A Technique for Synthesizing Distributed Burst-mode Circuits

Prabhakar Kudva[†]
IBM T.J. Watson Research Center
Yorktown Heights

Ganesh Gopalakrishnan[‡] Hans Jacobson
Department of Computer Science
University of Utah

Abstract

We offer a technique to partition a centralized control-flow graph to obtain distributed control in the context of asynchronous high-level synthesis. The technique targets Huffman-style asynchronous controllers that are customized to the problem. It solves the key problem of handling signals that are shared between the partitions—a problem due to the incompletely specified nature of asynchronous controllers. We report encouraging experimental results on realistic examples.

Introduction

Asynchronous circuits are receiving considerable attention of late due to their promise in many areas including performance and energy consumption. A central problem in asynchronous high-level synthesis is that of partitioning a centralized control-flow graph to obtain distributed controllers. A centralized controller can often be more complex (in terms of logic) than a collection of distributed controllers, and can have slower signal paths through it. They can also result in increased wire lengths, involve timing assumptions of a global nature. In this paper, we present the automated control partitioning algorithm incorporated in our high-level synthesis tool for asynchronous circuits called ACK which accepts a subset of high-level Petri-nets as input and generates partitioned two-phase controllers and the associated data-path as output. A Verilog front-end is also available for ACK.

One way to obtain distributed control circuit realizations is by employing macromodules [15]. However, most macromodule libraries contain only a limited number of macromodule types, and hence distributed control realizations based on macromodules are often inefficient [6]. A class of controllers called burst-mode controllers that are potentially more efficient than macromodules, and can be customized [4, 11, 17] have been proposed and widely used in a number of non-trivial designs. However, burst-mode synthesis procedures cannot handle designs beyond a certain input/output (I/O) size, due to the complexity of many of the global optimizations used. Hence, in previous designs where this I/O size was exceeded, burst-mode controllers were manually partitioned largely depending on the designer's intuitions. This procedure is very tedious and results in burst-mode controller descriptions that are incomprehensible and hard to verify. Moreover, even when the burst-mode synthesis of centralized controllers with large I/O sets is possible, the resulting controllers can be inefficient, as pointed out above. Therefore, in an automated high-level synthesis environment such as ACK, where large control graphs can be generated from users's high-level HDL description of non-trivial designs, an automated controller partitioning method is essential. This is the problem addressed in this paper.

A key problem in partitioning stems from the fact that asynchronous controllers are, in general, incompletely specified. More specifically, the steps of critical race free state assigment and hazard-free logic minimization in burst mode synthesis rely on the fact that the environment of the controller does not present any of the unspecified behaviors. Under this assumption, the sharing of signals between the partitions is a non-trivial problem. For example, suppose an input signal is shared between a collection of partitions. When the environment generates a change on this signal, to which of these partitions must the change be sent to? We provide a method to address this issue.

Related Work

In [10], a technique called *process decomposition* is proposed. Process decomposition does not involve signal-sharing between incompletely specified machines. Signal-sharing is addressed in macromodule based design systems [1, 2] by using additional macromodules such as Toggles [15] and Decision-waits [5] to steer the global input to the correct sub-controller. In [3, 14], a method called *contraction* has been suggested as a decomposition technique for signal transition graph (STG) specifications. Contraction preserves the global nature of the controller. It does not turn a large-grain controller into many smaller-grain controllers. Contraction and partitioning are orthogonal techniques, and both have been successfully integrated in ACK.

Overview of ACK

A design entered in ACK is a Petri-net description organized as a collection of sequential processes communicating through CSP-style channels. Each transition of the Petri-net (except fork and join) is annotated with an *action*, which can be a two-phase [15] signal transition on an input or output wire (input transition names are underlined), an assignment statement, a Boolean expression (used for choices), or a CSP-style communication primitive. Fork-join concurrency is allowed within sequential threads. The fork and join transitions are labeled by an "ϵ" action denoting a no-op.

Synthesis in ACK proceeds by first allocating requisite data path resources, which include library/Viewlogic-synthesized operators for computation actions, C-elements for channel communication actions, select elements for data dependent choices, and library registers for storage. The underlying control-graph is then obtained by refining each high-level Petri-net action into two-phase handshake actions, using standard approaches [1, 16]. The end result is one centralized control graph per sequential process.

The partitioned synthesis problem addressed in this paper is: given a centralized control graph and a set of partitions on it chosen

[†] *This research was done when the first author was a graduate student at the University of Utah and was supported in part by University of Utah Research Fellowship.
[‡] Supported in part by NSF Award MIP 9215878

DAC 96 - 06/96 Las Vegas, NV, USA ©1996 ACM, Inc. 0-89791-833-9/96/0006..$3.50

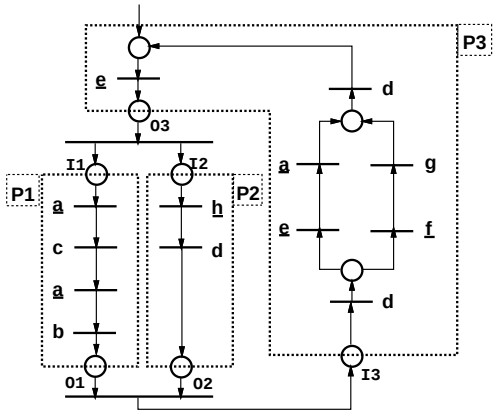

Figure 1: An Example Partitioned Petri-net

by the designer (as in Figure 1), how do we synthesize separate controllers for each of the partitions that correctly orchestrate control flow between the partitions and correctly handle signal sharings? The identification of the partitions is not addressed here, though a few automatable heuristics, such as keeping logically unrelated iterative loops that share signals in separate partitions, usually yield good results in terms of increased performance and reduced logic complexity.

Note that some places in the Petri-net in Figure 1 have been omitted due to paucity of space.

Partitioning Algorithm

Centralized control graphs which form the input to the partitioning phase of ACK are "state machines [13] with fork/joins" (SFJ), that is, a single threaded graph with fork/join concurrency. SFJ graphs are triples $C = (P, T, F)$ where $P = \{p_1, p_2 \ldots, p_n\}$ is a finite set of places, $T = \{t_1, t_2 \ldots, t_m\}$ is a finite set of transitions, and $F \subseteq (P \times T) \cup (T \times P)$ is a flow relation. T consists of fork transitions (T_f), join transitions (T_j), and sequential transitions (T_s) which have an in-degree and out-degree of one. For each $t \in T_f$, there is exactly one $t' \in T_j$ (and vice versa) such that the out-degree of t, N, is the same as the in-degree of t'. The ith output place of fork transition t and the ith input place of corresponding join transition t' are, respectively, the (only) input place and (only) output place of a *single threaded subgraph* (STS) that models "the i-th thread" of the fork/join. An STS, $C_{st} = (P_{st}, T_{st}, F_{st})$ is a subgraph of C where $P_{st} \subseteq P$, $T_{st} \subseteq T_s$, and $F_{st} \subseteq F$ is the flow-relation restricted to P_{st} and T_{st}. An STS must be disjoint from all other STSs (not share any place or transition). T_{st} should not contain a fork or a join (but may contain choices). The unique entry-place and exit-place of an STS are called its *input place* and *output place*, respectively.

Each transition in T_s is annotated by a non-empty burst of two-phase input- or output- (but not both) signal transitions. Transitions in T_f and T_j are labeled by ϵ. To simplify things, we assume that no two bursts labeling transitions contained in two different STS graphs of the same fork/join involve the same wire name. Also, in order to generate legal burst-mode machines [11] *from the partitions* through burst-mode reduction [6], the original SFJ graphs must obey the following restrictions, in that they are (1) initially quiescent, and attain quiescence infinitely often (a quiescent state is one where no output must be produced before consuming at least one input); (2) deterministic, (3) delay insensitive, and (4) obey the subset property [11].

At the end of partitioning, the goal is to generate sequential machines where each transition is annotated with input and output bursts. It has been shown [6] that such a partitioned machine P can be converted into a burst-mode machine C that has, as its interface traces, the set of traces generated by P when operated in the fundamental mode (P is allowed to attain quiescence after each set of inputs to it). Thus, the real proof obligation of our partitioning procedure (to be described) is to ensure that the interface traces for a collection of burst-mode controllers implementing the partitions of a well-formed SFJ graph are the same as that for a centralized burst-mode controller implementing the same SFJ graph.

A *partition* of an SFJ graph is either any of the STS subgraphs of a fork/join (these are called *required partitions*) or one of the STSs obtained by the following procedure applied to each of the fork/join pairs, (t, t'): (1) remove the fork/join pair and all the STSs subtended by them; (2) assign the input place of t as the output place of the STS preceding t; (3) assign the output place of t' as the input place of the STS following t'. Any of the STSs obtained above can be further partitioned into its constituent STSs. The set of input- and output-places of the partitions of the given SFJ are called *partitioning places* (PP). In Figure 1, I_k and O_k (for $k \in 1 \ldots 3$) are the input- and output places of the three partitions, and form the partitioning places. Additional partitioning places may be chosen from within the partition P3 (though we don't do so in our example). We do not consider STSs with an empty set of transitions as partitions.

Partitioned Controller Synthesis

Each partition is supported by its own controller initialized to its own initial state. The partition controllers also incorporate provisions to hand-over control to other partition controllers. We simplify our initial exposition by: (1) considering all PPs to be either the input(s) or output(s) of forks and joins; (2) assuming that signals are not shared between partitions. *These assumptions will be relaxed momentarily.*

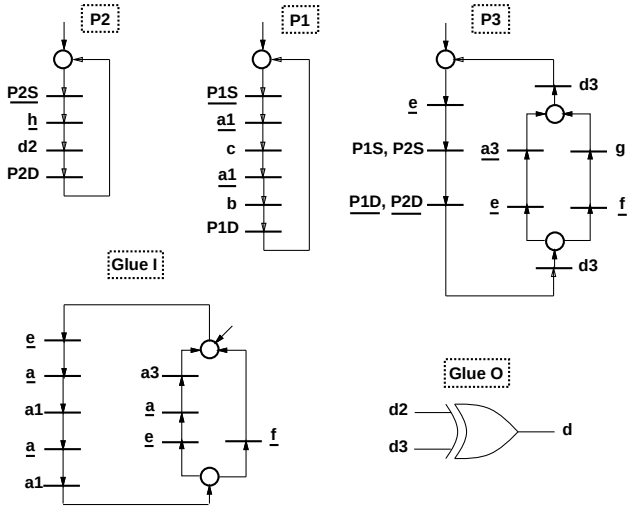

Figure 2: Illustration of Decomposition, and Input Translators

Consider the controller C_i supporting the ith required partition of a fork/join. Proper control hand-over between C_i and the controller for the partition preceding the fork, C^-, is arranged by making the very first transition processed by C_i from its start state to be $\{\underline{\text{done}}\}$, where done is the last signal transition generated by C^- before *it* goes back to its start state where it in turn waits for a $\{\underline{\text{done}}\}$ signal from another partition, telling it to resume execution. This ensures that whenever controller C^- finishes its execution, the C_is are all started. The remaining actions of C_i are the same as the actions present in the STS graph of the ith required partition. Proper control hand-over between C_i and the controller for the partition following the join, C^+, is arranged by making the very last transition processed by C_i before it goes back to its start state to be $\{\text{done}_i\}$. The very first transition processed by C^+ out of its start state, then, is $\{\text{done}_i\}$, $i \in 1 \ldots K$, where K is the arity of the fork/join. This ensures that whenever all controllers C_i finish, C^+ takes over. Note that C^- and C^+ may be the same partition.

Now, relax assumption (1) above and consider PPs that are not associated with fork/joins. Every such PP demarcates two partitions with their own supporting controllers starting in their own initial states. The PP serves as a "merge" place for the threads preceding it and as a "choice" place for the threads following it. Observe that these threads are sequential with respect to eachother. Consider the modifications that must be made to the controller that supports the partition preceding PP. For this partition we add the output burst $\{\text{done}\}$ to be the last action of this controller before it goes back to its start state. Similarly, consider the controller supporting the partition following PP; we add the input burst $\{\underline{\text{done}}\}$ as the very first action of this controller from its start state. This ensures that whichever way the merge-place is entered, the choice-place is enabled. The controller then proceeds to carry out the remaining actions of the partition following the PP. Figure 2 illustrates these ideas.

Such a distributed control realization of an SFJ graph (as described above) manifests exactly the same interface traces as a centralized controller realization of the same SFJ graph when the distributed control realization is operated in the fundamental mode. This is because whenever control is handed over through the "done" signals, the done signal generated by preceding partition(s) are (all) absorbed by the following partition(s) *before the environment is allowed to send any new inputs to the following partition*. Thus, as far as the environment is concerned, the right set of partitions become active at the right times. The rest of this paper concerns itself with relaxing assumption (2) above.

Synthesize Sharing Arrangements

Define the input set $Inp(K) \subseteq W$ of a partition K to be a set of input wires which K is sensitive to, meaning these inputs make a transition somewhere within partition K. Define $Out(K)$ similarly. If a partition is sensitive to a set of input and output wires disjoint from that of all other partitions, it can be directly implemented. For output wire o that is shared between partitions K_1 and K_2, we rename o to o_1 in K_1 and to o_2 in K_2, and synthesize the resulting controllers using our method. The outputs o_1 and o_2 are then merged using an XOR-gate to produce the output signal o. In Figure 2, output d is generated in this manner. This method works because of the two-phase nature of the control signals, and because any two occurrences of an output signal (the two input signals of the XOR) transition are guaranteed to be *sequentially* ordered.

The difficulty in handling input sharing is that we must ensure that only the "right partition" must see the input transition in each state. In Figure 2, the first and the second occurrences of input a are seen by partition P1 while the third occurrence must be seen by partition P3 *if* the choice is resolved through e, and by partition P1 if the choice is resolved through f. As with shared outputs, the first step is to rename the shared inputs within each partition. An *input-translator state machine* is then derived that translates the input signal from the environment into these renamed signals, local to the component machines, at the right times. The controller GlueI in Figure 2 achieves this for signal a, in our example.

For the algorithm to generate input translators, we assume that any input signal that resolves a choice (appears in the burst labeling the transition immediately following a choice place) occurs in no other partition than the partition that contains the choice. A solution that relaxes this assumption exist [8] and proof is in progress. The steps in obtaining input translators are as follows: (1) Create an input set for the input translator. This input set consists of the shared input signal of interest and all the choice signals. Remove all signals except members of the input set from the original machine. (*e.g.*, erase b, c, d and g from Figure 1). (2) For a collection of threads subtended by a fork/join pair, retain only that thread, if any, in which the input signal of interest ($\underline{\text{a}}$ in our case) appears. In our case, we retain partition P1. (3) In the resulting graph, following each occurrence of the transition of the shared signal i, introduce a corresponding output o_k. In our example, following the two occurrences of inputs $\underline{\text{a}}$ falling in partition P1, we generate output a1 whereas following the occurrence of input $\underline{\text{a}}$ in P3, we generate output a3. These, then, are internal signal transitions that get sent to the right partition at the right time.

Synthesize Final Circuits

Each of the controller descriptions can now be synthesized into asynchronous burst-mode circuits following the procedure described in [6]. In order to obtain a burst mode controller specification from a two-phase controller specification, we need to know the initial input signal values for each of the controllers. The initial values of all external input signals are specified by the user. All the internal signals that are introduced during partitioning can be initialized to any value due to use of two phase protocol (we initialize them to 0). The resulting description can be synthesized into a burst-mode machine which can then be synthesized using (*e.g.*) 3D tool [17] or ASSASSIN [9].

Results and Conclusions

We have conducted comparisons between centralized and partitioned controllers on a large number of examples, some of which are shown in Table 1. Apart from making it possible to synthesize larger designs, partitioning can also decrease synthesis time by several orders of magnitude. Partitioning also often significantly decreases the number of literals in the synthesized design and increases the overall controller performance compared to that of a centralized implementation.

In Table 1 we show the partitioning results for a CD Player Error Detector from [7], a barcode reader from the High Level Synthesis Design benchmarks [12] adapted to asynchronous operation, a loop example, a factorial computation unit, and an iterative implementation of the GCD algorithm. For the CD Player Error Detector and the Barcode Reader the synthesis of the centralized controllers did not complete due to the complexity of the synthesis task. The results for these are marked with $n.a.$ In the table the $\#BM\ trans$ column is a measure of controller complexity and shows the number of burst mode transitions in the specification of the controller. $IO\ size$ shows the size of the input and output set of the controller, $Synth\ time$ shows the time in seconds for burst mode synthesis and $\#\ of\ literals$ stands for the number of

literals in the implementation of the controller. In the $Controller$ column $Centralized$ means the centralized contoller, $Part\ x$ means partition number x and $ISM\ x$ means Input Translator State Machine number x.

For the examples where the centralized controllers finished synthesis, a layout was generated from a two level standard gate implementation and the performance between the centralized and partitioned controllers was measured. The comparison showed an average performance increase for the partitioned controllers of between 10 to 30%. Note that this comparison only exploited performance advantages due to temporal locality. Partitioning also gives us the possibility to take advantage of spatial locality, which as feature sizes gets smaller and wire delays become significant, is an important factor for high performance designs.

In this paper, we have presented a method to deal with the partitioning of asynchronous controllers. This work specifically provides a partitioning method in the context of asynchronous high level synthesis methods that target state machine controllers, although the basic ideas can be extended to other asynchronous partitioning problems.

Controller	# BM trans	IO size	Synth time	# of literals
CD Error Det.				
Centralized	1824	68	n.a.	n.a.
Part 1	110	18	800	94
Part 2	32	29	220	96
Part 3	28	25	140	122
ISM 1-3	81	13	230	63
ISM 4	16	6	90	14
ISM 5	7	14	80	92
Barcode Reader				
Centralized	960	49	n.a.	n.a.
Part 1	26	14	34	14
Part 2	40	8	25	3
Part 3	72	19	500	13
Part 4	26	23	53	43
ISM 1-2	56	8	22	14
ISM 3	64	9	28	53
GCD				
Centralized	126	25	33420	207
Part 1	22	15	30	33
Part 2	72	18	340	12
ISM 1-2	48	7	25	14
Factorial				
Centralized	44	20	620	88
Part 1	12	13	24	6
Part 2	28	15	36	9
ISM 1-2	16	5	20	14
Loop example				
Centralized	32	5	38	199
Part 1	14	5	20	58
Part 2	12	6	16	3

Table 1: Results for partitioning

Acknowledgment: The authors would like to thank Al Davis for many helpful discussions that brought our attention to the problem.

References

[1] AKELLA, V., AND GOPALAKRISHNAN, G. SHILPA: A high-level synthesis system for self-timed circuits. In *Proc. International Conf. Computer-Aided Design (ICCAD)* (Nov. 1992), IEEE Computer Society Press, pp. 587–591.

[2] BRUNVAND, E., AND SPROULL, R. F. Translating concurrent programs into delay-insensitive circuits. In *Proc. International Conf. Computer-Aided Design (ICCAD)* (Nov. 1989), IEEE Computer Society Press, pp. 262–265.

[3] CHU, T.-A. *Synthesis of Self-timed VLSI Circuits from Graph Theoretic Specifications*. PhD thesis, Department of EECS, Massachusetts Institute of Technology, Sept. 1987.

[4] COATES, B., DAVIS, A., AND STEVENS, K. The Post Office experience: Designing a large asynchronous chip. *Integration, the VLSI journal 15*, 3 (Oct. 1993), 341–366.

[5] EBERGEN, J. C. *Translating Programs into Delay-Insensitive Circuits*, vol. 56 of *CWI Tract*. Centre for Mathematics and Computer Science, 1989.

[6] GOPALAKRISHNAN, G. C., KUDVA, P., BRUNVAND, E. L., AND AKELLA, V. Peephole optimization of asynchronous macromodule networks. In *Proceedings of the International Conference on Computer Design (ICCD)* (1994), pp. 442–446.

[7] KESSELS, J., VAN BERKEL, K., BURGESS, R., RONCKEN, M., AND SCHALIJ, F. An error decoder for the compact disc player as an example of VLSI programming. Tech. rep., Philips Research Laboratories, Eindhoven, The Netherlands, 1993.

[8] KUDVA, P. N. *High Level Synthesis of Asynchronous Circuits Targeting Finite State Machines*. PhD thesis, Dept. of Computer Science, University of Utah, Salt Lake City, UT 84112, 1995.

[9] LIN, B., AND VERCAUTEREN, S. Synthesis of concurrent system interface modules with automatic protocol conversion generation. In *Proc. International Conf. Computer-Aided Design (ICCAD)* (Nov. 1994), pp. 101–108.

[10] MARTIN, A. J. Programming in VLSI: From communicating processes to delay-insensitive circuits. In UT Year of Programming Institute on Concurrent Programming (1989), e. C.A.R. Hoare, Ed., Addison-Wesley.

[11] NOWICK, S. M., AND DILL, D. L. Synthesis of asynchronous state machines using a local clock. In *Proc. International Conf. Computer Design (ICCD)* (Oct. 1991), IEEE Computer Society Press, pp. 192–197.

[12] PANDA, P. R., AND DUTT, N. 1995 High level synthesis design repository. Tech. Rep. 95-04, University of California, Irvine, Feb. 1995.

[13] PETERSON, J. L. *Petri Net Theory and The Modeling Of Systems*. Prentice-Hall, 1981.

[14] PURI, R., AND GU, J. A modular partitioning approach for asynchronous circuit synthesis. In *Proc. ACM/IEEE Design Automation Conference* (June 1994), pp. 63–69.

[15] SUTHERLAND, I. Micropipelines. *Communications of the ACM* (June 1989). *The 1988 ACM Turing Award Lecture*.

[16] VAN BERKEL, C., NIESSEN, C., M.REM, AND R.SAEIJS. Vlsi programming and silicon compilation: a novel approach from phillips research. In *Proceedings of IEEE International Conference on Computer Design (ICCD)* (1988).

[17] YUN, K. Y. *Synthesis of asynchronous controllers for heterogeneous systems*. PhD thesis, Stanford University, Aug. 1994.