Multiple Behavior Module Synthesis Based on Selective Groupings

Ju-Hwan Yi, Hoon Choi, In-Cheol Park, Seung Ho Hwang and Chong-Min Kyung
Department of Electrical Engineering
Korea Advanced Institute of Science and Technology
373-1, Kusong-dong, Yusong-gu, Taejon, 305-701, Korea
{yjh, hchoi}@snoopy.kaist.ac.kr, icpark@ee.kaist.ac.kr

Abstract

In this paper, we present an approach to synthesize multiple behavior modules. Given n *DFGs to be implemented, the previous methods scheduled each of them sequentially, and implemented them as a single module. Though the method is appropriate for sharing the functional units, it ignored the following two aspects: 1) different interconnection patterns among DFGs can increase the interconnection area and delay of the critical path, 2) the sequential scheduling of DFGs has a difficulty in considering the effects on the other DFGs not scheduled yet. We show an efficient way to solve the problems using a selective grouping method and the extensions of the traditional scheduling methods. The experimentation reveals that the result obtained by the proposed method is better to reduce interconnection area and to meet the timing constraints than those obtained by the previous methods.*

1. Introduction

Due to the advance of VLSI technology, a lot of ASICs (Application Specific Integrated Circuit) are being used in various systems. Compared to the general purpose processors, an ASIC can satisfy various constraints such as performance, area, and power, with low cost by finding an optimal architecture for a given application. However, as the complexity of applications is increased, more flexibility is required to accommodate design errors and specification changes which may happen later. Since ASICs are specially designed for one behavior, it is difficult to adopt any changes at the later design stage. In contrast, a programmable processor can be easily adapted to different applications by changing only the programs. This is the reason why ASIPs(Application Specific Instruction set Processor) are widely accepted in a number of systems.

Generally an ASIP has a programmable architecture which is tuned to a number of different behaviors. Choosing an optimal instruction set proper for specific applications under constraints such as chip area and power consumption is crucial in maximizing the performance of the ASIP. This leads to several researches to look for tools which analyze several input behaviors and synthesize a single programmable architecture that is good for the throughput of the behaviors.

An integrated ASIP design system was proposed in [1][2][3]. Given example programs written in C and typical data as inputs, the system profiles the programs with the given data set, and decides an instruction set and a hardware architecture based on the profiles. It also automatically generates software development tools for the ASIP such as compiler and simulator. However, the architecture of an ASIP generated is based on the GCC's abstract machine model, and only the subset of GNU intermediate language is taken as the instructions of the ASIP. In other words, the supported instruction set is restricted by the intermediate language of GCC.

An evolution programming approach is applied for the behavior-level area efficient design of ASIP[4]. The method, based on a given behavioral-level kernel, randomly transforms each of the given DFGs, and then the behavioral kernel is used in the evolution process to guide the survival of DFGs. Finally instead of the given DFGs, the surviving DFGs are used to synthesize a programmable architecture. Though it could lead to an area-efficient design which can support all the input behaviors, it did not consider the connection cost and performance constraints. We may not combine all the DFGs into one due to the interconnection cost and delay. Hence, we have to selectively partition the DFGs into several groups, and each group is implemented into a single hardware module that can perform all the operations in the group. The previous method ignored the necessity of grouping. The area and delay of interconnection can not be ignored as the chip becomes complex.

The recent work presented in [7] considered the grouping problem where *n* control-data flow graphs are bundled into at most *m* groups. It is different from our work in that its target architecture is the application specific programmable processor (ASPP) not the ASIP, our target architecture. Therefore, it has almost no consideration on the relation between the synthesized hardware and the corresponding instruction. In addition, it has the following two problems. First, in application grouping process, a probabilistic framework based on the incompatibility between an application and a group was used. This means that the grouping result is highly dependent on the incompatible measure. Furthermore, the applied measure for the incompatibility was based on a simple comparison between the area of each application and that of the predicted group area. In our method, a more accurate measure, *grouping gain*, is used in deciding the group of each application. Second, following the application grouping, each group is synthesized into a separate hardware. As the synthesis of an ASIP has a difficulty in considering all the applications and their respective constraints is simultaneously, the work synthesized the ASPP by considering one application at a time. Thus, ordering of applications impacts the synthesized results. In contrast, we use a new method that considers all the applications and their respective constraints simultaneously.

This paper deals with the grouping of several DFGs, which is used in our ASIP synthesis system, *Partita*. After we describe the overview of Partita in section 2, the proposed grouping method is described in section 3. Experimental results and conclusions are described in section 4 and 5.

2. Overview of Partita

The target architecture of Partita is a highly programmable ASIP for DSP applications. The application program is located in the external program memory as shown in Fig. 1. Simple instructions are executed with the assistance of hardwired controllers, and complex instructions are controlled by a μ-ROM. The decoded result of a complex instruction includes the start address of the corresponding μ-routine in the μ-ROM.

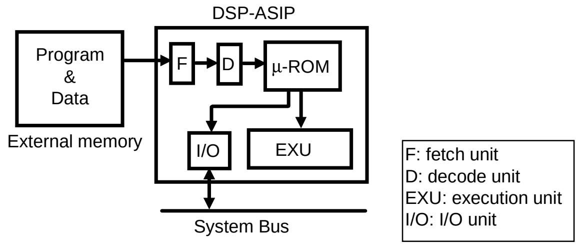

Fig. 1: Target architecture of Partita

Partita supports three classes of instructions according to the complexity. First, P class contains instructions that are not only primitive but also essential in all applications, i.e., simple arithmetic instructions and control instructions like branch and call. The P class instructions are always supported in all the generated ASIPs by a simple execution kernel and a hardwired controller. Second, B class is composed of instructions that are more complex than P class instructions but not time critical. The B class instructions are implemented by a combination of instructions of P class and controlled by a μ-ROM. B class is provided to minimize the external memory required. Lastly, S class is a set of instructions that are time critical as well as complex. Hence S class instructions are implemented by special hardware units called *S-HWs.*

The reason behind this classification is as follows. As applications are changed, the instructions in B and S class are changed for optimal performance in the given constraints. In addition, as instructions in B and S class are generally complex, it is not easy for the application programmers to use such instructions in order to modify the automatically generated programs for the purpose of algorithm changing or debugging. Fortunately, what to be changed at the late stage of design or after the fabrication is generally not the data path part but the control part. Since the frequent operations of data path are mapped into S and B class and the control operations are mapped into P class, we can accommodate the needed changes only using P class instructions. The simplicity of the operations makes P class instructions easily understood and used by the application programmers. There is no predefined set of S and B class instructions. Thus the selecting time-critical parts of the given algorithm as S class instructions offers more freedom than the previous methods.

The inputs of Partita are an application written in silage and typical input data for the application. We sample-run the application with the given typical input data to know the running frequency of each line in the silage description. The system selects parts of description which run frequently as candidates for B and S class instructions. Then, instructions of S class are partitioned into several groups with considering similarity among instructions. A hardware module to implement each group of S class, S-HW, is synthesized and μ-codes for B class instructions are generated. All instructions are encoded and other necessary hardware modules such as decoding unit and fetch unit are produced. The μ-ROM containing all μ-codes are then optimized. The program is also synthesized by covering the DFG of the given application with the generated instructions. Fig. 2 shows the flow diagram of Partita.

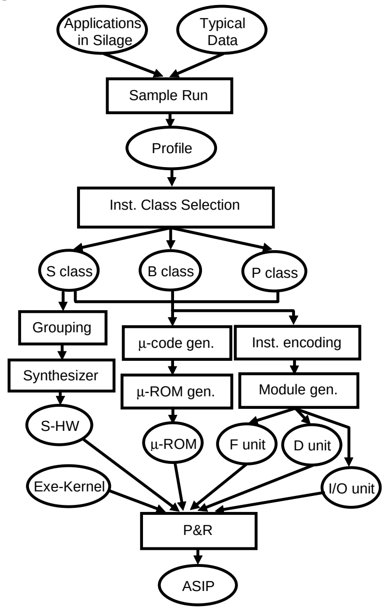

Fig. 2: Flow diagram of Partita

3. Grouping Method for S-HW Generation

3.1 Motivation of Grouping

After the sample running, parts of DFG descriptions which run frequently are selected as candidates for B and S class instructions. We select time-critical parts of them as S class if they meet the various constraints, i.e., area and power, and then we have several sub-DFGs to be implemented in S-HWs. There are three possible ways to implement them. First, the simplest way is to implement each graph into a separate S-HW. Though the method is simple and straightforward, it often results in an implementation of the largest area because the resources in the different S-HWs are not shared at all. Second, we can merge all the sharable DFGs, i.e., DFGs in the mutually exclusive branches, into one, and build a single S-HW for the merged DFGs. This maximizes the resource sharing, especially functional units. However, the more DFGs are merged, the more the interconnections among functional units become complex. Thus the interconnection cost, i.e., area of wires and MUXs, can be increased. In addition, the delay is increased because of the raised wiring load and MUX depth. This problem becomes severe as the throughput requirements are significant. Lastly, we can use a hybrid approach between the first and the second method; selectively merge the DFGs only if the merging does not violate the timing constraints and the increased interconnection cost is acceptable. For example, let's assume the three DFGs shown in Fig. 3 are to be mapped into S-HWs under the timing constraints 4, 3 and 3 steps respectively. We also assume they are mutually exclusive to one another. If we use the first method, one to one mapping, three S-HWs shown in Fig. 4 are obtained. The three S-HWs use three adders, three multipliers and no MUXs. In the second method, the three DFGs are merged into one, and as a result we obtain a S-HW in Fig. 5, where an adder, a multiplier and six MUXs are used. Two of the six MUXs are serially connected. Compared with the first method, the S-HW uses much less functional resources, but has much more interconnections and MUXs. Furthermore the two MUXs which are serially connected may violate the timing constraints. The possible groupings of the third method are 1) DFG-A and DFG-B, 2) DFG-B and DFG-C, and 3) DFG-A and DFG-C. The datapaths each of which implements a group are shown in Fig. 6. To cover all the three DFGs, we should select one from Fig. 6 and one from Fig. 4. We select the first one from Fig. 6, i.e., a datapath for DFG-A and DFG-B, and the last one from Fig. 4, i.e., a datapath for DFG-C, because it results in the minimal area implementation satisfying the timing constraints. To find the best solution, therefore, we should selectively group the DFGs that are similar to one another.

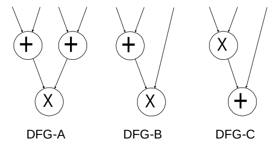

Fig. 3: Three example DFGs

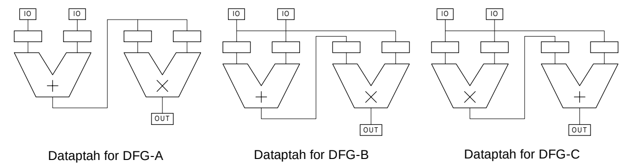

Fig. 4: Results of the first method

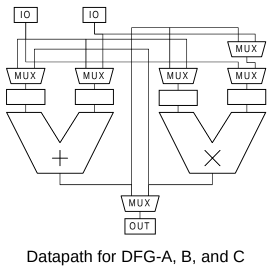

Fig. 5: Result of the second method

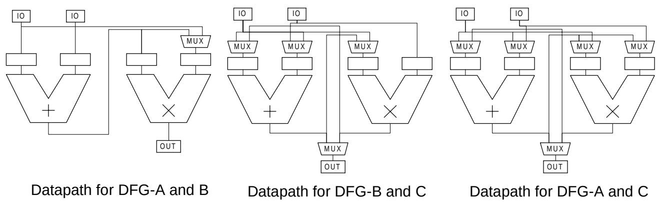

Fig. 6: Three possible groups of the third method

3.2 Grouping Gain

Suppose that the minimum area implementation of DFG A and B is I_A and I_B, respectively. And assume that the minimum area implementation of merged DFGs is I_{AB}. The *grouping gain* between DFG A and B is defined as:

$$G_{AB} = \begin{cases} Area(I_A) + Area(I_B) - Area(I_{AB}), & \text{if } I_{AB} \text{ meets constraints} \\ 0, & \text{otherwise} \end{cases}$$

However, since the implementation procedures take much time, we approximate the gain by using the number of functional units and the number of interconnections after the scheduling and allocation procedures. We can further reduce the time consumption by using the estimations like the one proposed in [5].

3.3 Grouping

Given n sub-DFGs, there are around 2^n groups to be considered. As it is difficult to take into account all the possible groups, we selectively merge two sub-DFGs at one time to reduce the grouping time required. Starting with the given DFGs and their area estimates, we calculate the gains between each pair of DFGs to build a *gain graph*. The gain graph consists of N and E. N is a set of nodes where each node denotes a DFG. E is a set of weighted edges between two nodes of N, and the edge weight represents the grouping gain of the two nodes. Fig. 7 shows the gain graph for three DFGs in the Fig. 3, where node A, B, and C represent DFG-A, DFG-B, and DFG-C, respectively, and the value on a edge is the grouping gain between nodes.

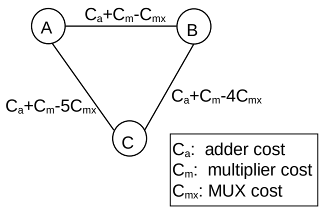

Fig. 7: Gain graph of Fig. 3

Two nodes connected by an edge whose weight is the largest are merged into a single node. We update the weight of edges that are connected to the merged node. This procedure is repeated until there are not positive weighted edges any more. In Fig. 7, at the first step the edge between node A and B is selected because its weight is the largest. After merging node A and B, the gain graph is re-constructed to update edge weights. Since there are no more positive weighted edges in the graph, we stop the procedure. In this way, we get two groups, AB and C, for the hardware blocks. The timing complexity of the proposed method is $O(n^3)$ where n is the number of DFGs.

3.4 Scheduling of merged DFGs

The previous method in [6, 7] synthesized a HW of merged DFGs as follows: First, it schedules each DFG separately. Second, it performs the allocation of the first DFG. Third, the second DFG is allocated with considering the first time step of the second DFG as the next time step of the first DFG's last time step. The third procedure is repeated until all the DFGs are allocated. However, the scheduling of a DFG has an influence on the schedulings of other DFGs to be merged with. Therefore, the scheduling of a DFG should consider the effects on the schedulings of other DFGs. Fig. 8 illustrates the problem.

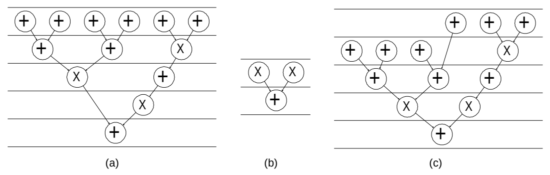

Fig. 8: Problems of merged DFGs scheduling

Fig. 8-(a) and (b) show the scheduling results of two DFGs without considering the effects between them. The scheduling in Fig.8-(a) requires one multiplier and six adders, while two multipliers and one adder are necessary for Fig. 8-(b). Thus, the merged DFG without considering the effects between them requires six adders and two multipliers. However, in fact the scheduling of the merged DFGs can be satisfied by using only two multipliers and three adders as shown in Fig. 8-(c) if the effects between them are considered. The followings deal with how to take into account scheduling effects between DFGs in a group.

3.4.1 ILP formulation

An ILP formulation for an optimal scheduling of merged DFGs is presented in this section. We extend the ILP formulation in [8] to schedule the merged DFGs. The objective is to meet the timing constraints with minimum hardware resources. We assume the propagation delay of every operation is one cycle for the sake of convenience.

First, we apply ASAP and ALAP schedulings to each DFG to know the *start time* and the *require time* of each operation. Suppose there are p DFGs, and each DFG l contains N_l operations, where $1 \le l \le p$. Each of the operations is labeled as O_{li}, where $1 \le i \le N_l$. In DFG l, a precedence relation between two operations is denoted by $O_{li} \rightarrow O_{lj}$, where O_{li} is the immediate predecessor of O_{lj} in DFG l. The start time and required time of each operation O_{li} is denoted as S_{li} and L_{li}, respectively. And there are m types of function units each of whose cost is C_{ti} , $1 \le i \le m$. The variables used in formulation are as follows:

- M_{ti} are integer variables that denote the number of functional units of type t_i needed.
- x_{lij} are 0, 1 integer variables associated with O_{li}
- x_{lij} = 1 if O_{li} is scheduled into step j, otherwise 0.

The objective function is $\sum_{i=1}^{m} C_{ti} \cdot M_{ti}$, and subject to

$$\sum_{\substack{i=1 \\ O_{li} \in FU_{tk}}}^{N_l} x_{l,i,j} - M_{tk} \le 0 \ \text{ for } 1 \le l \le p,\ 1 \le j \le s, \text{ and } \ 1 \le k \le m$$

$$\sum_{j=S_{li}}^{Ll_i} x_{l,i,j} = 1 \text{ for } 1 \le l \le p,\ 1 \le i \le N_l$$

$$\sum_{j=S_{li}}^{L_{li}} j \cdot x_{l,i,j} - \sum_{j=S_{lk}}^{L_{lk}} j \cdot x_{l,k,j} \le -1 \text{ for all } 1 \le l \le p,\ O_{li} \rightarrow O_{lk}$$

3.4.2 Force Directed scheduling

The force directed scheduling [9] is extended to schedule merged DFGs by inserting a *pseudo node(PN)* between two DFGs. The pseudo node does not perform any operation, and has a fixed time step. An operation of a sub-DFG can not move across the pseudo-node.

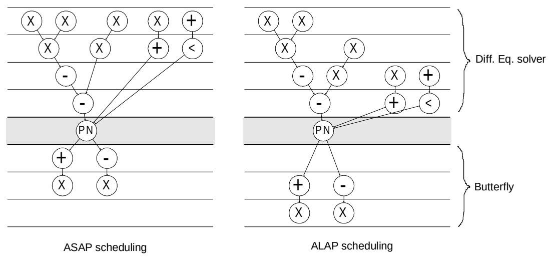

Fig. 9: ASAP and ALAP schedulings of merged DFGs with pseudo node

To illustrate the method, we show an example in Fig. 9 where a merged DFGs containing two DFGs, namely a differential equation solver and a butterfly calculation, is scheduled by ASAP and ALAP, respectively. Timing constraint of the differential equation solver is 4 and that of the butterfly calculation is 3. The pseudo node in Fig. 9 is used to fix the boundary between two DFGs. We apply ASAP and ALAP schedulings to the two behaviors with fixing the time step of *PN*. After the ASAP and ALAP schedulings, we can get time frames of all the operations as shown in Fig. 10.

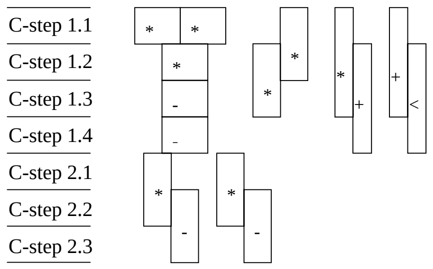

Fig. 10: Time frames of all the operations in Fig. 9

With the time frames, we do the remaining procedures of the force directed scheduling like the conventional case of one DFG. With the scheduling result, it is not hard to estimate the area of implementation and check whether the result can meet the given timing constraints.

4. Experimental Results

We have incorporated our grouping method in HYPER system [10]. Five benchmark circuits are used to see the grouping effects. The minimum area implementation of each benchmark circuit is summarized in Table 1.

Table 2 shows the grouping results; we measured the area of functional units and interconnections, and the maximum depth of serially connected MUXs for three different groupings. The first grouping entitled Imp1 in Table 2 merged all the five DFGs into a single S-HW. In the row entitled Imp2, we show the result obtained by partitioning them into two groups; the iir filter and the noise canceller are merged into one and the others are merged into the other one. The row entitled Imp3 shows the result obtained by a different grouping which was selected by the proposed grouping gain. The wdc filter, iir filter, and fir filter are merged into one and the others are merged into the other one. Table 2 shows the followings: 1) Imp1 shows the best result in terms of the area of functional units. This is due to the fact that as all the DFGs are merged into one, the functional units are shared as much as possible. 2) Imp2 and Imp3 show the importance of proper grouping. Imp2 shows the larger MUX area than that of others, while that of Imp3 is the smallest. This is caused by the fact that the DFGs in each group of Imp3 have similar interconnection patterns, but those of Imp2 have not. Therefore, the grouping of DFGs having the similar interconnection patterns reduces the interconnection costs. 3) The maximum depth of MUXs in Imp3 is two, which is less than that of Imp1 and Imp2. This is important to meet the timing constraints. Thus, though the Imp3 uses more area for functional units than the imp1, Imp3 has more chance to meet the required timing constraints. The experiments indicate that the proposed grouping is effective in reducing the interconnection costs.

5. Conclusion

An approach to synthesize multiple behavior modules using a selective grouping is presented in this paper. Given *n* DFGs to be implemented, we selectively partition them into several groups with considering not only the hardware resources such as functional units and interconnections but also the given timing constraints. To consider the effect to the other DFGs, the DFGs in a group are scheduled simultaneously using an ILP formulation or an extended version of the force directed scheduling. The experimental results showed that the implementation obtained by the proposed grouping is better in reducing interconnection cost and meeting the timing constraints than that obtained by merging all the DFGs into a single group.

References

[1] J. Sato, M. Imai, T. Hakata, A.Y. Alomary and N. Hikichi, “An Integrated Design Environment for Application Specific Integrated Processor,” *Proc. of ICCD*, pp. 414-417, 1991.

[2] M. Imai, A. Alomary, J. Sato and N. Hikichi, “An Integer Programming Approach to Instruction Implementation Method Selection Problem,” *Proc. of Euro-DAC*, pp. 106-111, 1992.

[3] A. Alomary, T. Nakata, Y. Honma, M. Imai and N. Hikichi, “An ASIP Instruction Set Optimization Algorithm with Functional Module Sharing Constraint,” *Proc. of ICCAD*, pp. 526-532, 1993.

[4] W. Zhao and C.A. Papachristou, “An Evolution Programming Approach on Multiple Behaviors for the Design of Application Specific Programmable Processors,” *Proc. of ED&TC*, pp. 144-150, 1996.

[5] A. Sharma and Rajiv Jain, “Estimating Architectural Resources and Performance for High-Level Synthesis Applications,” *Proc. of DAC*, pp. 355-360, 1993.

[6] W. Zhao and C.A. Papachristou, “Synthesis of Reusable DSP Cores Based on Multiple Behaviors,” *Proc. of ICCAD*, pp. 103-108, 1996.

[7] K. Kim, R. Karri and M. Potkonjak, “Synthesis of Application Specific Programmable Processors,” *Proc. of DAC*, pp. 353-358, 1997.

[8] J.H. Lee, Y.C. Hsu, and Y.L. Lin, “A New Integer Linear Programming Formulation for the Scheduling Problem in Data path synthesis,” *Proc. of ICCAD*, pp. 20-23, 1989.

[9] P.G. Paulin and J.P. Knight, “Force-Directed Scheduling for the Behavioral Synthesis of ASIC's,” *IEEE Transactions on Computer-Aided Design*, vol. 8, pp. 661-678, June 1989.

[10] C. Chu, M. Potkonjak, M. Thaler, and J. Rabaey, “HYPER: An Interactive Synthesis Environment for High Performance Real Time Applications,” *Proc. of ICCD*, pp. 432-435, 1989.

Table 1: Benchmark circuits

	FU			MUX			buf	bus	reg
	+	-	*	no	inputs	depth	no	no	no
fir filter	1	0	1	3	6	1	8	7	27
iir filter	1	1	1	5	12	1	15	15	34
wdc filter	1	1	1	7	17	1	18	17	48
noise canceller	1	1	1	6	15	2	15	12	45
wavelet	1	1	1	4	9	1	13	12	51

Table 2: Grouping results

		FU			MUX			buf	bus	reg
		+	-	*	no	inputs	depth	no	no	no
Imp1	total	1	1	1	31	118	3	27	24	105
Imp2	g1	1	1	1	18	58	3	24	21	59
	g2	1	1	1	18	62	3	21	20	84
	total	2	2	2	36	120	3	45	41	143
Imp3	g1	1	1	1	13	44	2	22	21	76
	g2	1	1	1	9	22	2	19	16	67
	total	2	2	2	22	66	2	41	37	143