Efficient Hardware Controller Synthesis for Synchronous Dataflow Graph in System Level Design

Hyunuk Jung, Kangnyoung Lee, and Soonhoi Ha

***Abstract*—This paper concerns automatic hardware synthesis from data flow graph (DFG) specification in system level design. In the presented design methodology, each node of a data flow graph represents a hardware library module that contains a synthesizable VHDL code. Our proposed technique automatically synthesizes a clever control structure, cascaded counter controller, that supports asynchronous interaction with outside modules while efficiently implementing the synchronous dataflow semantics of the graph at the same time. Through comparison with previous works with some examples, the novelty of the proposed technique is demonstrated.**

***Index Terms*—Data flow graph (DFG), synchronous data flow (SDF), system level design, VHDL.**

I. Introduction

THOUGH hardware description languages (HDLs), such as VHDL and Verilog, have gained considerable popularity in system design, there is also increasing need for easier and more intuitive method for system specification. Moreover, high level system design requires specification method of higher level of abstraction than HDLs which are biased to hardware model. Among many candidates, data flow graph (DFG) is adopted in many high level design frameworks [1], [2], specially for signal processing applications, because of formality and readability.

In our design methodology, functionality of a system is specified with a hierarchical data flow graph [Fig. 1(a)]. An atomic node represents a computation block, such as multiplier or filter, and an arc represents the flow of data samples between two end nodes. We use a rather restricted data flow model in this paper, synchronous data flow (SDF) [4], in which the number of data samples produced or consumed on an arc is fixed *a priori*. This restricted semantics enables us to verify important system properties such as memory boundedness and termination, and to estimate the system performance statically.

After the functionality of the system is verified by static analysis and behavioral level simulation, the nodes are partitioned into several graphs to be synthesized into hardware or software modules. VLSI design for a hardware module begins with a partitioned subgraph as shown in Fig. 1(b). For hardware synthesis, we generate a structural VHDL code which will be finally synthesized to hardware through any standard logic synthesis tool. Since we assume that a node describes its functionality with a synthesizable VHDL code in the library, the hardware synthesis problem is to stitch the library VHDL codes into a whole by automatically synthesizing interface codes and control codes for register initialization, appropriate clocking and signaling, and other glue logics.

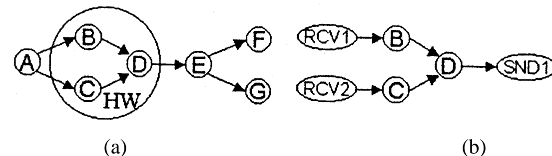

Fig. 1. (a) A DFG before partitioning. (b) A partitioned DFG for hardware module using send and receive nodes.

When synthesizing a hardware module from a partitioned subgraph, there are two issues to be addressed. First, the SDF semantics should be preserved in the generated code so that the resulting hardware has the behavior intended by the original DFG. This achieves the useful design concept of "`correct by construction`." Second, the partitioned DFG needs special nodes for communication with other partitioned graph. Two receive nodes and one send node are added in the partitioned subgraph in Fig. 1(b) for this purpose. While there are some previous works that addressed the former issue [3], [5], [7], [8], there has been no research result known to us except Lauwereins' work (GRAPE) [6], [7] that addressed both issues at the same time.

This paper proposes a novel technique to automatically synthesize a clever control structure, cascaded counter controller, that supports asynchronous interaction with outside modules while efficiently implementing the dataflow semantics of the graph at the same time. Through comparison with Lauwereins' distributed approach with some examples including DES cryptographic algorithm, the novelty of the proposed technique will be demonstrated.

II. Related Works

There are two possible approaches for system level design from a DFG specification. One is the distributed approach and

Manuscript received October 20, 2000; revised August 16, 2001. This work was supported by the National Research Laboratory (NRL) Grant and the Brain Korea 21 Project. The RIACT at Seoul National University provides research facilities for this study. A preliminary version of this work was presented as a regular paper in Proc. Int. Symposium on System Synthesis, pp. 79-84 Sept. 2000.

The authors are with the School of Electrical Engineering and Computer Science, Seoul National University, Seoul, 151-744, Korea (e-mail: {jung; knlee; sha}@iris.snu.ac.kr).

Digital Object Identifier 10.1109/TVLSI.2002.807765

1063–8210/02$17.00 © 2002 IEEE

the other is the centralized approach. These approaches are classified by how to implement the control logic to satisfy the semantics of DFG.

In the distributed approach, control logic is distributed and each node is executed by the local control logic to implement handshaking protocol between node pairs for determining the completion time of the source node. GRAPE [6] and [7] is a well-known tool of this approach. There is a buffer component between a pair of task nodes and handshaking protocol is implemented between the buffer component and each task node. The handshaking mechanism is also a natural method to communicate with the outside.

The centralized approach is used in Ptolemy [2] and [3] and Meyr's work [5] and [8], where there is no control signal exchange between task nodes. Through static analysis, the start time and the completion time of each node is predetermined with respect to the global clock, and the central controller generates the appropriate control signals. However, they assume that the execution time, at least the worst-case execution time, of each node is known *a priori*; without this assumption, the fully-synchronous design is not possible. Moreover, the interface with the outside has not been addressed except the batch communication scheme of [8].

Since the distributed approach has larger area overhead of local controllers than the central controller of the centralized approach, we follow the centralized approach in this paper. We propose a technique to overcome the limitations of the previous works, allowing both nodes of unknown execution time and asynchronous interfaces of unknown arrival time with the outside.

III. Central Controller

In this section, we identify the control signals that the central controller should generate, and the desired behavior of the controller. There are four types of library blocks: combinational logic, single-cycle sequential logic, multicycle sequential logic with fixed execution time, and multicycle sequential logic with variable execution time.

Depending on the block type, we predefine the minimal set of control signals for proper operation. A block of combinational logic needs no control signal. For a single-cycle sequential logic block, we extract the state element from the body to make it a Mealy-type state machine. Then, the *state-update* signal as well as clock and reset signals are needed for this state element. Multicycle sequential logic blocks contain state elements inside. Therefore, we need to provide clock and reset signals to the block. Moreover, a *start* signal enters into the block to indicate the start time. If a block has unknown execution time, the block should inform the controller the end of execution by providing *done* signal.

Fig. 2(a) shows a simple example that consists of all types of blocks. We integrate the library blocks and the synthesized central controller into a synchronous hardware as shown in Fig. 2(b). Here, all blocks are connected directly unlike in case of distributed approach that needs glue logic. In Fig. 2(c) the expected timing of control signals are drawn. Here, we assume that execution times of nodes A, B, and C are 30, 20,

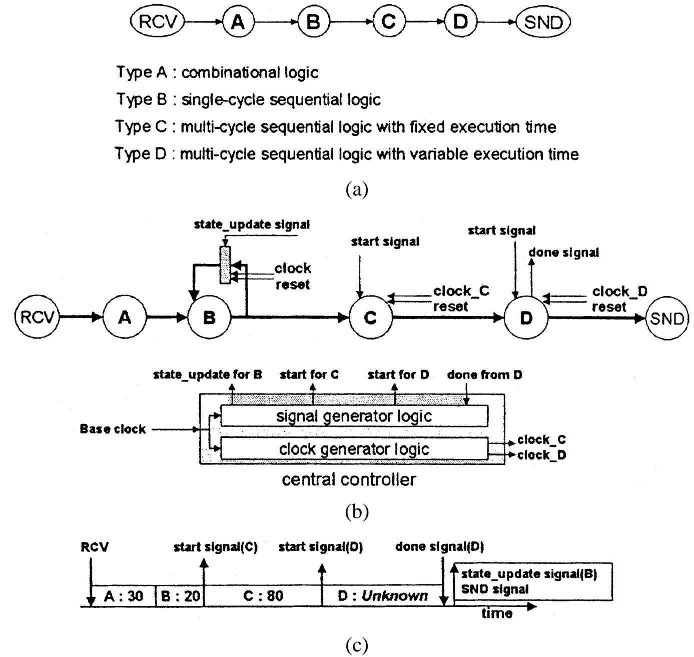

Fig. 2. (a) An example DFG with various types of nodes. (b) Synthesized hardware structure and (c) expected signal timings.

80 time units, respectively, while the execution time of node D is unknown. Though the hardware is fully synchronous, the timing of *done* signal for block D should vary at run-time: how to achieve this will be explained later.

One important design parameter is the global clock. All clock signals of Fig. 2(c) should be multiples of the base clock period. The optimal base clock period can be found using the greatest common divisor (GCD) of the minimum periods of all clock signals [9]. However, the GCD is sometimes too small to be practically implemented. And faster global clock also makes the control hardware more complex. Thus, there is a tradeoff the performance and the controller area with respect to the global clock selection.

IV. Interface Problem With the Outside

The interface nodes with the outside can be implemented in various ways according to the interface architecture or the communication scheme between modules. When synchronous communication or batch communication is chosen in the system architecture, the timing of reading the received samples are predetermined. Then, the interface logic becomes trivial and the whole hardware can be synthesized as a synchronous logic. But we may not always use such communication schemes. For example, batch communication scheme cannot be applied to the DFGs which have global feedback arcs. If synchronous communication scheme is used, tasks should be scheduled assuming the worst case execution time resulting in large overhead.

On the other hand, an asynchronous communication scheme can be used when the receive nodes do not know when data samples arrive from the outside. At the receiver side, the incoming samples are delivered by the synchronizer to the inside, and the hardware module is signaled somehow and starts processing. After data processing is completed, the result sample is placed

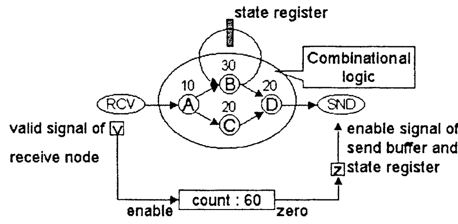

Fig. 3. Simple example using a counter. Node execution time of A, B, C, and D are 10, 30, 20, and 20 time units, respectively.

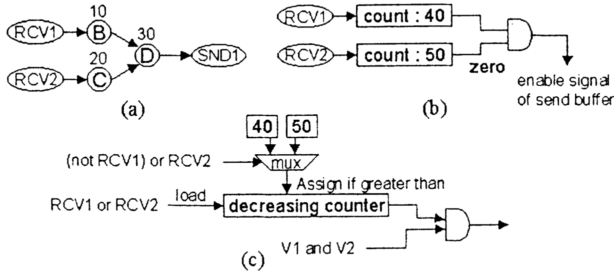

Fig. 4. Solutions for multiple receive nodes.

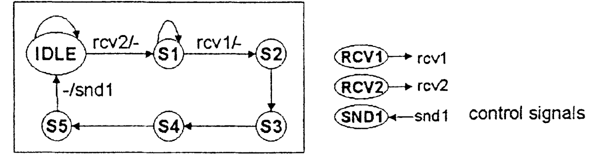

Fig. 5. FSM controller for multiple receive nodes.

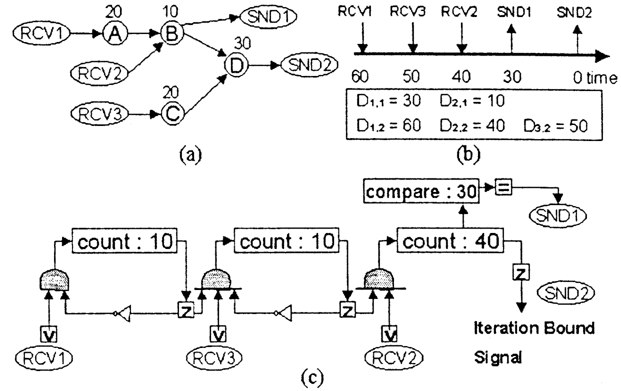

Fig. 6. A DFG that has receive nodes and send nodes and implemented cascaded counter controller.

to a certain location from which the sending operation is performed. In this paper, we are seeking for a hardware controller that can be implemented with small area overhead but keeping the performance requirements based on asynchronous communication scheme.

The key idea is to utilize the properties of the synchronous data flow model where the execution sequence of the nodes is predetermined at compile-time. If the execution time of a block varies at run time over a certain duration, the block is regarded as an asynchronous block that is addressed in Section V-D. For simple explanation of the proposed idea, we assume that input samples do arrive at the hardware module in the nonpipelined fashion. Also, the hardware module expresses the state registers explicitly as illustrated in Fig. 3. Since we do not need to pipeline the hardware module further, all state registers can be latched at the end of the execution.

The value of counter is initially set to the critical path length, 60, of the hardware graph ignoring the execution times of RCV and SND nodes. When the RCV node receives data, counter starts decreasing. When the counter value becomes zero, the output of node D is written to the buffer of the SND node (send buffer) and latch the state registers if any. The counter is reloaded with the initial value. Thus the simple counter accomplishes a controller that is triggered by the hardware valid flag written by the RCV node.

Now consider a DFG with multiple receive nodes as displayed in Fig. 4(a). If we use a single counter, the counter should be enabled only when two input samples both arrive at the interface. Thus, the hardware module performs a strict execution and may pay significant performance penalty. In order not to lose performance, we compute the pair-wise critical path lengths between all receive nodes and the send nodes. The following equation suggests the valid timing of the jth send node

$$VT_j = \max_i (RT_i + D_{i,j}) \tag{1}$$

where the maximum is taken over all receive nodes. VT_j denotes the valid timing of the jth send node, RT_i the receive timing of the ith receive node and $D_{i,j}$ is the critical path length from the ith receive node to the jth send node.

There are several approaches to implement the control logic that satisfies the above equations. The first approach simply uses counters as many as the number of receive-send pairs and ANDs the completion times of all counters destined for each SND node [Fig. 4(b)]. This implementation requires too many counters. However, we can reduce the number of counters using other glue logic like Fig. 4(c) or cascaded counter structure which is presented in the next section.

Another approach is to synthesize the FSM that satisfies the above equation. The FSM should wait the receive signal at scheduled receive timing. Fig. 5 shows the constructed FSM controller for the example of Fig. 4 assuming that the base clock has period 10. In S1 state, the FSM checks if RCV1 has received a data sample, and waits until a data sample arrives at RCV1. In the last state, S5, the FSM generates the enable signal for the send block.

V. Cascaded Counter Controller

A. Basic Idea

Consider an example of Fig. 6(a), which has three receive nodes and two send nodes. Assuming that node execution times of A, B, C, and D are 20, 10, 20, and 30 time units, respectively, critical path lengths from each receive node to the second send node (SND2), $D_{1,2}$, $D_{2,2}$, and $D_{3,2}$ are 60, 40 and 50 time units, respectively.

Intuitively, the ideal timing of send and receive nodes is like Fig. 6(b). This schedule can be obtained by ALAP (as late as

possible) scheduling from the instant of SND2. However, we may not assume that the RCV nodes do always receive data samples on time. Therefore, an efficient controller should figure out the earliest time for the readiness of output, regardless of the order in which the inputs arrive, assuming nonstrict execution of the graph. So we design a controller that has some *check* points while executing. If the check fails due to absence of input samples, the controller should stop executing and wait until the check succeeds. The proposed controller, called cascaded counter controller [Fig. 6(c)], satisfies this requirement.

In this controller, a single counter is split into many sections to form a cascaded counter array and the check points are made between the sections. Initially, the counter values are set to 10, 10, and 40, respectively, corresponding to the scheduled time intervals in Fig. 6(b). Only after the RCV1 node receives data, the first counter starts decreasing. Each counter starts decreasing after the previous counter is decreased to 0 and the current *valid* signals of RCV nodes are set to TRUE. At this point, the AND gate plays a role of check point logic. The *zero* signal not only enables the next counter but also disables the current counter. The last *zero* signal is the *iteration bound* signal that indicates the end of the current iteration and the start of the next iteration. This *iteration bound* signal becomes the enable signal for the last SND buffer, state register and delay register update signals, clear signal for valid and zero flags, and the counter initialization signal.

The complexity of the proposed cascaded counter controller is not much different from the simple counter structure shown in Fig. 3 except the flip-flops associated with the valid flags of receive nodes and AND gates. This cascaded counter controller can be also used for low-power design because at most one counter is operating at any instant.

In case the input DFG has a single send node as its destination node, the cascaded counter structure achieves the optimal performance. This fact is rephrased in the following theorem.

Theorem 1: For a DFG with a single send node as its destination node, the finish time of cascaded counting is equal to the valid timing of the send node of (1).

Proof: First, we simplify the valid timing equation with a single send node

$$VT = \max_i (RT_i + D_i). \tag{2}$$

We assume that D_i is sorted in the decreasing order such that $D_i \geq D_j$ when $i < j$ and the number of receive (source) nodes is n. Thus the value of each counter, d_i is

$$d_i = D_i - D_{i+1} (1 \leq i \leq n, \, D_{n+1} = 0).$$

We are now going to prove that VT is equal to FT, the finish time of the cascaded counter

$$P_1 = RT_1$$
$$P_i = \max (RT_i, \, P_{i-1} + d_{i-1}) (2 \leq i \leq n)$$
$$FT = P_n + d_n$$

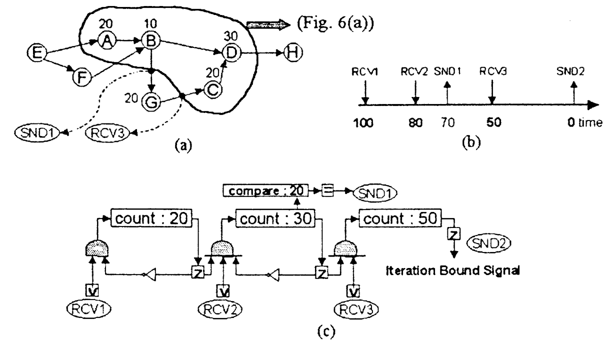

Fig. 7. Dependency between send and receive node. (a) Original graph before partitioning. (b) Scheduling to prevent deadlock. (c) Cascaded counter controller that implements the schedule (b).

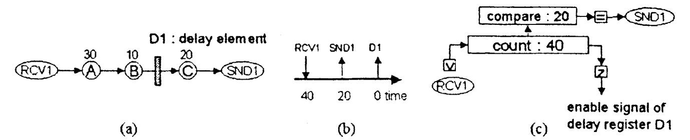

Fig. 8. A DFG having delay element(s).

where P_i is the start timing of the ith counter. By recursive substitution

$$FT = P_n + d_n = \max_i \left(RT_i + \sum_{k=i}^{n} d_k \right)$$
$$= \max_i (RT_i + D_i) = VT.$$

B. Dependency Between Send and Receive Nodes

With the dependency between send and receive nodes, deadlock may occur if such dependency is ignored and an overlapped cascaded counter controller is used. In case the RCV3 node is dependent on the SND1 node in Fig. 6(a), deadlock occurs obviously. The fact that a receive node is dependent on a send node means that the receive node cannot receive data until the send node sends its data. To understand this situation, we need to look into the original DFG before partitioning. Fig. 7(a) shows this original DFG before partitioning.

According to the DFG in Fig. 7(a), a partial execution order of nodes becomes B, SND1, G (in other module), RCV3, and C. To obtain the schedule result like Fig. 7(b), we have to add a pseudo dependency arc between SND1 and RCV3 node. Fig. 7(c) shows the implementation of cascaded counter controller according to the schedule of Fig. 7(b).

C. Delay Registers

In a DFG, delay elements may exist and they correspond to data registers in hardware implementation. If a DFG has a delay element, the delay register plays a role of a termination node. The graph is separated by this delay element in Fig. 8(a) and separated graphs are running concurrently in hardware implementation. Pipelined hardware implementation can be also obtained simply by adding delay elements. In case there is a node that contains pipeline registers, we first make those pipeline registers visible and apply the same method. Fig. 8(b) shows the concurrent schedule of the graphs in Fig. 8(a) and this schedule

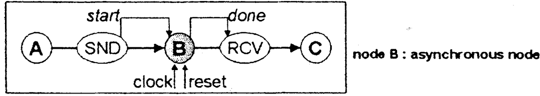

Fig. 9. Modified graph for internal asynchronous communication.

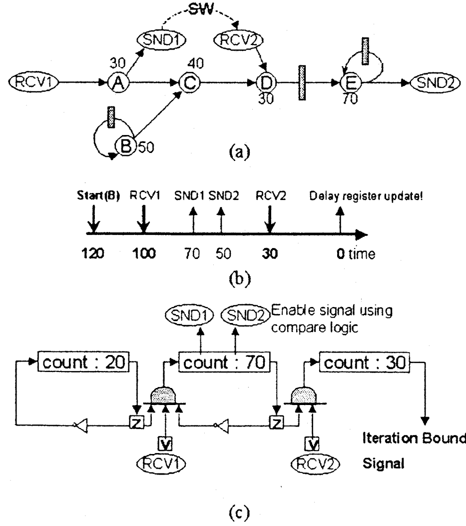

Fig. 10. A more complicated example.

can be implemented with only one cascaded counter controller depicted in Fig. 8(c).

D. Variable Execution Time

A hardware node may take nondeterministic time units for their executions. As mentioned in Section III, multicycle sequential logic blocks with variable execution time need *start* and *done* signal. In this case, we regard such a node as an asynchronous node and append send nodes before the input ports, and the receive nodes after the output ports. The *start* signal is generated from the send nodes and transferred to the asynchronous node and *done* signal is generated from the asynchronous node and transferred to the receive nodes. With this modified graph, we construct the cascaded counter controller. Fig. 9 shows the modified graph for internal asynchronous communication.

E. Cascaded Counter Construction Algorithm

We summarize how the corresponding cascaded counter controller is constructed in general using a complex example of Fig. 10(a).

The construction algorithm of the cascaded counter controller is summarized below.

Step 1) Find out all critical path lengths between destination nodes and source nodes. Destination nodes are the sink nodes, the send nodes, or delay elements from the partitioned graph. The critical path length is 120 time units from node B to the delay element next to node D.

Step 2) Compute ALAP (as late as possible) and ASAP (as soon as possible) schedule times of all nodes. And, assign the ALAP time for each receive node and the ASAP time for each send node as the expected start time of the node execution. Fig. 10(b) shows the schedule of send and receive nodes.

Step 3) Sort the execution times of destination nodes and source nodes. Construct the cascaded counter controller from this timing information. The firing time of RCV1 is scheduled at 100, while the cascaded counter controller begins with the critical path length of the graph, which is 120. Counters are split by the receive firing times 100 and 30. So the cascaded counter controller is split into three counters that load the initial values of 20, 70 and 30, respectively.

Step 4) Add the control logic for send nodes at their firing times. Since the firing times of the send nodes are 70 and 50, compare logic should be added at the output of the second counter to generate the enable signal to the send buffer. Finally the zero signal of the last counter is used for updating signal of delay registers. This signal notifies that one iteration ends and the next iteration is prepared.

VI. Optimal Clock Period Selection

In the previous section, the execution time of each node is specified in virtual time units for simplicity. The virtual time unit can be a clock period in the practical synthesis. Therefore, the actual execution time considering clock period can be described by

$$ET_c = \lceil PD/CP \rceil$$
$$ET_s = \lceil MCP/CP \rceil \times NumCycles$$

where ET_c and ET_s are the execution time of combinational and sequential logic. CP and PD represent clock period and propagation delay. MCP and $NumCycles$ are minimum clock period and required number of clocks of sequential logic. So the optimal clock period in this paper means the clock with the shortest DFG complete time.

The methods of selecting the optimal clock period for given DFG have studied in some works such as [9], [10]. All these methods compute a set of candidate of clock periods by taking the integral divisors of the execution time of each node. In [10], an idea that can reduce the size of the set was presented. This method can be applied to our scheduling algorithm in the cascaded counter construction by the repetitive scheduling for each candidate clock period.

VII. Experiments

We compare the area overhead between using our approach and the distributed approach such as used in the GRAPE tool. Since we used local counters inside the nodes and handshaking protocol between nodes in the distributed approach, we expect that hardware resources will be consumed more with the distributed approach. The latency of our approach is similar to that of distributed approach as expected in the optimality proof in Section V-A.

We experimented a DES encryption algorithm implementation. The target library is “xfpga_4000ex-4.db” of the commercial Xilinx FPGA and the clock periods we used are 5, 20, and

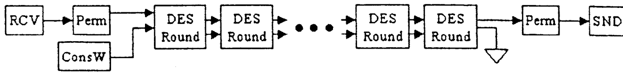

Fig. 11. Specified 16 round DES encryption algorithm. Internal details are not specified.

TABLE I
EXPERIMENT RESULT: DES

Approach		Distributed	Centralized	
			Cascaded counter	FSM
Area overhead (control logic + buffer)	50ns	(32+930) cell	3 cell	2 cell
	20ns	(32+930) cell	6 cell	10 cell
	5ns	(64+930) cell	7 cell	51 cell
Original area (computational logic)		3259 cell		

TABLE II
EXPERIMENT RESULT: FIG. 10

Approach		Distributed	Centralized	
			Cascaded counter	FSM
Overhead area (control logic + buffer)	50ns	(20+27) cell	12 cell	15 cell
	20ns	(24+27) cell	17 cell	24 cell
	5ns	(32+27) cell	21 cell	137 cell

50 ns. We make 16 rounds of the DES encryption algorithm (Fig. 11). The first round execution time is 26 ns and the other rounds have the execution time of 24 ns.

The specified DES algorithm is implemented in our codesign framework. Automatically generated codes contain the centralized cascaded counter. In the distributed approach, the interface between each node assumes two-phase handshaking. We also implemented a central controller using FSM. The experimental results are summarized in Table I. The area overhead in the distributed approach mainly consists of counters and registers to support handshaking protocol between each node.

We also experimented with the example of Fig. 10 assuming that one time unit is 10 ns. So the iteration period is 1200 ns. The result is summarized in Table II. As shown in Table II, the FSM implementation with 5 ns clock period has large overhead.

In the centralized approach using FSM, the area overhead grows more rapidly than using cascaded counter as the iteration period increases. It is because the FSM states are implemented using binary encoding.

As shown in Tables I and II, the whole system can be implemented efficiently by the centralized approach without handshaking between internal nodes. This means that we exploit the static schedule information at compile time as much as possible. At the same time, we use handshaking protocol to communicate asynchronously with the outside.

VIII. CONCLUSION

In this paper, we addressed how to synthesize the hardware control module from the initial DFG specification for hardware–software codesign. We proposed a novel control structure, cascaded counter controller, to reduce the hardware area or time considerably, compared with the previous approaches.

Two issues are addressed efficiently with the cascaded counter controller: the initial DFG semantics should be preserved and the performance loss should be avoided as much as possible. We experimented with some examples including DES encryption algorithm to demonstrate how the proposed technique truly builds the working VHDL code, which is verified with a commercial Xilinx tool.

REFERENCES

[1] Synopsis, Inc., "COSSAP User's Manual: VHDL code generation,", Mountain View, CA 94043.
[2] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems," *Int. J. Comput. Simulation*, vol. 4, pp. 155–182, Apr. 1994.
[3] M. C. Williamson and E. A. Lee, "Synthesis of parallel hardware implementations from synchronous dataflow graph specifications," in *Proc. 30th Asilomar Conf. Signals, System Computers.*, Pacific Grove, CA, Nov. 1996.
[4] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proc. IEEE*, Sept. 1987.
[5] P. Zepter, T. Groker, and H. Meyr, "Digital receiver design using VHDL generation from data flow graphs," in *Proc. Design Automation Conf.*, 1995.
[6] R. Lauwereins, M. Engels, M. Ade, and J. A. Peperstraete, "Grape-II: A system-level prototyping environment for DSP applications," *IEEE Comput.*, pp. 35–43, Feb. 1995.
[7] J. Dalcolmo, R. Lauwereins, and M. Ade, "Code generation of data dominated DSP applications for FPGA targets," in *Proc. IEEE Int. Workshop Rapid System Prototyping*, 1998, pp. 162–167.
[8] J. Horstmannshoff and H. Meyr, "Optimized system synthesis of complex RT level building blocks from multirate dataflow graphs," in *Proc. Int. Symp. System Synthesis*, Nov. 1999, pp. 38–43.
[9] E.-S. Chang, D. D. Gajski, and S. Naraya, "An optimal clock period selection method based on slack minimization criteria," *ACM Trans. Design Automation Electron. Syst.*, vol. 1, pp. 352–370, July 1996.
[10] S. A. Blythe and R. A. Walker, "Efficient optimal design space characterization methodologies," *ACM Trans. Design Automation Electron. Syst.*, vol. 5, pp. 322–336, July 2000.

Hyunuk Jung photograph and biography not available at the time of publication.

Kangnyoung Lee photograph and biography not available at the time of publication.

Soonhoi Ha photograph and biography not available at the time of publication.