Thread Partitioning Method for Hardware Compiler Bach

Mizuki TAKAHASHI[†‡] Nagisa ISHIURA[‡] Akihisa YAMADA[†] Takashi KAMBE[†]

[†]Design Technology Development Center
IC group
SHARP Corporation
Tenri, Nara, 632-8567 Japan
e-mail: {mizuki,yamada,kambe}@icg.tnr.sharp.co.jp

[‡]Department of Information Systems Engineering
Graduate School of Engineering
Osaka University
Suita, Osaka, 565-0871 Japan
{mizuki,ishiura}@ise.eng.osaka-u.ac.jp

Abstract—This paper presents a method for *thread partitioning* for a hardware compiler *Bach*. Bach synthesizes RT level circuits from a system description written in *Bach-C* language, where a system is modeled as communicating processes running in parallel. The system description is decomposed into *threads*, i.e., strings of sequential processes, and then converted into synthesizable behavioral VHDL models. The proposed method attempts to find a partitioning of a given system description into threads that maximize resource sharing among processes in the threads. Experiments on two real designs show that the circuit sizes were reduced by 3.7% and 14.7%. We also show the detailed statistics and analysis of the size of the resulting gate level circuits.

I. Introduction

LSIs embedded in various consumer products, such as digital audio and video players and mobile terminals, are key devices characterizing their functionality, performance, and market prices. Rapid progress of IC process technology has enabled integration of large digital systems on a single chip and a large part of the functionalities are now implemented by hardware. However, due to strong time-to-market pressure, increase of design period in accordance with the increase in the design size is becoming unacceptable. On the contrary, it is required that larger systems be designed within shorter period of time.

Behavioral synthesis [1, 2] is now one of the indispensable design automation techniques to ease this "design crisis" problem. However, one limitation of the behavioral synthesis is that it is often hard to deal with large systems *directly*. One reason for this is that existing behavioral synthesizers are not able to produce efficient circuits for large scale behavioral specifications consisting of thousands lines. Another reason is that the behaviors of entire systems do not always fit into the model of a sequential process. Some systems may be better modeled, in terms of understanding and in terms of succinctness of the descriptions, by a set of parallel or concurrent processes communicating with each other. For these reasons, large system specification must be broken down into smaller pieces which are well modeled as sequential processes and yet efficiently handled by behavioral synthesizers.

As an attempt to extend the frontier of design automation beyond behavioral synthesis, we have developed a Bach hardware compiler [3], which synthesizes circuits from system descriptions in Bach-C language. Bach-C is an extension of the programming language C which features parallel execution of sequential processes communicating with each other. A given Bach-C description is decomposed into *threads*, strings of sequential processes, which are synthesized by behavioral synthesizer along with circuits for communication.

The Bach compiler has been already used in several real designs, from which two major problems have been reported. One is that threads sometimes become too large for behavioral synthesis, and the other is that resource sharing among processes in the same thread is sometimes poor so that resulting circuits become larger than expected. It turned out that these crudities are attributed to a naive thread partitioning algorithm, in which threads are constructed simply depending on the order of the appearance of the processes in the description.

This paper presents a refined way of partitioning a given Bach-C description into threads. The problem of thread partitioning, which takes the thread size limitation and hardware resource sharing into account, is formulated as an optimization problem. Then the problem is solved by means of integer linear programming. In a preliminary experiment on two real examples, the circuit size reduction of 3.7% and 14.7% was achieved.

In the following section, we briefly introduce the Bach-C language and how system descriptions in Bach-C are compiled by Bach. Then we formalize the thread partitioning problem and show how the problem is solved by integer linear programming in subsequent sections. Finally, we show experimental results on two design examples and discuss the effectiveness of the proposed method.

II. Hardware Compiler Bach

Bach-C [3] is a system description language based on ANSI C with extensions for expressing explicit parallelism, communications between parallel processes and bit-width specification of data types. The semantics for parallelism and communication are based on CSP [4].

Fig. 1 shows a simple example of Bach-C description with `par` and `send`/`receive` statements. Statements are executed sequentially, like in C language, until the `par` statement is encountered. The `par` statement starts the parallel execution of the sequential blocks underneath it. We call each sequential block a *process*. In this example, execution of process *A* is

```
x = a * b;
y = x + c;          process A
par {
  {
    p = x + y;
    q = p - d;
    r = receive(c1);    process B
    s = q + r;
  }
  {
    u = x * y;
    send(c1,u);         process C
    v = u - y;
  }
}
z = s + v;
k = k + 1;          process D
```

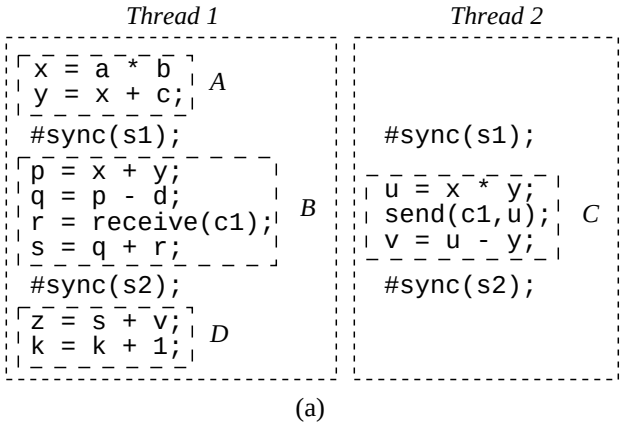
Figure 1: Bach-C description.

followed by the parallel execution of processes *B* and *C*. The control is passed to process *D* after both *B* and *C* are finished. `send` and `receive` statements are used to express communications between parallel processes. The argument `c1` is a channel, through which data are transmitted. The communication is synchronized: If the receive statement is reached in process *B* before the send statement in process *C*, the receive waits for the send, and vise versa.

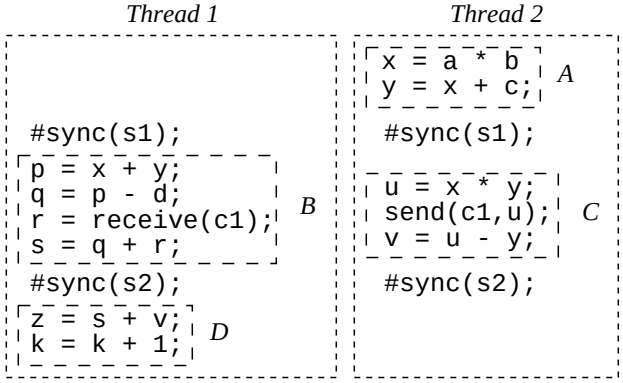
Bach system takes a system description in Bach-C and synthesizes RT level circuit models using a behavioral synthesizer. In more precise, Bach compiler converts the Bach-C description into a set of sequential VHDL processes along with handshake circuits and pass them to the behavioral synthesizer. One of the major tasks in compilation is to partition the given parallel-serial construct into sequential processes.

The simplest way of partitioning is to break the system description down into individual processes: The description in Fig. 1 may be decomposed into four processes *A*, *B*, *C*, and *D*, each of which will be synthesized separately. Although this strategy may work if each process is large enough, module instantiation for large number of small processes would result in an inefficient implementation with lots of idle hardware resources. Instead, the parallel-serial construct can be partitioned into *threads*, which are strings of sequential processes. For example, the model in Fig. 1 can be decomposed into two threads as shown in Fig. 2, based on the property that only processes *B* and *C* are subject to parallel execution. `#sync` is a meta statement (inserted only for synthesis and is not open to users) for synchronization. It works in the same way as `send`/`receive` statement except that no data are passed.

There are many ways of thread partitioning by which the configurations of the resulting circuits differ largely. This paper focuses on how we can make the best selection.



(a)



(b)

Figure 2: Thread partitioning.

III. Thread Partitioning Problem

A. Thread Partitioning Based on Resource Sharing

Let us examine the example in Fig. 2. Partitioning of (a) is obtained by a naive method, which was employed in the initial version of the Bach compiler. First, process A is put into Thread 1. Then the parallel processes B and C are accommodated into Threads 1 and 2, according to the order they appear in the source description. Finally, D is placed into Thread 1.

Obviously, this partitioning is not preferable in terms of the hardware cost. Processes A and C have multiplications. In partitioning (a), we need two multipliers, each of which is used only once. On the other hand, (b) needs only one multiplier. In this way, thread partitioning should be done taking the hardware resource sharing into account.

Another factor to be considered is the complexity or the size of each thread. Although this is highly dependent on the synthesizer, behavioral synthesis of large models often leads to enormous computation cost and sometimes it fails because of exhaustion of the computation resources. (We are actually suffering from this problem.) If not so serious, large models will debase the quality of optimization. Swollen control logic is another well-known problem. Thus, we must keep the size of each thread moderate, while attempting to group processes with the same operations as much as possible.

B. Problem Formulation

Our goal here is to find a partitioning of a given description into threads so that the total cost of hardware resources is minimized and each thread satisfies the complexity constraints.

Here, the number of threads is another subject to be discussed. We may face a choice between 1) the best solution in terms of the hardware cost and 2) the second best solution with smaller number of threads. However, the cost of hardware associated with thread is considerably large as compared with that of the functional units, we may assume that the feasible solution with the minimum number of threads gives practically the best partitioning. Thus, we will only search for the partitioning with the minimum number of threads that satisfies the complexity constraints.

Suppose a given Bach-C description consists of N processes $P = \{p_0, p_1, \ldots, p_{N-1}\}$. The serial-parallel configuration of the description is expressed by a directed graph $G_s = (V, E)$, which is called a *sequence graph*. For example, Fig. 3 shows the sequence graph of the description in Fig. 1. Node $v_i \in V$ corresponds to a process p_i. Direct edge $e_{ij} = (p_i, p_j) \in E$ represents the sequential relation where p_j is executed after p_i. If there is an edge between two nodes, we can put the two processes corresponding to the nodes into the same thread, for it is guaranteed that they will never be executed in parallel.

Let $D = \{d_0, d_1, \ldots, d_{Z-1}\}$ be a set of hardware resource types appearing in the synthesized circuits. For example, assuming all the variables in Fig. 1 are 16 bit long, the minimum set D for implementing this description is $D = \{\text{adder(16)}, \text{multiplier(16)}\}$. Note that functional units of different precisions are considered to be of different types and thus adder(8) and adder(16) are distinguished, for example. The use of multi-function hardware resources is not considered and we only deal with single-function hardware resources in this paper. We assume that the implementation cost, denoted as $cost(d_i)$, is defined for each d_i. Such metrics as the gate count of d_i or the chip area of d_i, available in module libraries, works as $cost(d_i)$.

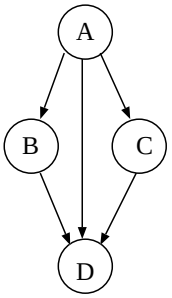


Figure 3: Sequential graph.

For each process p_i, we assume that its hardware cost information $n(p_i, d)$ and its complexity $comp(p_i)$ are defined. $n(p_i, d)$ represents the number of the hardware resources of type d required to synthesize process p_i. We assume the value of $n(p_i, d)$ is precomputed by synthesizing each p_i separately. As for the complexity of the process, we define $comp(p_i)$ to be the number of operations in p_i.

Thread partitioning $T = \{t_0, t_1, \cdots, t_{H-1}\}$ is a partition of P where no processes in the same partition are executed in parallel.

- $\bigcup_{t_k \in T} t_k = P$ and $t_{k_1} \cap t_{k_2} = \phi$ if $k_1 \neq k_2$.
- $\forall p_i, p_j \in t_k : e_{ij} \in E \vee e_{ji} \in E$.

Let $cost(t_k)$ and $comp(t_k)$ be expected cost of the hardware resources in the circuit synthesized from t_k and expected complexity of t_k, respectively. We compute $cost(t_k)$ by

$$cost(t_k) = \sum_{d \in D} cost(d) * \max_{p \in t_k}(n(p, d)).$$

Although the exact number of hardware resource d in the circuit for t_k is highly dependent on the synthesizer and there is no guarantee that it coincides with $\max_{p \in t_k}(n(p, d))$, we believe this still gives a good estimation. $comp(t_k)$ is also dependent on the synthesizer, but we use

$$comp(t_k) = \sum_{p \in t_k} comp(p)$$

as an estimation.

With the definition stated so far, our thread partitioning problem is defined as to find partitioning T with minimum H which satisfies

$$\forall t_k \in T : comp(t_k) \leq comp_{max}$$

and yet minimizing

$$cost(T) = \sum_{t_k \in T} cost(t_k)$$

where $comp_{max}$ is the upper limit of the complexity allowed for each thread.

IV. Solving the Thread Partitioning Problem by Integer Linear Programming

It is proved that the thread partitioning problem here is NP-hard (NP-complete): It is restricted to the bin packing problem [5] by setting $|D| = |P| + 1$, $n(p_i, d_i) = comp(p_i)$ for $i = 0, \cdots, N-1$ and $n(p_i, d_N) = 1$ and letting no pair of processes be executed in parallel.

Fortunately, the size of the problem to be solved is not very large. In our experience so far, N (the number of processes) is not more than 20.

Exhaustive search works for $N \leq 10$, but not for larger instances. On the other hand, we do not need to resort to an elaborate branch-and-bound search nor heuristics approximations. Furthermore, in order to leave flexibility for other factors, we decided to solve it by integer linear programming (ILP).

The base 0-1 variable to be solved is $x_{i,k}$ $(0 \leq i \leq N-1, 0 \leq k \leq H-1)$ which becomes 1 iff process p_i is assigned to thread t_k. Auxiliary integer variable $y_{l,k}$ $(0 \leq l \leq Z-1, 0 \leq k \leq H-1)$ represents the number of hardware resources of type d_l used in thread t_k.

The objective function to be minimized is the total cost of the hardware resources summed over all the threads.

$$Minimize : \sum_{l=0}^{Z-1} \sum_{k=0}^{H-1} cost(d_l) * y_{l,k}$$

There are four constraints for this problem.

1. Partition constraint:

   Each process must belong to one and only one thread.

$$\forall i (0 \leq i \leq N-1) : \sum_{k=0}^{H-1} x_{i,k} = 1$$

2. Sequential constraint:

   Only those processes in the sequential relation can be accommodated in the same thread.

$$\forall i, j, k (0 \leq i, j \leq N-1, 0 \leq k \leq H-1, \\ i \neq j, e_{ij} \notin E \wedge e_{ji} \notin E) : \\ x_{i,k} + x_{j,k} \leq 1$$

3. Cost constraint:

   This constraint computes the maximum of the number of hardware resources of type d_l in each thread t_k.

$$\forall i, j, k (0 \leq i \leq N-1, 0 \leq k \leq H-1, \\ 0 \leq l \leq Z-1) : \\ n(p_i, d_l) * x_{i,k} \leq y_{l,k}$$

4. Complexity constraint:

   The complexity of each thread must not exceed the predetermined limit $comp_{max}$.

$$\forall k (0 \leq j \leq H-1) : \\ \sum_{i=0,\cdots,N-1} Comp(p_i) * x_{i,k} \leq comp_{max}$$

```
{
  A();
  par {
    B();
    C();
  }
  par {
    D();
    E();
  }
  par {
    F();
    G();
  }
  H();
}
```

(a) Parallel-serial structure.

	add			inc	dec			mult
	10	19	20	13	10	11	20	10
A								
B	80							
C	2				2			
D			1				1	4
E	4			1		2		
F	2				2			
G		1	1				20	6
H	2							

(b) Hardware cost information.

Figure 4: Features of SFIL.

Note that we assume the number of threads H is fixed in this formulation. Since our goal is the solution with the minimum H, we start from the lower limit of H and increase it one by one until we find the minimum feasible solution. The lower limit of H is equal to the maximum width of the sequence graph and is easily computed.

V. Experimental Results

We implemented the thread partitioner based on the method stated so far and conducted some experiments. Obtained VHDL models by the partitioning were passed to behavioral synthesizer (Synopsys Behavioral Compiler) and then to logic synthesizer to attain gate level circuits.

Fig. 4 and 5 show the properties of the two system descriptions we tested. SFIL is a circuit for communication mainly consisting of digital filters. DINT too is a circuit for communication dedicated to interleaving of the data. Fig. 4 (a) and Fig. 5 (a) shows the parallel-serial structures of the descriptions. Fig. 4 (b) and Fig. 5 (b) summarize the numbers of the hardware resources required to synthesize each process. For example, in Fig. 4 (b), process E needs four 10-bit adders, one 13-bit incrementor, and twenty 20-bit subtractors if it is synthesized separately.

Using ILP package XPRESS-MP, it took 0.9 sec. and 0.4 sec. on SUN SparcStation10 for computing the minimum solution of SFIL and DINT, respectively. The numbers of the variables and inequalities are 34 (vars.) and 53 (inequals.) for

```
{
  par {
    A();
    B();
  }
  par {
    C();
    D();
  }
  par {
    E();
    F();
  }
  G();
}
```

(a) Parallel-serial structure.

	add				dec	mult	
	4	9	14	24	14	14	24
A	3		2		4	1	
B		1				1	
C			2	1		1	1
D	1	1					
E						1	
F							1
G		2					

(b) Hardware cost information.

Figure 5: Features of DINT.

SFIL, and 28 (vars.) and 41 (inequals.) for DINT.

Since the number of the processes in the examples were small, we synthesized gate level circuits for all the possible combination of thread partitioning and evaluate the effectiveness of our method. Table 1 and 2 summarizes the result. The column "thread partitioning" lists all the possible combinations of processes for partitioning to two threads. In both examples, none of the thread violated the complexity constraint of $comp_{max} = 200$. The partition "c1" is obtained by the naive method in the initial version of Bach compiler. Underlined partitioning c3 of SFIL and c4 of DINT are the optimum solutions obtained by ILP. The second column shows the value of objective function $cost(T)$. "Gate count" is the size of the resulting circuits, which is further categorized for closer analysis: [Op] functional units, [Mux] multiplexers, [Reg] registers, [FSM] finite control logic, and others. The percentage and standard deviation of each part is shown in the bottom two rows. The last column "ratio" shows the percentage of the total gate count against the original partitioning c1.

Comparing c3 of SFIL and c4 of DINT with c1, we see the circuit sizes are reduced by 3.8% in SFIL and 14.7% in DINT as compared with the naive partitioning. The reduction rate against the average are 2.5% and 7.3%, respectively. We can conclude that improved partitioning has its effect on hardware cost reduction.

We can observe that the values of the objective function do not exactly match the gate counts of the functional units but they have good correlation. So the objective function works as a good estimation of the final circuit size. We can also see that the functional units occupy about one third of the total gate count. Although registers and multiplexers have compatible or larger costs, the standard deviation of the gate count of the functional units is still larger than those of the other parts. These explains the reasons why the thread partitioning based on the estimated resource sharing worked well.

VI. Conclusion

We have presented a method of optimal thread partitioning method focusing on resource sharing among processes.

The detailed examination of the experimental result suggests that there is a possibility that deviation of the hardware cost of registers and multiplexers may have as much impact than that of functional units depending on systems. Researches will continue on the investigation of larger examples and development of efficient ways of incorporating the costs of registers and multiplexers as well as functional units.

Acknowledgments

The authors would like to thank Prof. Isao Shirakawa of Osaka University for his support and valuable comments on this research. We would also like to thank Dr. Takao Onoye, Mr. Gen Fujita, and Mr. Tatsuo Watanabe for their discussion and comments.

References

[1] D. Gajski, A. Wu, N. Dutt and S. Lin, "High-level Synthesis: introduction to Chip and System Design: " Kluwer Academic Publishers, 1992.

[2] "Behavioral Compiler User Guide" version 1997.01, Synopsys, 1997.

[3] R. Sakurai, M. Takahashi, A. Kay, A. Yamada, T. Fujimoto and T. Kambe: "A Scheduling Method for Synchronous Communication in the Bach Hardware Compiler," ASP-DAC '99, 1999.

[4] C. A. R. Hoare: "Communicating Sequential Processes," Prentice-Hall, London, 1985.

[5] M. R. Garey, D. S. Johnson: Computers and Intractability—A guide to the Theory of NP-Completeness—, W. H. Freeman and Company, New York (ISBN0-7167-1045-5).

Table 1: Experimental result on SFIL.

Thread partitioning	Obj. func. $cost(T)$	Gate count						ratio
		OP	Mux	Reg	FSM	other	total	
c1: {A,B,D,F,H},{C,E,G}	14076	16795.7	7651.3	17519.3	690.0	2705.7	45362.0	100.0%
c2: {A,B,D,F},{C,E,G,H}	14076	16817.3	7720.0	17714.3	735.3	2709.7	45696.7	100.7%
c3: {A,B,D,G,H},{C,E,F}	11040	14246.3	8159.0	17742.0	766.0	2708.3	43621.7	96.2%
c4: {A,B,D,G},{C,E,F,H}	11040	14246.3	8159.0	17742.0	766.0	2708.3	43621.7	96.2%
c5: {A,B,E,F,H},{C,D,G}	11040	13855.7	8619.3	17502.0	690.7	2706.7	43374.3	95.6%
c6: {A,B,E,F},{C,D,G,H}	11040	13853.0	8620.3	17572.0	728.3	2706.7	43480.3	95.9%
c7: {A,B,E,G,H},{C,D,F}	13872	21661.7	7892.0	17540.0	695.3	2709.7	50498.7	111.3%
c8: {A,B,E,G},{C,D,F,H}	13872	16475.0	7908.7	17545.3	714.0	2705.7	45348.7	100.0%
c9: {A,C,D,F,H},{B,E,G}	13872	16503.0	7818.3	17548.0	725.7	2706.3	45301.3	99.9%
c10: {A,C,D,F},{B,E,G,H}	13872	16446.3	7992.0	17681.0	716.3	2706.3	45542.0	100.4%
c11: {A,C,D,G,H},{B,E,F}	11040	13908.3	8575.7	17693.0	725.3	2706.7	43609.0	96.1%
c12: {A,C,D,G},{B,E,F,H}	11040	13908.3	8575.7	17693.0	725.3	2706.7	43609.0	96.1%
c13: {A,C,E,F,H},{B,D,G}	11040	14153.7	8029.7	17541.7	745.7	2709.0	43179.7	95.2%
c14: {A,C,E,F},{B,D,G,H}	11040	14239.7	7908.7	17443.3	736.0	2708.0	43035.7	94.9%
c15: {A,C,E,G,H},{B,D,F}	14076	16810.0	7511.7	17651.3	720.7	2707.0	45409.7	100.1%
c16: {A,C,E,G},{B,D,F,H}	14076	16830.3	7566.7	17632.7	719.3	2708.0	45457.0	100.2%
ratio of average		35.0%	18.0%	39.3%	1.6%	6.0%	100%	
standard deviation		2070.3	377.6	94.3	22.4	1.3	1836.7	

Table 2: Experimental result on DINT.

Thread partitioning	Obj. func. $cost(T)$	Gate count						ratio
		OP	Mux	Reg	FSM	other	total	
c1: {A,C,E,G},{B,D,F}	2944	2824.0	1815.0	3239.7	311.7	118.7	8309.0	100.0%
c2: {A,C,E},{B,D,F,G}	2908	2299.3	1601.3	3219.3	299.0	128.7	7547.7	90.8%
c3: {A,C,F,G},{B,D,E}	2235	1818.0	1620.7	3167.7	308.7	145.0	7054.0	84.9%
c4: {A,C,F}.{B,D,E,G}	2199	1829.0	1573.0	3238.7	303.0	145.7	7089.3	85.3%
c5: {A,D,E,G},{B,C,F}	2331	2157.7	1616.7	3544.7	301.3	95.7	7716.0	92.9%
c6: {A,D,E},{B,C,F,G}	2331	1861.7	1617.0	3551.3	300.0	80.7	7410.7	89.2%
c7: {A,D,F,G},{B,C,E}	3040	2195.7	1625.7	3551.3	300.0	103.3	7700.0	92.7%
c8: {A,D,F},{B,C,E,G}	3040	2395.3	1637.7	3576.3	299.7	99.3	8008.3	96.4%
ratio of average		28.6%	21.5%	44.4%	4.0%	1.5%	100.0%	
standard deviation		345.5	73.9	175.1	4.7	23.8	429.3	