be implemented by transforming the original program in Fig. 1(a) into one shown in Fig. 2(b). A function `f_merge` is newly defined, which calls either `f1` or `f2` depending on a parameter `id`. Next, when synthesizing from `f_merge`, `f1` and `f2` are inlined into `f_merge`, while `f_merge` itself is not inlined into the main function.

As shown above, clustering enables sharing of hardware resources among functions and also results in a smaller number of states than inlining does. However, the amount of communication between modules is increased in order to send `id` of the function to be called. This may degrade the performance. In addition, the sub module requires additional comparators to select the function to be executed. Also, it should be mentioned that clustering too many functions in the same module results in a complex control path.

3.2 Preliminary Comparisons between Three Techniques

We made preliminary comparisons to show that function-based partitioning with clustering can generate more efficient circuits than inlining and function-based partitioning without clustering. We used a benchmark program consisting of three functions: a main function, addition and subtraction functions for double-precision floating-point numbers [9]. The addition and subtraction functions are called from the main function twice and once, respectively. The hardware resources required by these two functions are almost same.

We compared the following five methods for the benchmark program:

partitioned without clustering: the addition and subtrac-

Table 2 Comparisons of traditional methods and function-based partitioning with clustering.

applied technique		gate count	clock period (ns)	No. of states		exec. cycles	exec. time (ns)	beh. syn. time (s)
`add`	`sub`			main module	sub module			
partitioned w/o clustering		136,427	35.937	18	61/61(`add`/`sub`)	61	2,192	494
inlined	inlined	93,403	36.648	177	—	46	1,686	1,734
partitioned	inlined	131,880	35.937	71	61	56	2,012	481
inlined	partitioned	145,669	36.419	124	61	51	1,857	1,071
partitioned w/ clustering		88,599	36.497	21	121	67	2,445	793

tion functions are partitioned without clustering.

inlined/inlined: the addition and subtraction functions are inlined into the main module.

partitioned/inlined: the addition function is partitioned into a sub module and the subtraction function is inlined into the main module.

inlined/partitioned: the addition function is inlined into the main module and the subtraction function is partitioned into a sub module.

partitioned with clustering: the addition and subtraction functions are partitioned with clustering.

We compared circuit area (equivalent gate count), clock period, the number of states, the number of execution cycles, execution time and synthesis time between five methods shown above. We used the same synthesis tools and constraints as described in Sect. 2.3. The results of the preliminary experiments are shown in Table 2.

The circuit area generated by partitioned without clustering is large since each module possesses the hardware resources and cannot share them between modules. While the area of the circuit generated by inlined/inlined is smaller than that by partitioned without clustering since both functions are inlined into the main module and can share the resources. In addition, there is no inter-module communication, resulting in both the smallest number of execution cycles and shortest execution time of all. However, the complexity of the control path is increased due to the large number of states in the main module, which leads to the longest time in behavioral synthesis.

In partitioned/inlined, its total time in behavioral synthesis can be kept short, but these two functions cannot share the resources, and therefore, the total area increases. In inlined/partitioned, the hardware resources cannot be shared between modules as in partitioned/inlined. In inlined/partitioned, a larger area is generated than in partitioned/inlined since the complexity of the control path is increased by inlining the addition function, which is called twice from the main function, in addition to sharing no resources between modules.

In partitioned with clustering, by partitioning after clustering the two functions in a single function, it is possible to share the hardware resources required by both these two functions as well as in inlining, and its datapath area is moderated. Also, it can suppress the number of states, which leads to the small control path. As a result, its total area is the smallest among the five methods.

As shown in these preliminary comparisons, it is possible to moderate the circuit area by applying partitioning with clustering for functions which require the same resources. In addition, this method can decrease the number of states in both modules compared with inlining, especially when the same function is called several times. Thus, the advantages of both inlining and partitioning without clustering are available in partitioning with clustering. If too many functions are clustered in the same module, however, the generated module has the large number of states, which leads to the long delay and a large area of the control path and long time in behavioral synthesis, similar to inlining. Therefore, it is important to appropriately determine functions to be inlined into the main module and ones to be partitioned into sub modules, when synthesizing from a large program with a number of functions because it is not practical to cluster all the functions in a single module.

4. Complexity-Constrained Partitioning

This section proposes a new behavioral level partitioning method based on integer programming.

4.1 Problem Description

As discussed in the previous sections, function inlining is effective in order to reduce execution cycles and datapath area. However, the size of functions after inlining should not exceed the limit manageable by synthesis tools.

Given a sequential program consisting of a set of functions, we face the following questions:

- Which functions should be inlined?
- Which functions should be clustered into the same module?
- How many modules are necessary?

This section proposes a behavioral partitioning method that simultaneously solves these three questions in a single combinational optimization framework based on integer programming. Our goal is to minimize the overall datapath area (i.e., the total cost of hardware resources), 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 [12].

For simplicity, we assume that no function except the main function calls other functions. Relaxation of this assumption is one of our future works. At present, 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 [2]

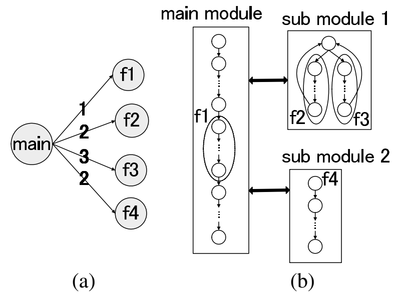


Fig. 3 Partitioning example: (a) Call graph of a program, (b) Partitioning result.

can be used.

Let us consider an example shown in Fig. 3(a). Fig. 3(a) describes a call graph of a program consisting of a main function and four functions which are called from the main function. The number associated with each edge denotes the number of function calls written in the program text[†]. Our method generates a partitioning as shown in Fig. 3(b), where function `f1` is inlined into the main module, functions `f2` and `f3` are clustered into the same sub module, and function `f4` is implemented in a different sub module.

4.2 Problem Formulation

In this section, we formulate the partitioning problem as an integer programming problem. First, we define the following 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 ($i = 0, 1, \ldots, N_F - 1$)
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
N_M**:** the number of hardware modules
m_k**:** hardware module ($k = 1, \ldots, N_M - 1$)
m_0 represents a main module synthesized from the main function.
S_k**:** the number of states in module m_k
d_i**:** the number of states for inter-module communication for function f_i
S_{const}**:** the designer-specified constraint on the number of states for each module
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 d_i is one in most cases, but may be more than one depending on the numbers and sizes of data to be transferred. Therefore, for generality, we leave d_i as a parameter rather than one.

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

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

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

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

$$S_0 = s_0 + \sum_i x_{i,0} \cdot (s_i - d_i) \cdot c_i \tag{1}$$

$$S_k = \sum_i x_{i,k} \cdot s_i + \min(1, \sum_i x_{i,k}), \quad (k = 1, \ldots, N_M - 1) \tag{2}$$

Formula (1) represents the estimated number of states in the main module. The number of states for inlined functions is added to that for the main module. Since no communication overhead is necessary, d_i is subtracted from s_i. With function inlining, the actual number of states can be less than the one obtained by formula (1) because of global optimization beyond function boundaries. This effect is not considered in our current formulation and should be incorporated in the future.

Formula (2) represents the estimated number of states in sub module m_k. The last term $min(1, \sum_i x_{i,k})$ denotes an additional state for selecting the function to be executed when multiple functions are clustered into m_k.

The number of states in each module cannot exceed the limit specified by a designer. Therefore, the formula below must hold.

$$S_k \le S_{const}, (k = 0, 1, \ldots, N_M - 1) \tag{3}$$

Next, the datapath area of the main module and sub modules can be estimated by formulas (4) and (5), respectively.

$$A_0 = \sum_j \{\max_i(x_{i,0} \cdot n_{i,j}) \cdot a_j\} \tag{4}$$

$$A_k = \sum_j \{\max_i(x_{i,k} \cdot n_{i,j}) \cdot a_j\} + N_k^{cmp} \cdot a^{cmp}, (k = 1, \ldots, N_M - 1) \tag{5}$$

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:

$$A_{total} = \sum_k A_k \tag{6}$$

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 (6) with

[†]Note that the number does not mean the dynamic number of function calls.

satisfying the constraints in formula (3). 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 method, simulated annealing, the genetic algorithm, and so on, can be applied. As we will see 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.

$$N_M = N_F \tag{7}$$

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.

4.3 Overall Synthesis Flow

As defined above, our partitioning method requires that, for each function, the numbers and types of hardware resources (i.e., $n_{i,j}$) and the number of states (i.e., s_i) be known. Therefore, we need to run behavioral synthesis for each function in order to obtain these pieces of information. Thus, the overall flow of behavioral synthesis is as follows.

1. Flatten the hierarchy of function calls into a single level by inlining.
2. Execute behavioral synthesis for each function to obtain $n_{i,j}$ and s_i.
3. Execute behavioral level partitioning by the proposed method.
4. Execute behavioral synthesis for each cluster.

One problem in Step 3 is how a designer should decide the constraint on the number of states. The optimal number of states is affected by several factors such as the ability of the behavioral and logic synthesis tools, the clock frequency constraint, and so on, so finding the optimal one is not an easy problem. Therefore, the synthesis steps described above need to be repeated with varying the constraint on the number of states. The key point is that the method proposed in this paper can be used as a tool which significantly reduces the design space to be explored. For each constraint, our method finds an optimal partitioning without actually synthesizing the circuits, so the designer does not have to run behavioral/logic synthesis for all the possible partitionings.

4.4 Discussion on the Constraint

In our function-level partitioning approach, we try to keep the control path complexity below a certain level. In general, however, it is a very difficult problem to estimate the complexity of the control path before behavioral and logic synthesis. In our formulation, therefore, we use the number of states as an approximate metric for control path complexity as defined in formulas (1)–(3). The reason is as follows. A control path can be divided into two parts. One is a circuit to generate control signals to the datapath (e.g., control signals of multiplexers, read/write signals of registers, and so on), and the other is one to compute the next state. Out of the two, the circuit to generate control signals often exists on the critical path of the overall circuit because computation in the datapath starts upon the arrival of the control signals. During behavioral synthesis, therefore, it is important not to increase the delay of the path for control signal generation. Note that the path delay for control signal generation is heavily dependent on the number of state bits, especially, in case of Moore's FSM-based control paths. In our ILP formulation, therefore, we constrain the maximum number of states for each module in order not to increase the critical path delay.

Although the number of states is not an accurate approximation for control path delay, our experimental results show the effectiveness of our ILP formulation as we will see in the next section. However, one may want to model the control path delay more accurately. Recent work by Gupta reveals in [12] that the number of operations in the behavioral description is another important factor for deciding the control path complexity. Our ILP formulation can be easily extended so that the number of operations should not exceed a certain level for each module. Such extention is realized by using the following constraint formulas (8)–(10) in addition to the original formulas (1)–(3)[†].

o_i**:** the number of operations in a module synthesized from function f_i individually
O_k**:** the number of operations in module m_k
O_{const}**:** the designer-specified constraint on the number of operations for each module

$$O_0 = o_0 + \sum_i x_{i,0} \cdot o_i \cdot c_i \tag{8}$$

$$O_k = \sum_i x_{i,k} \cdot o_i, (k = 1, \ldots, N_M - 1) \tag{9}$$

$$O_k \leq O_{const}, (k = 0, 1, \ldots, N_M - 1) \tag{10}$$

It should also be noted that, since the behavioral synthesis runtime significantly depends on the number of operations in the behavioral description, the use of our new formulas (8)–(10) is also effective in order to complete behavioral synthesis in a practical time. In fact, the number of operations and that of states are not independent, i.e., more operations tend to result in more states. In other words, using the number of states as a constraint implicitly constrains that of operations. Actually, our experimental results presented in the next section show that the tighter constraint on the number of states leads to shorter runtime of behavioral synthesis.

In our ILP formulation, the performance (i.e., the number of execution cycles) is also taken into account not explicitly but implicitly. As shown in experiments in the next

[†]In [12], the authors have developed a more complex formula for estimation of the control delay.

section, looser constraints on the number of states tend to result in an increase in execution cycles with the area reduced. This indicates that designers can efficiently explore performance-area trade-off points by varying the constraint on the number of states.

5. Experiments

We conducted two sets of experiments to demonstrate the effectiveness of our partitioning approach. We used two benchmark programs: `sqrt`, which computes the square-root of a given double-precision floating-point number, and `ludcmp`, which performs LU decomposition for a given matrix of double-precision floating-point numbers. Note that the two benchmark programs are rather large, each of which consists of more than 1,000 lines of C code. Each program is composed of a main function and several double-precision floating-point arithmetic functions such as `double_add`, `double_sub`, `double_mul`, `int_to_double` and so on [9]. Each function contains a number of integer and logic operations as well as complex control structures. For example, `double_add` includes 337 operations, 87 *if* statements, 26 *goto* statements and so on. Table 3 shows the number of function calls in each benchmark program, where `double_` is omitted from some function names due to the limited space. For exmaple, `double_add` is called once in the both programs, while `double_mul` is called three times and five times in `sqrt` and `ludcmp`, respectively.

First, we ran behavioral synthesis for each function in order to obtain the types and numbers of required hardware resources and the number of states. We used a commercial behavioral synthesis tool eXCite 3.0 from YXI [10]. Scheduling was performed without resource constraint. The number of states for each function is shown in Table 3.

Table 3 Characteristics of functions in benchmark programs.

		`main`	`add`	`sub`	`mul`	`div`	`lt`	`le`	`eq`	`int_to_double`
No. of function calls	`sqrt`	—	1	3	3	2	1	1	1	0
	`ludcmp`	—	1	6	5	3	0	3	0	2
No. of states	`sqrt`	61	61	61	33	55	9	9	7	7
	`ludcmp`	153								

Table 4 Experimental results for `sqrt`.

constraint on states	partitioning	No. of states	gate count	clock period (ns)	exec. cycles	exec. time (μs)	area-delay ($\times 10^7$)	beh. syn. time (s)
61 (w/o clustering)	`main`	61	601,256	46.6	1,915	89.3	5.37	834
	`add`	61						
	`sub`	61						
	`mul`	33						
	`div`	55						
	`lt`	9						
	`le`	7						
	`eq`	7						
70 - 80	`main`	67	595,928	46.2	1,977	91.4	5.45	872
	`add`	61						
	`sub`	61						
	`mul, lt, le, eq`	54						
	`div`	55						
90	`main`	70	403,171	51.5	1,996	102.8	4.14	1,091
	`add, lt, le, eq`	82						
	`sub`	61						
	`mul, div`	86						
100 - 120	`main`	70	405,272	50.9	1,996	101.7	4.12	990
	`add, lt, eq`	75						
	`sub`	61						
	`mul, div, le`	93						
130 - 160	`main`	73	351,582	51.5	2,032	104.5	3.67	1,397
	`add, sub`	122						
	`mul, div, lt, le, eq`	105						
170 - 210	`main`	73	383,638	50.3	2,032	102.1	3.92	1,459
	`add, mul, div, le`	152						
	`sub, lt, eq`	75						
220	`main, lt, le, eq`	73	331,066	52.0	1,953	101.6	3.36	2,339
	`add, sub, mul, div`	204						
230	`main, le`	68	330,210	50.3	1,972	99.1	3.27	2,330
	`add, sub, mul, div, lt, eq`	216						
240 - 510	`main`	73	329,466	51.4	2,032	104.4	3.44	2,683
	`add, sub, mul, div, lt, le, eq`	223						
520 - (inlining)	`main, add, sub, mul div, lt, le, eq`	—	—	—	—	—	—	—

Table 5 Experimental results for `ludcmp`.

constraint on states	partitioning	No. of states	gate count	clock period (ns)	exec. cycles	exec. time (μs)	area-delay ($\times 10^8$)	beh. syn. time (s)
61 (w/o clustering)	`main`	153	1,142,470	46.6	5,080	236.8	2.71	916
	`add`	61						
	`sub`	61						
	`mul`	33						
	`div`	55						
	`le`	7						
	`int_to_double`	7						
160	`main`	168	898,931	56.0	5,638	315.6	2.84	1,436
	`add, sub`	122						
	`mul, div, le, int_to_double`	104						
170 - 210	`main`	163	919,131	55.2	5,414	298.7	2.75	1,697
	`add, mul, div, le int_to_double`	165						
	`sub`	61						
220	`main, le`	169	874,399	58.6	5,566	326.0	2.85	2,594
	`add, sub, mul, div int_to_double`	217						
230 - 900	`main`	168	873,584	60.2	5,638	339.2	2.96	2,599
	`add, sub,mul, div, le int_to_double`	226						
910 - (inlining)	`main, add, sub, mul div, le, int_to_double`	—	—	—	—	—	—	—

Due to the limited space, the types and numbers of hardware resources required for each function were omitted in this paper. It should be mentioned that some functions are very resource-hungry. For example, `double_mul` requires four 64-bit adders, one 64-bit subtractor, one 32-bit subtractor, four 64-bit multipliers, one 64-bit divider, two 64-bit shifters, seven 64-bit comparators, and three 32-bit comparators.

Next, we performed behavioral level partitioning proposed in this paper. At that time, we varied the constraint on the number of states. 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.

Then, for each partitioning result, we performed behavioral synthesis, logic synthesis, and place-and-route to evaluate the area and clock period of the design. Xilinx Virtex 2 was used as a target device, and Xilinx ISE [11] was used for logic synthesis and place-and-route. All of these synthesis processes were optimized for performance maximization. Register-transfer level simulation was also performed to measure the execution cycles.

Tables 4 and 5 summarize the synthesis and simulation results for `sqrt` and `ludcmp`, respectively. Behavioral synthesis runtime is also shown in the tables. The first column denotes the constraint on the number of states in individual modules. The fourth column "gate count" means the equivalent gate count including not only LUTs but also built-in multipliers and block RAMs occupied by the design. The eighth column represents area-delay product, which is defined as the product of area (gate count) and execution time. Since in general area and execution time are in a trade-off relation, area-delay product is used to comprehensively evaluate the designs. In the first column, "w/o clustering" denotes the function-based partitioning without clustering described in Sect. 2.2. In both of the benchmark programs, this function-based approach gives the highest performance but also the largest area.

In Table 4, when the state constraint is 70 to 80, five modules were generated. In the fourth module, four functions are clustered. The performance is slightly worse than "w/o clustering" due to the communication overhead required to specify the function ID for the clustered module. When the constraint is 220, the number of execution cycles is relatively small. This is because several functions are inlined into the main function (see Sect. 2.1 for the advantages of function inlining). As the constraint becomes loose, the area tends to decrease due to the increased resource sharing. When the constraint is 170 to 210, the actual number of states in the largest module is 152. This satisfies the more severe constraint (i.e., 160 states). If multiple functions are clustered, the functions can share not only hardware resources but also states. However, this possibility is not captured at present (it should be captured in the future) in our formulation in Sect. 4.2. When the constraint is 520 or larger, all the functions are inlined into the main module. In this case, however, behavioral synthesis could not be completed within 24 hours.

The results for `ludcmp` are shown in Table 5. Similar to the results for Table 4, "w/o clustering" yields the highest performance and the largest area. Also, behavioral synthesis could not be completed when the constraint was the least severe. The area of `ludcmp` is much larger than that of `sqrt` because of the memory for the matrix to be LU-decomposed. When the constraint is 160, the actual number of states in the main module exceeds the constraint. This is due to the underestimation of the communication overhead in the main module, which should be improved in the future.

When the constraint is 100 to 120 in Table 4, in spite of sharing more resources, the area is slightly larger than that when the constraint is 90. Other partitionings with a similar discrepancy are also found in Tables 4 and 5. This is mainly due to the following reason. As described in Sect. 4.2, the proposed method determines the functions to be inlined into the main module and partitioned into sub modules based on estimated datapath area in formulas (4) and (5). Indeed, the estimated area when the constraint is 100 to 120 is smaller than that when the constraint is 90. On the other hand, the area in Table 4 is the actual one after logic synthesis and place-and-route. Logic-level optimization often affects the area significantly, but that is not taken into account in our formulas (4) and (5). This problem needs to be handled more carefully in future.

The experimental results in Tables 4 and 5 demonstrate the effectiveness of the partitioning method proposed in this paper. In the behavioral synthesis from a large program, inlining of all the functions is impractical because the main module becomes too complex to be synthesized. Therefore, the complexity of individual modules should be constrained within a manageable level. Function-based partitioning without clustering leads to a high-performance design, but the area becomes large. The function-level partitioning approach presented in this paper enables efficient exploration of area-performance trade-off points as shown in Tables 4 and 5.

6. 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 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 sequential program.

The current partitioning method assumes a single-level hierarchy of function calls, and does not consider the potential parallelism between modules. These assumptions should be relaxed in future. Also, overestimation and underestimation of the number of states in our formulation need to be improved.

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 TODAES, vol.7, no.3, pp.413–429, July 2002.

[3] S. Gupta, R.K. Gupta, N.D. Dutt, and A. Nicolau, "Coordinated parallelizing compiler optimizations and high-level synthesis," ACM TODAES, vol.9, no.4, pp.441–470, Oct. 2004.

[4] M. Nishimura, K. Nishiguchi, N. Ishiura, H. Kanbara, H. Tomiyama, Y. Takatsukasa, and M. Kotani, "High-level synthesis of variable accesses and function calls in software compatible hardware synthesizer CCAP," Proc. Synthesis And System Integration of Mixed Information technologies (SASIMI), pp.29–34, 2006.

[5] Y. Hara, H. Tomiyama, S. Honda, and H. Takada, "Function call optimization for efficient behavioral synthesis," IEICE Trans. Fundamentals, vol.E90-A, no.9, pp.2032–2036, Sept. 2007.

[6] F. Vahid, "Procedure exlining: A transformation for improved system and behavioral synthesis," Proc. International Symposium on System Synthesis (ISSS), pp.84–89, 1995.

[7] F. Vahid, "Procedure cloning: A transformation for improved system-level functional partitioning," ACM TODAES, vol.4, no.1, pp.70–96, Jan. 1999.

[8] F. Vahid, "Techniques for minimizing and balancing I/O during functional partitioning," IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol.18, no.1, pp.69–75, Jan. 1999.

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

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

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

[12] G.R. Gupta, M. Gupta, and P.R. Panda, "Rapid estimation of control delay from high-level specifications," Proc. Design Automation Conference (DAC), pp.455–458, 2006.

Yuko Hara received her B.E. in Information Engineering from Nagoya University in 2006. Currently she is an M.S. student at the Graduate School of Information Science, Nagoya University. Her research interests include behavioral synthesis and embedded systems. She is a member of ACM.

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 system-level design automation, architectures and compilers for embedded systems and systems-on-chip. He is currently serves as an associate editor of ACM TODAES and an editorial board member of International Journal on Embedded Systems. He has also served on the organizing and program committees of several premier conferences including ICCAD, ASP-DAC, 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 received his Ph.D. degree in Information Science from University of Tokyo in 1996. He was a Research Associate at University of Tokyo from 1989 to 1997, and was an Assistant Professor 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.

Katsuya Ishii received his BSc. and DSc. in Science, University of Tokyo in 1975 and 1980, respectively. He was an Assistant Professor in Faculty of Science, University of Tokyo from 1980 to 1986, and in Faculty of Engineering, University of Tokyo in 1987. From 1988 to 1994, he was a Head of Research Division, ICFD, Co. In 1995, he joined Faculty of Engineering, Nagoya University as an Associate Professor, where he is now a Professor in Information Technology Center, Nagoya University.