IEICE TRANS. FUNDAMENTALS, VOL.E90–A, NO.9 SEPTEMBER 2007

LETTER

Function Call Optimization for Efficient Behavioral Synthesis

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

SUMMARY Behavioral synthesis, which automatically synthesizes an RTL circuit from a sequential program, is one of promising technologies to improve the design productivity. This paper proposes a function call optimization method in behavioral synthesis from large sequential programs with a number of functions. We formulate the optimization problem using integer linear programming. Our experimental results show the reduction in the circuit area by up to 44.6%, compared with a traditional method.

key words: *behavioral synthesis, function calls, integer programming problem*

1. Introduction

The size of LSIs is increasing year by year and the traditional Register-Transfer Level (RTL) design with Hardware Design Languages (HDLs) is approaching to its limit. Behavioral synthesis, which automatically synthesizes an RTL circuit from a sequential program, is one of promising solutions to improve the design productivity [1], and is now being widely used in industry, especially in Japanese industry [2].

This paper addresses behavioral synthesis from large sequential programs written in C. In general, large programs consist of a number of functions. There are two traditional methods to handle function calls in behavioral synthesis: *inlining* and *exlining*. Inlining creates a single large function by inline expansion of all the functions and synthesize, while exlining runs behavioral synthesis for each function. For large programs, inlining suffers from resulting in a huge area and long delay of controller circuits. Exlining, on the other side, leads to a large datapath since the hardware resources cannot be shared between functions.

This paper proposes a method to optimally determine functions to be inlined and ones to be exlined. We formulate the function call optimization problem using integer linear programming.

2. Traditional Methods

First, this section describes two traditional methods, i.e., exlining and inlining, for handling function calls in behavioral synthesis. After that, an improvement of function exlining, called function merge, is presented.

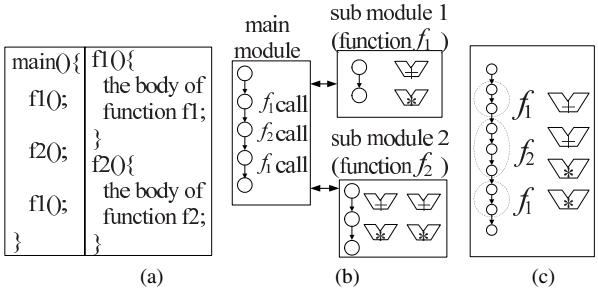

Fig. 1 (a) An example program, (b) Function exlining, (c) Function inlining.

2.1 Function Exlining

Function exlining is to run behavioral synthesis for each function. This approach produces *N* hardware modules from a program consisting of *N* functions.

The program in Fig. 1(a) consists of main function and two functions, `f1` and `f2`. The circuit structure from this program by function exlining is described in Fig. 1(b), where one hardware module is synthesized from one function. Thus, only a single module is synthesized from function `f1` in spite of its being called twice.

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

2.2 Function Inlining

Function inlining replaces function calls with the bodies of the called functions. Fig. 1(c) shows the circuit structure by function inlining for the program in Fig. 1(a). Note that function `f1` is inlined twice since it is called twice by main function.

Inlining has an advantage of moderating the total datapath by sharing the resources between different functions.

Manuscript received February 27, 2007.
Manuscript revised April 20, 2007.
Final manuscript received May 24, 2007.

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

DOI: 10.1093/ietfec/e90–a.9.2032

Copyright © 2007 The Institute of Electronics, Information and Communication Engineers

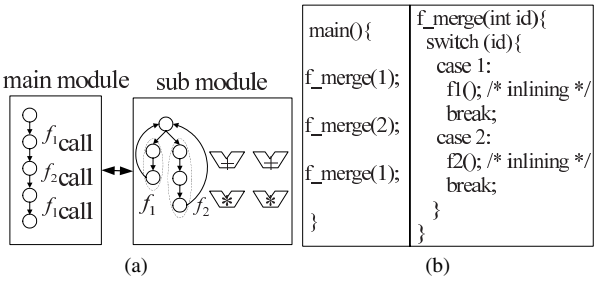

Fig. 2 (a) Function merge, (b) A refined program for function merge.

Moreover, there is no performance degradation caused by inter-module communication. Also, inlining extends operation-level parallelism and the scope of optimizations such as common sub-expression elimination, constant propagation, copy propagation, dead-code elimination and so on. However, the large number of states in main module may produce an inefficient circuit with a long critical path delay due to the complicated control path, or behavioral synthesis may not be completed within a practical time. These disadvantages become significant, especially for programs with large functions called a number of times from different points of the program text.

2.3 Function Merge

Function merge is a method by which functions are exlined into one module, instead of exlining them individually. Figure 2(a) shows the circuit structure by merging functions `f1` and `f2` in Fig. 1(a). It suppresses the number of states in main module. Also, sub module moderates the datapath area by sharing the resources between functions. Moreover, merged functions are implemented only once in sub module, leading to the small control path area in sub module. Thus, merging multiple functions can moderate the overall circuit area.

Function merge can be achieved by transforming a program as shown in Fig. 2(b). A function `f_merge` is newly defined, which calls either function `f1` or `f2` based on a parameter `id`. Functions `f1` and `f2` are inlined into fuction `f_merge`, while fuction `f_merge` itself is exlined.

If too many functions are merged when synthesizing from a program with a number of functions, however, sub module may have the large number of states, similar to inlining. Moreover, the inter-module communication for `id` and the number of comparator in sub module may be increased in order to select the function to be executed. Thus, it is important to appropriately determine functions to be implemented in main module and sub module.

3. Function Call Optimization

In this section, we propose an optimization method to determine functions to be inlined in main module and ones to be exlined/merged into sub module.

3.1 Problem Formulation

We formulate the function call optimization problem as an integer programming one. For simplicity, we assume that no function except the main function calls other functions. At present, if a function `f` calls another function `f'`, either `f'` or `f` needs to be inlined into its caller. For this purpose, granularity selection techniques presented in [3] can be used. This paper also assumes that there exists only a single sub module in addition to main module. Extension towards multiple modules is one of our future works.

Our approach optimally determines the functions to be inlined and ones to be exlined/merged into sub module in such a way that the overall datapath is minimized. Each module should fulfill the constraint on the number of states given by designers. This prevents from too much complexity of controller circuit in each module. If the complexity is not alleviated, the growth in control path area can cause the long critical path, which degrades the clock frequency.

First, we define the following notations:

N_R: the number of hardware resource types
r_j: hardware resource ($j = 0, 1, \cdots, N_R - 1$)
a_j: the area of resource r_j
N_F: the number of functions in a program
F_i: function called by main function ($i = 0, 1, \cdots, N_F - 1$)
c_i: the number of times function F_i is called
d_i: the number of states required for inter-module communication for function F_i
$n_{i,j}$: the number of resources r_j required by function F_i
s_i: the number of states in a module synthesized from function F_i individually
s_{main}: the number of states in main module when all the functions are individually exlined
N_{cmp}: the number of comparator required in sub module
a_{cmp}: the area of a comparator
S_{const}: the designer-specified constraint on the number of states
S_M: the number of states in main module
S_S: the number of states in sub module
A_M: the datapath area of main module
A_S: the datapath area of sub module
A_{total}: the total datapath area

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 as follows:

$$x_i = \begin{cases} 1 & \text{if } F_i \text{ is exlined/merged into sub module} \\ 0 & \text{if } F_i \text{ is inlined in main module} \end{cases}$$

The number of states in main module S_M and sub module S_S are estimated as follows:

$$S_M = s_{main} + \sum_i \{(1 - x_i) \cdot s_i \cdot c_i + d_i \cdot x_i \cdot c_i\} \qquad (1)$$

$$S_S = \sum_i x_i \cdot s_i + \min\left(1, \sum_i x_i\right) \quad (2)$$

The last term in Eq. (1) denotes the inter-module communication overhead. The last term in Eq. (2) represents the number of states to determine the function to be executed in sub module, which is assumed one in this paper.

With function inlining, the actual number of states can be less than the one obtained by Eq. (1) because of global optimization beyond function boundaries. This effect is not considered in our current formulation and should be incorporated in future.

The number of states in each module cannot exceed the limit specified by designers. The formulas below must hold.

$$S_M \leq S_{const}, \quad S_S \leq S_{const} \quad (3)$$

Although the same constraint value is used for main module and sub module in Eq. (3), it is possible to specify the different constraint for each module by slightly modifying Eq. (3).

Next, the number of resources r_j in main module is $\max_i((1 - x_i) \cdot n_{i,j})$, so the datapath area in main module, A_M, is estimated by Eq. (4).

$$A_M = \sum_j \{\max_i((1 - x_i) \cdot n_{i,j}) \times a_j\} \quad (4)$$

Sub module needs not only the resources used by merged functions but also comparators to determine a function to be executed. The number of comparators N_{cmp} is equal to that of merged functions, that is, $N_{cmp} = \sum_i x_i$. The area of sub module, A_S, is estimated in Eq. (5).

$$A_S = \sum_j \{\max_i(x_i \cdot n_{i,j}) \times a_j\} + N_{cmp} \cdot a_{cmp} \quad (5)$$

Thus, the total datapath area is given by Eq. (6).

$$A_{total} = A_S + A_M \quad (6)$$

As explained above, the function call optimization problem is defined as an integer programming problem, which finds x_i's minimizing the total area in Eq. (6) with meeting Eq. (3).

Given a program with function calls, designers can obtain the optimal hardware structure by solving the problem defined above. To find the optimal solution, some commercial solvers for integer linear programming 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.

3.2 Overall Synthesis Flow

As defined in Sect. 3.1, our 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_{main} and 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 should be conducted as follows.

1. Flatten the hierarchy of functions into a single level by inlining.
2. Execute behavioral synthesis for each function to obtain $n_{i,j}$, s_{main}, and s_i.
3. Solve the proposed problem with varying the constraint.
4. Execute behavioral synthesis for each module.

The synthesis steps 3 and 4 need to be repeated with varying the constraint. The key point is that the proposed method can be used as a tool which significantly reduces the design space to be explored. For each constraint, our method finds an optimal structure without actually synthesizing the circuits, so designers do not have to run behavioral/logic synthesis for all the possible designs.

3.3 Discussion on the Constraint

In Sect. 3.1, we use the number of states as a metric for control path complexity. According to the work in [4], the number of operations to be scheduled as well as that of states is an important factor for deciding the control path complexity. Also, it is widely known that the behavioral synthesis runtime significantly depends on the number of operations. Therefore, one may think that the number of operations should be used as a constraint in the function call optimization problem. In fact, there is some relationship between the number of states and that of operations, i.e., more operations tend to result in more states. Then, using the number of states as a constraint implicitly constrains that of operations to be scheduled and executed in each module. However, it is also easy to explicitly constrain the number of operations with small modification of the ILP formulation as follows.

o_i: the number of operations in a module synthesized from function F_i individually
o_{main}: the number of operations in main module when all the functions are exlined
O_M: the number of operations in main module
O_S: the number of operations in sub module
O_{const}: the designer-specified constraint on the number of operations

$$O_M = o_{main} + \sum_i (1 - x_i) \cdot o_i \cdot c_i \quad (7)$$

$$O_S = \sum_i x_i \cdot o_i \quad (8)$$

$$O_M \leq O_{const}, \quad O_S \leq O_{const} \quad (9)$$

The new constraint Eq. (9) can be used instead of, or in addition to, Eq. (3).

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 section, looser constraint on the number of states tends to result in an increase in execution cycles, while the area tends to be reduced. This indicates that designers can efficiently explore

performance-area trade-off points by varying the constraint on the number of states.

4. Experiments

We conducted two sets of experiments to demonstrate the effectiveness of the proposed method for two benchmark programs; one is `sqrt` (the square-root), and the other is `ludcmp` (LU decomposition for a matrix). Note that these programs are rather large (more than 1,000 lines of C code) compared with the DSP kernels which have been traditionally used in the past literature on behavioral synthesis. Each program consists of a main function and several double-precision floating-point arithmetic functions such as addition (`add`), multiplication (`mul`), and so on.

In the experiments, eXCite from [5] was used for behavioral synthesis. The clock frequency constraint was set to 50 MHz. Table 1 shows the number of states for each function and that of times which each function is called in each benchmark program.

Table 1 The number of states and times called by main module in each sub module.

		main	add	sub	mul	div	lt	le	eq	int_to_double
the number of times called by main module	sqrt	—	1	3	3	2	1	1	1	0
	ludcmp	—	1	6	5	3	0	3	0	2
the number of states	sqrt	61	61	61	33	55	9	9	7	7
	ludcmp	153								

Table 2 Experimental results for `sqrt`.

constraint on states	partitioning	states	gate count	clock period (ns)	execution cycles	execution time (μs)	beh. syn. time (s)	area-delay product ($\times 10^6$)
exline	main	61	601,256	48.7	1,915	93.3	819	56.1
	add	61						
	sub	61						
	mul	33						
	div	55						
	lt	9						
	le	9						
	eq	7						
160–210	main, add, lt, eq	124	377,087	46.6	1,947	90.1	1,655	34.0
	sub, mul, div, le	152						
220	main, lt, le, eq	73	334,216	52.1	1,953	101.7	2,113	34.0
	add, sub, mul, div	204						
230	main, lt, eq	72	334,833	51.9	1,989	103.1	2,187	34.5
	add, sub, mul, div, le	211						
240–530	main	73	333,343	52.8	2,032	107.3	2,527	35.8
	add, sub, mul, div, lt, le, eq	223						
540– (inline)	main, add, sub, mul div, lt, le, eq	—	—	—	—	—	—	—

Table 3 Experimental results for `ludcmp`.

constraint on states	partitioning	states	gate count	clock period (ns)	execution cycles	execution time (μs)	beh. syn. time (s)	area-delay product ($\times 10^6$)
exline	main	153	1,143,410	48.5	5,110	247.7	1,134	283.2
	add	61						
	sub	61						
	mul	33						
	div	55						
	le	9						
	int_to_double	7						
210	main, le, int_to_double	169	875,450	51.6	5,350	275.9	2,077	241.5
	add, sub, mul, div	204						
220–290	main, le	172	876,173	61.5	5,601	344.7	3,198	302.0
	add, sub, mul, div, int_to_double	217						
300–920	main	173	872,900	61.6	5,678	350.0	2,901	305.5
	add, sub,mul, div, le, int_to_double	235						
930–950	main, add, sub, mul, div, le	—	—	—	—	—	—	—
	int_to_double	7						
960– (inline)	main, add, sub, mul div, le, int_to_double	—	—	—	—	—	—	—

Next, the proposed method gave the optimal partitioning of functions to be inlined and to be merged, with varying the constraint on the number of states. It took less than one second to find the optimal solution with an exhaustive search algorithm.

For each result given by the proposed method, we ran 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 ISE from [6] was used for logic synthesis and place-and-route. Register-transfer level simulation was also performed to measure the execution cycles.

Tables 2 and 3 summarize the synthesis and simulation results for `sqrt` and `ludcmp`, respectively. The first column denotes the constraint on the number of states. 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 last column describes *area-delay product*. Area-delay product is defined as the product of area (gate count) and execution time. Since there is generally a trade-off between area and execution time, area-delay product is useful to comprehensively evaluate the designs. The results with function exlining are also shown for comparison. In both sets of experiments, function exlining gives the largest area due to sharing no resources between modules. On the other hand, the number of execution cycles is the shortest without necessity to send a parameter to determine the function to be executed.

As relaxing the constraint, on the contrary, the area tends to be smaller. It is because some resources required by several functions are shared by merging the functions into the same sub module, which also reduces the control path area. However, the number of execution cycles increased because of inter-module communication overhead. When all the functions are inlined, which is described in the lowest row in both Tables 2 and 3, and when the constraint is between 930 and 950 in Table 3, the number of states in main module became too large to normally finish behavioral synthesis.

When the constraint is 230 in Table 2, the area is slightly larger than that when the constraint is 220. A similar discrepancy is also seen in Table 3. This is mainly due to the following reason. As described in Sect. 3, the proposed method determines the functions to be inlined and exlined/merged based on estimated datapath area in Eqs. (4) and (5). Indeed, the estimated area at constraint 230 is slightly (approx. 400 gates) smaller than that at constraint 220. On the other hand, the area in Table 2 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 Eqs. (4) and (5). This problem needs to be handled more carefully in future.

For simplicity, we assumed that no function except the main function calls other functions and there exists only a single sub module in addition to main module. In spite of these simplifications, however, it should be noted that our method outperforms the traditional techniques.

5. Conclusions

This paper has proposed a method to optimally determine functions to be inlined and ones to be exlined/merged into sub module, which makes behavioral synthesis efficient. We have formally defined this method as an integer programming problem. Two sets of experiments have demonstrated its effectiveness such as the reduction in the circuit area by up to 44.6%, compared with a traditional method.

In the future, we plan to test our method more extensively using various application programs. Extension towards multiple modules is another future work.

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] K. Wakabayashi, "CyberWorkBench: Integrated design environment based on C-based behavior synthesis and verification," Int. Symp. VLSI Design, Automation and Test, pp.173–176, 2005.
[3] F. Vahid, "Partitioning sequential programs for CAD using a three-step approach," ACM TODAES, vol.7, no.3, pp.413–429, July 2002.
[4] 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.
[5] Y Explorations, Inc., http://www.yxi.com/
[6] Xilinx, http://www.xilinx.com/