Software Synthesis through Task Decomposition by Dependency Analysis

Youngsoo Shin Kiyoung Choi
School of Electrical Engineering
Seoul National University, Seoul, Korea, 151-742

Abstract

Latency tolerance is one of main problems of software synthesis in the design of hardware-software mixed systems. This paper presents a methodology for speeding up systems through latency tolerance which is obtained by decomposition of tasks and generation of an efficient scheduler. The task decomposition process focuses on the dependency analysis of system i/o operations. Scheduling of the decomposed tasks is performed in a mixed static and dynamic fashion. Experimental results show the significance of our approach.

I. Introduction

Recently, software synthesis and code generation process has become an important step in the cosynthesis of mixed hardware-software systems. This has been driven by advances in the performance of modern microprocessors and microcontrollers and increasingly large portion of software in embedded systems.

Most of the previous approaches to the development of software components in mixed systems use the execution model of mutual exclusion between hardware and software. A processor is in a "busy-wait" or "hold" state until the hardware components complete their computation and return the results. This strategy is simple and reasonable when hardware components complete their execution in a short time interval. However, when the execution delay is relatively long due to data-dependent loops or the environment, this type of execution model can cause very low processor utilization and long latency. This problem can be solved by the execution model of multi-thread of control which is achieved by code restructuring through task decomposition.

Manual decomposition and code restructuring are error-prone, hard-to-work, and difficult to preserve the semantics of the original system description. We took automatic task decomposition and code restructuring schemes in [5], where we assume that the hardware is capable of executing a single task at a time. In this paper, we extend the idea to a general case, where the hardware is composed of multiple functional resources and can execute multiple tasks at the same time. In summary, we make assumptions as follows.

- Hardware is composed of a set of functional resources which can be accessed concurrently.
- Hardware functional resources are already bound to software i/o operations.

When there are multiple i/o operations that access the same hardware resource and have no dependencies between them, access serialization should be guaranteed not to cause a resource contention. We decompose a task and generate a scheduler in such a way that the requirement is not violated.

This research was supported in part by the Institute of Information Technology Assessment under contract 95-X-5916.

The overall structure of our software synthesis process is as follows. First, the system specification is transformed to a control data flow graph(CDFG) model. Our target for the description of system specification is mixed VHDL and C under Ptolemy[6] environment. The current implementation supports cospecification and co-simulation in both VHDL and C with extended Ptolemy. For the generation of CDFG, however, only VHDL is supported currently. The CDFG is partitioned manually into two parts to be implemented in hardware and software. Then the interface between the two parts is generated and annotated to each partitioned CDFGs[2]. Automatic partitioning is currently under development and is beyond the scope of this paper. CDFG representing the software part consists of basic operations, control constructs, and system interface operations. It is partitioned into a set of segments which we call threads. The set of threads together with their scheduler form the synthesized software. The thread scheduling is mixture of static and dynamic scheduling but is maximally static in that all threads that can be given static ordering are scheduled statically. Threads that is to be scheduled dynamically are put on an ordered list. Dynamic scheduling is done by dispatching threads one by one from the ordered list while polling the synchronization signal from the hardware. The dynamic dispatch continues until the synchronization signal is received or there is no more thread ready to be scheduled.

The rest of this paper is structured as follows. The next section presents the motivation of our work. System model and problem formulation are discussed in Section III. Section IV describes the task decomposition and the generation of the thread scheduler. Experimental results are presented in Section V. We draw conclusions and present future work in Section VI.

II. Motivation

There are many approaches to software synthesis in hardware-software codesign environment. In [1][3], timing constraints are specified in the form of min/max constraints between operations or rate constraints of operations. In these approaches, software is structured as a set of threads which start from non-deterministic operations. In [4], timing constraints are specified as speed up factors and software part and hardware part function as master and slave mode, respectively. A processor which runs the software part is in hold state until the hardware part completes its execution. All these approaches do not use or limitedly use implicit parallelism between software and

1063-6757/96 $5.00 © 1996 IEEE

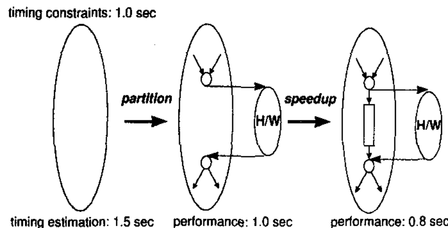

Fig. 1. Speedup of partitioned task by execution overlap.

hardware which is extensively exploited in our approach.

The goal of this paper is to optimally speed up the software part through latency tolerance achieved by task decomposition and restructuring. There exist many reasons why we must speed up the software. For example, in hard-real time systems where tasks are defined with their timing attributes/constraints such as periods, deadlines, execution times, assume there is no feasible schedule for a given task set. In such cases, there are many alternatives for satisfying schedulability such as code tuning, changing deadlines, and so on. Reduction of the execution time of tasks by implementing them in hardware gives another alternative.

Fig.1 shows an example of the proposed approach. Assume a task is to be completed within 0.8 sec but all software solution takes 1.5 sec. It can be partitioned toward hardware solution until given timing constraints are met. With a given partition that completes the task in 1.0 sec with mutually exclusive execution model, our approach can be applied to reduce the execution time further down to 0.8 sec.

III. System Models and Problem Formulation

We use a control data flow graph(CDFG) as an abstract model of the system specification. Our CDFG is defined as a graph $G(N, E)$ where N is a set of nodes $n_i^j (i = 1, ..., m, j = s, e, op, ct, w, r)$ and E is a set of directed edges between nodes. We distinguish four types of nodes. n_i^s and n_i^e are introduced as polar nodes to an entire graph, to a condition clause or predicate of each conditional statement, and to a conditional branch. All nodes and edges on the paths from n_i^s to n_i^e form a convex subgraph. n_i^{op} is an operational node. n_i^{ct} is a control node which represents conditional statement. n_i^r and n_i^w are nodes for system i/o operations. They represent any sequence of operations required to satisfy the selected communication protocol. So their granularity can vary according to the complexity of the communication protocol. These nodes are generated and annotated to software parts and hardware parts, respectively, according to the selected target architecture and communication protocol[2]. We define $HW(n_i^r)$ and $HW(n_i^w)$ as hardware resources accessed by n_i^w and n_i^r, respectively.

The software interface module combines device driver routine calls, I/O function calls, and load/store commands to read/write data from/to the system bus. The hardware interface module consists of signal registers and a protocol converter[2].

An edge $n_i > n_j$ denotes a dependency from node n_i to node n_j. When there is a transitive dependency relation between n_i and n_j, that is, there exists a *path* from n_i to n_j, we denote the relation as $n_i >_t n_j$. Dependency between nodes exists when there are data dependency or control dependency.

In this paper, we define a thread T_i as a subset of N that consists of a sequence of successively connected nodes and has the property that once the first node fires then it executes to the end without interruption. Dependency relations between threads arise from the relation between nodes in the threads. For example, when $n_i \in T_k$, $n_j \in T_l$, and $n_i > n_j$, we say that T_l depends on T_k and denote it as $T_k > T_l$. n_i^r is always a start node of a thread T because it is a non-deterministic operation as defined in [3]. We define $T(n_i^r)$ as a thread that starts from n_i^r.

The problem can be stated as two folds. First, partition the graph into threads and determine threads that can be executed in parallel with the hardware such that the parallelism is maximally exploited while preserving the semantics of the original system specification. Second, schedule the candidate threads such that the speedup is maximized while giving as soon as possible response for the synchronization with hardware.

IV. Task Decomposition and Scheduler Generation

This section presents the solution to the previous mentioned two problems in Section III. The task decomposition is oriented toward finding out candidate threads for execution in parallel with the hardware execution and restructuring the CDFG for efficient scheduling. After the task decomposition, we generate a low overhead scheduler which performs both static and dynamic scheduling.

A. Task Decomposition

There is a trade-off between the number of threads and the average length of a thread. To reduce the cost of scheduling, it is important to keep the number of threads small. But it is also important to have a sufficient number of threads which are neither directly nor indirectly dependent on the result of hardware operation, so that they can be executed while the scheduler is waiting for the completion of the corresponding hardware operation having unbounded delay. Task decomposition consists of thread partitioning and clustering which are performed in four steps described below.

Step 1: Find $P'(n_i^r)$ for each n_i^r

We define $P(n_i)$ to denote a set of nodes which have paths neither from n_i nor to n_i. It can be defined recursively as follows.

$$Pred(n_i) = \bigcup_{n_j > n_i} Pred(n_j) \cup \{n_i\} \qquad (1)$$

$$Succ(n_i) = \bigcup_{n_j < n_i} Succ(n_j) \cup \{n_i\} \qquad (2)$$

$$P(n_i) = N - Pred(n_i) - Succ(n_i) \qquad (3)$$

Successors of n_i^r can be fired only after the completion of the corresponding hardware execution because n_i^r

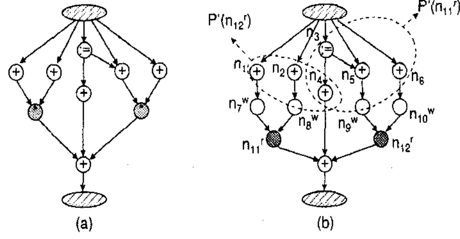

Fig. 2. An example of $P'(n_i^r)$

is assumed to be synchronized with the execution of the hardware. Nodes in $P(n_i^r)$ found by the above formulas can be fired even when the execution of n_i^r and its successors is blocked due to unavailability of data from the hardware. In other words, $P(n_i^r)$ is a set of nodes which are candidate operations to be executed concurrently with hardware components.

When there are two read nodes, n_i^r and n_j^r, there are four cases based on which hardware resources are accessed by the nodes and what kind of dependency relation exists between the two nodes.

1. $n_i^r >_t n_j^r, HW(n_i^r) \neq HW(n_j^r)$
2. $n_i^r >_t n_j^r, HW(n_i^r) = HW(n_j^r)$
3. $n_i^r \not>_t n_j^r, n_i^r \not<_t n_j^r, HW(n_i^r) \neq HW(n_j^r)$
4. $n_i^r \not>_t n_j^r, n_i^r \not<_t n_j^r, HW(n_i^r) = HW(n_j^r)$

In case 1 and 2, there is an implicit ordering between two nodes because of transitive dependency relation. In case 3, there is no ordering between two nodes, but because n_i^r and n_j^r access different hardware resources they need not be serialized. Serialization should be guaranteed in case 4 because there are no implicit ordering between two nodes and they access the same hardware resource. Not to cause hardware contention in case 4, we compute $P'(n_i^r)$ as follows. n_j^r is a node which satisfies condition in case 4.

$$P'(n_i^r) = P(n_i^r) - \bigcup_{n_k^w > n_j^r} Succ(n_k^w) \qquad (4)$$

An example of the CDFG is shown in Fig.2(a). If the two multiplication nodes depicted as shaded circles in the figure are implemented as a single hardware multiplier, the resultant CDFG of software parts becomes as in Fig.2(b). $P'(n_{11}^r)$ and $P'(n_{12}^r)$ are found by formula (4) as follows.

$$P'(n_{11}^r) = \{n_3, n_4, n_5, n_6\}, P'(n_{12}^r) = \{n_1, n_2, n_4\}$$

Step 2: Find $T(n_i^r)$ for each n_i^r

After finding $P'(n_i^r)$, we construct a thread for each n_i^r which starts from n_i^r node. From the successor nodes of n_i^r we find recursively candidates which can be included in $T(n_i^r)$. The algorithm for the construction of $T(n_i^r)$ is shown in Fig.3.

Our strategy for scheduling is based on the independency relation between a set of threads constructed from $P'(n_i^r)$ and $T(n_i^r)$. This guarantees semantically correct reordering of threads between two sets. In the procedure *NodeSerialize* of Fig.3, the first condition for inclusion of

BuildThreadofNr(n_i^r) {
  create a new thread T; $T = \{n_i^r\}$;
  for (all immediate successors n_j of n_i^r)
    NodeSerialize($n_j, P'(n_i^r), T$);
}
NodeSerialize($n_j, P'(n_i^r), T$) {
  //condition for inclusion of n_j in T
  if (($\forall n_k > n_j, n_k \notin P'(n_i^r)$) and
   ($\forall n_k > n_j, \exists T_m | n_k \in T_m$) and ($n_j^l, l \neq r$)) {
    $T \cup \{n_j\}$;
    for (all immediate successors n_k of n_j)
      NodeSerialize($n_k, P'(n_i^r), T$)
  }
}

Fig. 3. Algorithm for construction of $T(n_i^r)$.

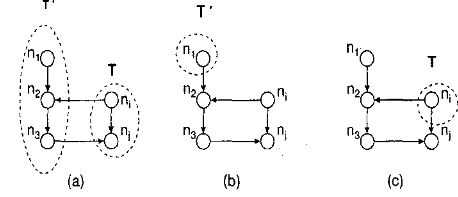

Fig. 4. Situation where deadlock is inhibited by Lemma 1.

n_j in T is necessary for this reason. The second condition guarantees a deadlock-free thread generation as explained in Lemma 1 below. A read node always starts a new thread, and this is the third condition.

Lemma 1: A sufficient condition for node n_j to be included in the thread T currently under construction without causing a deadlock situation is

$$\forall n_k > n_j, \exists T_m | n_k \in T_m$$

Let's prove this lemma informally using an example shown in Fig.4. A deadlock between two threads occurs when there is a cyclic dependency relation as depicted in Fig.4(a). However, this situation is inhibited by applying the above condition. Let's assume that T has not been constructed yet and T' is currently under construction as shown in Fig.4(b). By the above condition, n_2 can not be included in T', because n_i which is an immediate predecessor of n_2 is not included in any thread yet. Now, let's assume that T' has not been constructed yet and T is currently being constructed as shown in Fig.4(c). Again, n_j can not be included in T, because n_3 which is an immediate predecessor of n_j is not included in any thread yet. Therefore, the deadlock situation such as that in Fig.4(a) never occurs.

Step 3: Build Basic Segments

In case $P'(n_i^r)$ and $P'(n_j^r)$ have intersections, some threads constructed from $P'(n_i^r)$ may be further partitioned during the construction of threads from $P'(n_j^r)$. Fig.5 shows this situation. Fig.5(a) shows case 1 or 2 in Step 1, where we can obtain $P'(n_i^r) = \{n_1, n_2, n_3, n_4\}$ and $P'(n_j^r) = \{n_3, n_4\}$. From $P'(n_i^r)$, we can construct thread $T_1 = \{n_1, n_2, n_3, n_4\}$, whereas, from $P'(n_j^r)$, we can construct thread $T_2 = \{n_3, n_4\}$. In this case, T_1 is further partitioned to $T_i' = \{n_1, n_2\}$ and $T_2 = \{n_3, n_4\}$. Fig.5(b) shows case 4.

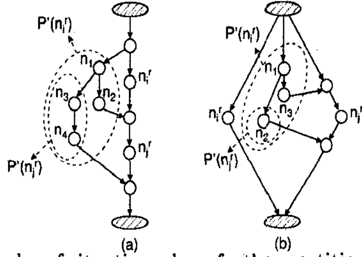

Fig. 5. Examples of situation where further partitioning is needed.

```
ClusterThreads() {
    while (there exist T_i and T_j that can be merged) {
        In(T_i) = {T_j|T_j > T_i}; Out(T_i) = {T_j|T_j < T_i};
        if ((S_Tnr(T_i) == S_Tnr(T_j)) and
            (T_i and T_j exist in the same subgraph))
            if ((In(T_i) == In(T_j)) or
                ((Out(T_i) == T_j) and (T_i ∈ In(T_j))))
                MergeThreads(T_i, T_j);
    }
}
```

Fig. 6. Thread clustering algorithm.

In our implementation, not to cause these complicated situations, we do not take construct-and-partition strategy, but take divide-and-merge strategy. For this purpose, we define a basic segment as a set of continuously connected nodes which have explicit ordering between any two nodes. This is similar as the definition of a basic block used in compiler discipline. The header node of a basic segment is a node with multiple incoming edges or a immediate successor node of a node with multiple outgoing edges. We define $S_T(n_i^r)$ as a set of threads(the basic segments) constructed from $P'(n_i^r)$.

Step 4: Cluster Threads

The number of threads found by the previous step is usually too big to be scheduled efficiently. However, we can decrease the number drastically through thread clustering, thereby reducing the scheduling overhead. The clustering process must be performed with care to ensure that the resultant threads are deadlock-free. For this purpose, we define $S_{Tnr}(T_i)$ as a set of threads $T(n_j^r)$ such that T_i is an element of $S_T(n_j^r)$. $S_{Tnr}(T_i)$ is formally defined as follows.

$$S_{Tnr}(T_i) = \{T(n_j^r) | T_i \subset S_T(n_j^r)\} \quad (5)$$

Two threads T_i and T_j are merged only when the two sets $S_{Tnr}(T_i)$ and $S_{Tnr}(T_j)$ are identical. The thread clustering algorithm is shown in Fig.6.

B. Generation of Thread Scheduler

We generate a mixed static and dynamic scheduler which statically schedules threads that can be given static orders. When the hardware starts its execution, the scheduler dispatches and fires a candidate thread from $S_T(n_i^r)$. The thread can be executed without regard to the result of the hardware execution. After the completion of execution of a selected thread, the scheduler checks the completion signal from the hardware. If this signal is not asserted, the scheduler repeats the above mentioned procedure until there are no candidates or the signal is asserted. If the signal is asserted, $T(n_i^r)$ is scheduled for the synchronization with hardware and then the list of threads whose value of $S_{Tnr}(T_i)$ are 1 is flushed by firing all of them. Execution overlap between software and hardware is achieved by this dynamic scheduling, thereby tolerating interface communication overhead.

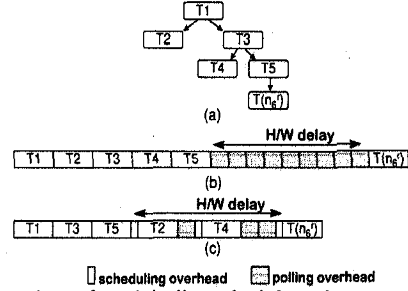

Fig. 7. Comparison of straight line schedule and proposed schedule.

All threads in $S_T(n_i^r)$ and threads $T(n_i^r)$, $1 \le i \le m$, are scheduled dynamically. The remaining threads which neither are in $S_T(n_i^r)$ nor are $T(n_i^r)$ are scheduled statically. This combined approach minimizes the number of threads to be scheduled dynamically and therefore reduces the total scheduling overhead. The sufficient condition for the threads that can be scheduled statically is justified by the following lemma. Refer to [5] for the proof.

Lemma 2: For any $T_j, T_k \in S_T(n_i^r)$, if a thread T_l satisfies both $T_j > T_l$ and $T_l > T_k$, then T_l should be in $S_T(n_i^r)$.

We assign a priority to each thread in $S_T(n_i^r)$ and the priorities remain static during runtime. Candidate threads are put in a list ordered by the priority. Dynamic scheduling selects the candidate with highest priority from the ordered list. Priority is assigned by three ordered rules. The first rule is based on dependency relations among threads. If $T_i > T_j$, then T_i gets higher priority. The second rule is based on a path length to $T(n_j^r)$ in $S_T(n_i^r)$. By the path length to $T(n_i^r)$, we mean the minimum number of threads from a thread to $T(n_j^r)$. Note that $T(n_j^r)$ accesses a different hardware resources from that of $T(n_i^r)$ by formula (4). A thread whose path length is shorter is assigned a higher priority. When the hardware consists of multiple resources, we can maximize the hardware utilization by concurrently executing as many resources as possible. This justifies the second rule. The third rule assigns a thread of which the number of threads in $S_{Tnr}(T_i)$ is smaller with a higher priority. The number of threads of $S_{Tnr}(T_i)$ indicates how many times T_i can be a candidate that can be executed in parallel with the hardware. Therefore, a thread with less chances gets a higher priority.

Fig.7 compares two schedule sequences, a schedule with a "busy-wait" type synchronization(straight line schedule) and a schedule generated by our algorithm. Assume a task is decomposed and the dependency relations are built as shown in Fig.7(a). Typical schedule with a busy-wait polling is shown in Fig.7(b). Fig.7(c) shows a schedule generated by our algorithm. With a given hardware delay, the schedule generated by our algorithm exhibits shorter execution time due to the execution overlap.

V. Experimental Results

We have performed two experiments to see the effectiveness of our algorithm. The first experiment is to compare the execution time and communication overhead between the code generated by our synthesis algorithm and the straight-line code. By straight-line code, we mean the code that enters idle wait state and remains there while the hardware is performing some task.

The second experiments are mixed implementation of Lempel-Ziv data compression algorithm. We have codesigned this example in [2], but in this paper we performed software synthesis algorithm to the software part. We have compared three kind of implementation, all-software solution, codesign which assumes mutual exclusion between software and hardware execution, codesign whose software part is restructured by our synthesis algorithm.

A. Experimental Co-design Environment

We implemented our synthesis algorithm in the C++ programming language on a SUN Sparc workstation. Our target architecture consists of an Intel 80486 processor and a prototyping board. The prototyping board contains Xilinx FPGAs(one 4025 and one 4010) and some glue logic for programming the FPGAs. Hardware components which are synthesized and prototyped with an FPGA communicate with software components via ISA Bus.

B. Example 1: Elliptical wave filter

For this experiment, we partitioned the filter design such that the multiplication operation is performed by hardware and the rest is performed by software. We intentionally put a variable delay element which asserts the completion signal to software after counting given FPGA clock cycle into the hardware part so that we can gather experimental data for various situations. The multiplier and the delay element were synthesized with an FPGA.

Fig.8 shows the experimental results. In this figure, TC indicates the code generated by our algorithm and SLC indicates the straight line code. Fig.8(a) compares the number of polling operations for various delay values. As the hardware delay increases, there are more polling operations wasting more processor time. Fig.8(b) compares the total execution time. As the hardware delay increases, TC shows better performance than SLC. This improvement is achieved by the execution overlap of software and hardware.

Usually, as the hardware delay increases, TC has more gain over SLC. This is because TC can execute more thread while waiting for the completion signal from the hardware, whereas SLC just polls the completion signal in an idle state. In our example, however, the gain saturates because there are not many threads to be executed during the hardware delay. We can have more gain for a larger system.

C. Example 2: Lempel-Ziv data compression

In this experiment, we have partitioned the system in such a way that the stream input data is handled by software and the core compression operation is performed by hardware. The hardware components are synthesized with FPGAs and the detailed description can be found in [2]

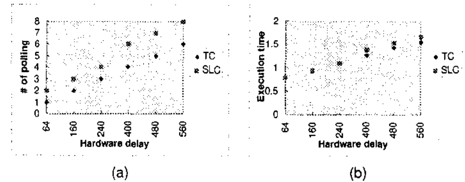

Fig. 8. Experimental results for an elliptical wave filter.

TABLE I

COMPARISON OF EXECUTION TIME OF THREE IMPLEMENTATION

	# bytes	Execution time(sec)		
		All S/W solution	Codesign with mutual exclusion	Codesign by our algorithm
File1	1320	0.28	0.17	0.11
File2	2293	0.44	0.27	0.17

We have compared three kinds of implementation and the results are shown in Table 1. By comparing the data for the mutual exclusion strategy and our strategy, we can see that about 30% speedup is obtained by our synthesis algorithm.

VI. Conclusions and Future Works

In this paper, we presented a software synthesis technique which generates codes based on threads. Our methodology tries to execute as many operations as possible before the completion of unbounded delay operations of hardware or environment, thereby reducing the total execution time. The software executions of operations are scheduled efficiently through thread partitioning and thread scheduling. It has been experimentally shown that the total execution time can be effectively reduced. The approach is more effective for larger systems. We are currently experimenting with several embedded system example.

We plan to extend our work to hardware-software codesign where a system is specified with mixed VHDL, C, and Ptolemy.

References

[1] F. Thoen, M. Cornero, G. Goossens, and H. De Man, "Real-time multi-tasking in software synthesis for information processing systems," *in Proc. of 8th Int. Symposium on System Synthesis*, pp. 48-53, 1995.

[2] K. Kim, Y. Kim, Y. Shin, and K. Choi, "An integrated hardware-software cosimulation environment with automated interface generation," *in Proc. of 7th IEEE Int. Workshop on Rapid Systems Prototyping*, pp. 66-71, June 1996.

[3] Rajesh K. Gupta, *Co-Synthesis of Hardware and Software for Digital Embedded Systems, Ph.D. thesis*, Stanford University, Dec. 1993.

[4] R. Ernst, J. Henkel, and T. Benner, "Hardware-software cosynthesis for micro-controllers," *IEEE Design & Test of Computers*, pp. 64-75, Dec. 1993.

[5] Y. Shin and K. Choi, "Thread-based software synthesis for embedded system design," *in Proc. of the European Design & Test Conf.*, pp. 282-286, Mar. 1996.

[6] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: a framework for simulating and prototyping heterogeneous systems," *Int. J. of Computer Simulation*, vol. 4, pp. 155-182, Apr. 1994.