OPTIMIZING SYNCHRONIZATION IN MULTIPROCESSOR IMPLEMENTATIONS OF ITERATIVE DATAFLOW PROGRAMS

by

Shuvra S. Bhattacharyya, Sundararajan Sriram,
and Edward A. Lee

Memorandum No. UCB/ERL M95/2

5 January 1995

OPTIMIZING SYNCHRONIZATION IN MULTIPROCESSOR IMPLEMENTATIONS OF ITERATIVE DATAFLOW PROGRAMS

by

Shuvra S. Bhattacharyya, Sundararajan Sriram,
and Edward A. Lee

Memorandum No. UCB/ERL M95/2

5 January 1995

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

OPTIMIZING SYNCHRONIZATION IN MULTIPROCESSOR IMPLEMENTATIONS OF ITERATIVE DATAFLOW PROGRAMS

by

Shuvra S. Bhattacharyya, Sundararajan Sriram,
and Edward A. Lee

Memorandum No. UCB/ERL M95/2

5 January 1995

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

OPTIMIZING SYNCHRONIZATION IN MULTIPROCESSOR IMPLEMENTATIONS OF ITERATIVE DATAFLOW PROGRAMS

Shuvra S. Bhattacharyya, Sundararajan Sriram, and Edward A. Lee

January 5, 1995

ABSTRACT

This paper is concerned with multiprocessor implementations of embedded applications specified as iterative dataflow programs, in which synchronization overhead tends to be significant. We develop techniques to alleviate this overhead by determining a minimal set of processor synchronizations that are essential for correct execution. Our study is based in the context of *self-timed* execution of *iterative dataflow* programs. An iterative dataflow program consists of a dataflow representation of the body of a loop that is to be iterated an indefinite number of times; dataflow programming in this form has been studied and applied extensively, particularly in the context of signal processing software. Self-timed execution refers to a combined compile-time/run-time scheduling strategy in which processors synchronize with one another only based on inter-processor communication requirements, and thus, synchronization of processors at the end of each loop iteration does not generally occur.

We introduce a new graph-theoretic framework for analyzing and optimizing synchronization overhead in self-timed, iterative dataflow programs. This framework is based on a data structure, which we call the *inter-processor communication* (IPC) *graph*, that was first proposed in [32] for analyzing the throughput of self-timed systems. We show that the comprehensive techniques that have been developed for removing *redundant synchronizations* in non-iterative programs can be extended in this framework to optimally remove redundant synchronizations in our context. We also introduce two new optimizations for reducing synchronization overhead in self-timed, iterative dataflow programs — *resynchronization* and the conversion of the synchronization graph into a strongly connected graph.

This research was partially funded as part of the Ptolemy project, which is supported by the Advanced Research Projects Agency and the U. S. Air Force (under the RASSP program, contract F33615-93-C-1317), Semiconductor Research Corporation (project 94-DC-008), National Science Foundation (MIP-9201605), Office of Naval Technology (via Naval Research Laboratories), the State of California MICRO program, and the following companies: Bell Northern Research, Cadence, Dolby, Hitachi, Mentor Graphics, Mitsubishi, NEC, Pacific Bell, Philips, Rockwell, Sony, and Synopsys.

S. S. Bhattacharyya is with the Semiconductor Research Laboratory, Hitachi America, Ltd., 179 East Tasman Drive., San Jose, California 95134, USA.

S. Sriram and E. A. Lee are with the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, California 94720, USA.

1. Introduction

In this paper, we address the problem of minimizing the overhead of inter-processor synchronization for an *iterative synchronous dataflow program* that is implemented on a shared-memory multiprocessor system. This study is motivated by the widespread popularity of the synchronous dataflow (SDF) model in DSP design environments; the suitability of this model for exploiting parallelism; and the high overhead of run-time synchronization, which can severely limit the speedup obtained by moving an implementation of an SDF program from a uniprocessor implementation to a multiprocessor implementation. Our work is particularly relevant when estimates are available for the task execution times, and actual execution times are usually close to the corresponding estimates, but deviations from the estimates of (effectively) arbitrary magnitude can occasionally occur due to phenomena such as cache misses or error handling.

SDF and closely related models have been used widely as foundations for numerous graphical DSP design environments, in which signal processing applications are represented as hierarchies of block diagrams. Some examples are described in [16, 22, 25, 26, 29]. In SDF, as in other forms of dataflow, a program is represented as a directed graph in which the vertices, called **actors**, represent computations, and the edges specify FIFO channels for communication between actors. The term *synchronous* refers to the requirement that the number of data values produced (consumed) by each actor onto (from) each of its output (input) edges is a fixed value that is known at compile time [18] and should not be confused with the use of "synchronous" in synchronous languages [2]. The techniques developed in this paper assume that the input SDF graph is *homogeneous*, which means that the numbers of data values produced or consumed are identically unity. However, since efficient techniques have been developed to convert general SDF graphs into equivalent (for our purposes) homogeneous SDF graphs [18], our techniques apply equally to general SDF graphs. In the remainder of this paper, when we refer to a **dataflow graph (DFG)** we imply a homogeneous SDF graph.

It is sometimes necessary to insert *delays* on the edges of a dataflow graph, to represent initial tokens on the edges. These delays (which can also be interpreted as registers) specify dependencies between iterations of the actors in iteratively executed dataflow graphs. For example, if tokens produced by the kth execution of actor A are consumed by the $(k+2)$th execution of actor B, then the edge from A to B will contain two initial tokens, or delays. We will represent an edge with n delays by annotating it with the symbol "nD" in the dataflow graph representation (see Figure 1).

Multiprocessor implementation of an algorithm specified as a DFG involves scheduling computations in the algorithm. By "scheduling" we collectively refer to the task of assigning actors in the DFG to processors, ordering execution of these actors on each processor, and determining when each actor fires (begins execution) such that all data precedence constraints are met. Each of these three tasks may be performed either at run time (a dynamic strategy) or at compile time (a static strategy). In [19] and [20] the authors propose a scheduling taxonomy based on which of these tasks are performed at compile time and which at run time; in this paper we will use the same terminology that was introduced there. To reduce run time computation costs it is advantageous to perform as many of the three scheduling tasks as possible at compile time. Which of these can be effectively performed at compile time depends on the information available about the execution time of each actor — that is, on the amount of time it takes for each actor to complete execution once it fires.

The performance metric that is of interest for evaluating schedules is the average iteration period T, which is the average time that it takes for all the actors in the graph to be executed once. Equivalently, we could use the throughput T^{-1} (that is, the number of iterations of the graph executed per unit time) as a performance metric. Thus an optimal schedule is one that minimizes T.

In the **fully-static** scheduling strategy of [4], all the three scheduling tasks — assigning actors to processors, ordering their execution on each processor, as well as determining exactly when an actor fires — are performed at compile time. This strategy involves the least possible amount of runtime overhead. All the processors run in lock step and no explicit synchronization is required when they exchange data. However, this strategy assumes that exact execution times of actors are known. Such an assumption is in general not practical. A more realistic assumption for DSP algorithms is that good estimates for the execution times of actors can be obtained. These estimates may not be accurate down to the clock cycle, because the object code for the processors might be compiled from a high-level language, which makes estimation of exact execution time difficult, or the processor itself might take a non-deterministic number of cycles to complete an instruction, if it employs a cache for instance. These estimates may not even have guaranteed worst case bounds, if, for example, at run time a processor has to respond to events that require error handling or has to process user inputs, which are infrequent (rare) compared to the sample rate at which the DFG executes.

Under such an assumption on timing, it is best to discard the exact timing information from the fully static schedule, but still retain the processor assignment and actor ordering specified by the fully static schedule. This results in the **self-timed** scheduling strategy of [19]. Each processor executes the actors assigned to it in the order specified at compile time. Before firing an actor, a processor waits for the data needed by that actor to become available. Thus in self-timed scheduling processors are required to perform run-time synchronization when they communicate data. Such synchronization is not necessary in the fully-static case because exact (or guaranteed worst case) times could be used to determine firing times of actors such that processor synchronization was ensured. As a result, the self-timed strategy incurs greater run-time cost than the fully-static case because of the synchronization overhead.

An example of a DFG and a corresponding self-timed schedule are shown in Figure 1. Note that inter-processor communication primitives (*send* and *receive* actors) need to be inserted when data cross processor boundaries. As a result, a multiprocessor schedule for a DFG falls naturally into a message passing inter-processor communication model. The execution times for the

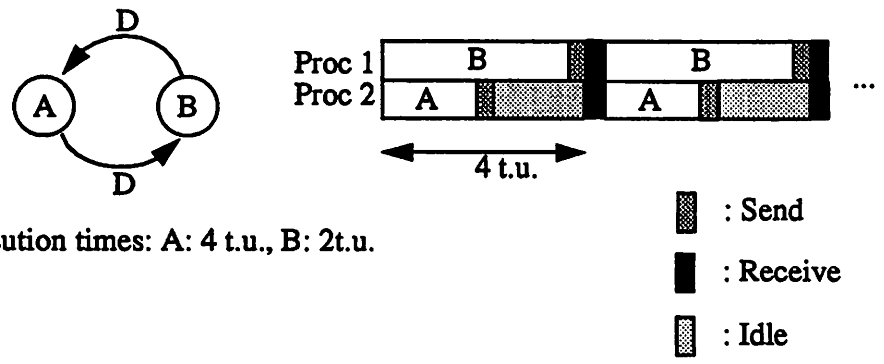


Figure 1. An example of a self-timed schedule.

actors A and B are estimates that are used to determine the processor assignment and ordering for the schedule. However, the processors need to explicitly synchronize at each communication point since the estimated execution times may not be exact or may vary from one iteration of the DFG to the next. Clearly, if these times were known precisely, we could eliminate the need for explicit synchronization by determining precisely when each actor fires and when the send and the receive primitives are executed. If we ignore communication costs, that is, we assume *sends* and *receives* take zero time, then the estimated iteration period (T) for this example is 4 time units.

A straightforward implementation of a self-timed schedule would require that for each inter-processor communication (IPC) the sending processor ascertains that the buffer it is writing to is not full, and the receiver ascertains that the buffer it is reading from is not empty. The processors block (suspend execution) when the appropriate condition is not met. Such sender-receiver synchronization can be implemented in many ways depending on the particular hardware platform under consideration: in shared memory machines, such synchronization involves testing and setting semaphores in shared memory; in machines that support synchronization in hardware (such as barriers), special synchronization instructions are used; and in the case of systems that consist of a mix of programmable processors and custom hardware elements, synchronization is achieved by employing interfaces that support blocking reads and writes.

In each kind of platform, every IPC that requires a synchronization check costs performance, and sometimes extra hardware complexity: semaphore checks cost execution time on the processors, synchronization instructions that make use of synchronization hardware also cost execution time, and blocking interfaces in hardware/software implementations require more hardware than non-blocking interfaces [12].

The main goal of this paper is to present algorithms and techniques that reduce the rate at which processors must access shared memory for the purpose of synchronization in embedded, shared-memory multiprocessor implementations of iterative dataflow programs. Thus the optimization procedure that we propose can be used as a post-processing step in any static scheduling technique for reducing synchronization costs in the final implementation. In this paper we assume that "good" estimates are available for the execution times of actors and that these execution times rarely display large variations so that self-timed scheduling is viable for the applications under consideration. If additional timing information is available, such as guaranteed upper and lower bounds on the execution times of actors, it is possible to use this information to further optimize synchronizations in the schedule. However, use of such timing bounds is beyond the scope of this paper.

Our paper is organized as follows. In Section 2 we review some of the related work in synchronization optimization, and in Section 3 we list some of the notation and terminology used in this paper. Sections 4, 5 and 6 present our graph-theoretic framework for analyzing and optimizing synchronization. In Section 7, we formally define the optimization problem addressed in this paper in terms of the model and results developed in Sections 4-6. Sections 8, 9 and 10 present the algorithms used for our proposed synchronization optimization scheme. Finally, in Section 11 we present the complete synchronization algorithm, and then end with conclusions in Section 12, and discuss directions for future work in Section 13. For reference, some of the terminology and notation used in this paper is summarized in a glossary at the end of the paper.

2. Related Work

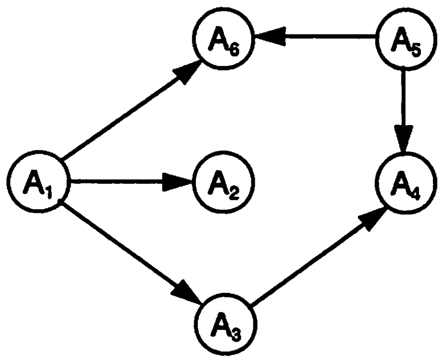
Numerous research efforts have focused on constructing efficient parallel schedules for DFGs. Parhi and Messerschmitt [23], and Chao and Sha [6] have developed systematic techniques for exploiting overlapped execution to generate schedules that have optimal throughput, assuming zero cost for IPC. Other work has focused on taking IPC costs into account during scheduling, such as that described in [1, 21, 27, 31]; these efforts have not attempted to exploit overlapped execution of dataflow graph iterations. Similarly, in [10], Govindarajan and Gao develop techniques to simultaneously maximize throughput, possibly using overlapped execution, and minimize buffer memory requirements under the assumption of zero IPC cost. Our work can be used as a post-processing step to improve the performance of implementations that use any of these scheduling techniques when the goal is a self-timed implementation.

Among the prior work that is most relevant to this paper is the *barrier-MIMD* principle of Dietz, Zaafrani, and O'keefe, which is a combined hardware and software solution to reducing run-time synchronization overhead [8]. In this approach, a shared-memory MIMD computer is augmented with hardware support that allows arbitrary subsets of processors to synchronize precisely with respect to one another by executing a synchronization operation called a *barrier.* If a subset of processors is involved in a barrier operation, then each processor in this subset will wait at the barrier until all other processors in the subset have reached the barrier. After all processors in the subset have reached the barrier, the corresponding processes resume execution in *exact synchrony.*

In [8], the barrier mechanism is applied to minimize synchronization overhead in a self-timed schedule with hard lower and upper bounds on the task execution times. The execution time ranges are used to detect situations where the earliest possible execution time of a task that requires data from another processor is guaranteed to be later than the latest possible time at which the required data is produced. When such an inference cannot be made, a barrier is instantiated between the sending and receiving processors. In addition to performing the required data synchronization, the barrier resets (to zero) the uncertainty between the relative execution times for the processors that are involved in the barrier, and thus enhances the potential for subsequent timing analysis to eliminate the need for explicit synchronizations.

The techniques of barrier MIMD do not apply to the problem that we address because they assume that a hardware barrier mechanism exists; they assume that tight bounds on task execution times are available; they do not address iterative, self-timed execution, in which the execution of successive iterations of the dataflow graph can overlap; and even for non-iterative execution, there is no obvious correspondence between an optimal solution that uses barrier synchronizations and an optimal solution that employs decoupled synchronization checks at the sender and receiver end (**directed synchronization**). This last point is illustrated in Figure 2. Here, in the absence of execution time bounds, an optimal application of barrier synchronizations can be obtained by inserting two barriers — one barrier across A_1 and A_3, and the other barrier across A_4 and A_5. This is illustrated in Figure 2(c). However, the corresponding collection of directed synchronizations (A_1 to A_3, and A_5 toA_4) is not sufficient since it does not guarantee that the data required by A_6 from A_1 is available before A_6 begins execution.

In [30], Shaffer presents an algorithm that minimizes the number of directed synchronizations in the self-timed execution of a dataflow graph. However, this work, like that of Dietz *et al.*, does not allow the execution of successive iterations of the dataflow graph to overlap. It also avoids having to consider dataflow edges that have delay. The technique that we present for removing redundant synchronizations can be viewed as a generalization of Shaffer's algorithm to handle delays and overlapped, iterative execution, and we will discuss this further in Section 8. The other major techniques that we present for optimizing synchronization — handling the feed-forward edges of the *synchronization graph* (to be defined in Section 6), discussed in Section 10,



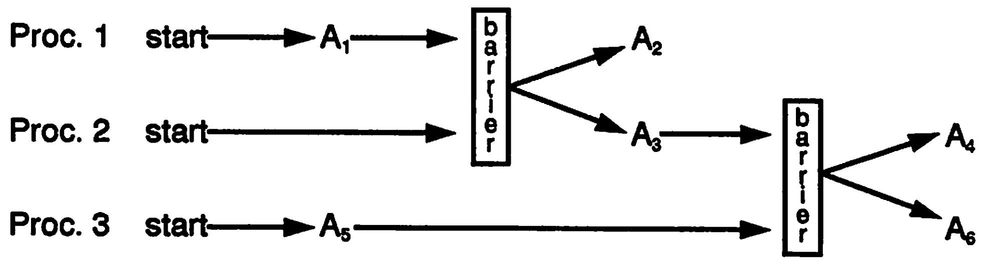
(a)

Proc. 1: A_1, A_2

Proc. 2: A_3, A_4

Proc. 3: A_5, A_6

(b)



(c)

Figure 2. (a). A DFG.
(b). A three-processor self-timed schedule for (a).
(c). An illustration of execution under the placement of barriers.

and “resynchronization”, defined and addressed in Sections 9 and the appendix — are fundamentally different from Shaffer’s technique since they address issues that are specific to our more general context of overlapped, iterative execution.

3. Background Terminology and Notation

We frequently represent a DFG by an ordered pair (V, E), where V is the set of vertices and E is the set of edges. We refer to the source and sink vertices of a graph edge e by $src(e)$ and $snk(e)$, and we denote the delay on e by $delay(e)$. We say that e is an **output edge** of $src(e)$, and that e is an **input edge** of $snk(e)$.

Given $x, y \in V$, we say that x is a **predecessor** of y if there exists $e \in E$ such that $src(e) = x$ and $snk(e) = y$; we say that x is a **successor** of y if y is a predecessor of x. A **path** in (V, E) is a finite, nonempty sequence $(e_1, e_2, \ldots, e_n)$, where each e_i is a member of E, and $snk(e_1) = src(e_2)$, $snk(e_2) = src(e_3)$, …, $snk(e_{n-1}) = src(e_n)$. We say that the path $p = (e_1, e_2, \ldots, e_n)$ **contains** each e_i and each subsequence of $(e_1, e_2, \ldots, e_n)$; p is **directed from** $src(e_1)$ **to** $snk(e_n)$; and each member of

$\{src(e_1), src(e_2), \ldots, src(e_n), snk(e_n)\}$ is **on** p. A path that is directed from some vertex to itself is called a **cycle**, and a **fundamental cycle** is a cycle of which no proper subsequence is a cycle.

If $p = (e_1, e_2, \ldots, e_n)$ is a path in a DFG, then we define the **path delay** of p, denoted $Delay(p)$, by $Delay(p) = \sum_{i=1}^{n} delay(e_i)$. Since the delays on all DFG edges are restricted to be non-negative, it is easily seen that between any two vertices $x, y \in V$, either there is no path directed from x to y, or there exists a (not necessarily unique) **minimum-delay path** between x and y. That is, if there is a path from x to y, then there exists a path p from x to y such that

$Delay(p') \geq Delay(p)$, for all paths p' directed from x to y. Given a DFG G, and vertices x, y in G, we define $\rho_G(x, y)$ to be equal to ∞ if there is no path from x to y, and equal to the path delay of a minimum-delay path from x to y if there exist one or more paths from x to y. If G is understood, then we may drop the subscript and simply write "ρ" in place of "ρ_G".

By a **subgraph** of (V, E), we mean the directed graph formed by any $V' \subseteq V$ together with the set of edges $\{e \in E | src(e), snk(e) \in V'\}$. We denote the subgraph associated with the vertex-subset V' by $subgraph(V')$. We say that (V, E) is **strongly connected** if for each pair of distinct vertices x, y, there is a path directed from x to y and there is a path directed from y to x. We say that a subset $V' \subseteq V$ is strongly connected if $subgraph(V')$ is strongly connected. A **strongly connected component** (SCC) of (V, E) is a strongly connected subset $V' \subseteq V$ such that no strongly connected subset of V properly contains V'. If V' is an SCC, then when there is no ambiguity, we may also say that $subgraph(V')$ is an SCC. If C_1 and C_2 are distinct SCCs in (V, E), we say that C_1 is a **predecessor** SCC of C_2 if there is an edge directed from some vertex in C_1 to some vertex in C_2; C_1 is a **successor** SCC of C_2 if C_2 is a predecessor SCC of C_1. An SCC is a **source** SCC if it has no predecessor SCC; and an SCC is a **sink** SCC if it has no successor SCC. An edge e is a **feedforward** edge of (V, E) if it is not contained in an SCC, or equivalently, if it is not contained in a cycle; an edge that is contained in at least one cycle is called a **feedback** edge.

Given two arbitrary sets S_1 and S_2, we define the difference of these two sets by $S_1 - S_2 = \{s \in S_1 | s \notin S_2\}$, and we denote the number of elements in a finite set S by $|S|$. Also, if r is a real number, then we denote the smallest integer that is greater than or equal to r by $\lceil r \rceil$.

In this paper, we assume that the source and sink vertices uniquely identify an edge in a DFG, and thus we may represent an edge $e \in E$ by the ordered pair $(src(e), snk(e))$. It is conceivable, however, that a practical system may have a DFG in which one or more pairs of vertices have multiple edges connecting them in the same "direction". Such graphs can very easily be preprocessed into a form to which the techniques of this paper can be applied; the details are beyond the scope of this paper. Finally, if x, y are vertices in (V, E), we define $d_n(x, y)$ to represent an edge (that is not necessarily in E) whose source and sink vertices are x and y, respectively, and whose delay is n (assumed non-negative).

For elaboration on any of the graph-theoretic concepts presented in this section, we refer the reader to [7].

4. Model of a Multiprocessor Executing a Self-timed Schedule

We model a multiprocessor executing a self-timed schedule as follows. Each processor is assigned a sequential list of actors, some of which are *send* and *receive* actors, which it executes in an infinite loop. When a processor executes a communication actor, it synchronizes with the processor(s) it communicates with. Thus exactly when a processor executes each actor depends on when, at run time, all input data for that actor is available, unlike the fully-static case where no such run time check is needed. In this paper we use "processor" in slightly general terms: a processor could be a programmable component, in which case the actors mapped to it execute as software entities, or it could be a hardware component, in which case actors assigned to it are implemented and execute in hardware. See [13] for a discussion on combined hardware/software synthesis from a single dataflow specification. Examples of application specific multiprocessors that use programmable processors and some form of static scheduling are described in [4, 14, 33].

Inter-processor communication between processors is assumed to take place via shared memory. Thus the sender writes to a particular shared memory location and the receiver reads from that location. The shared memory itself could be global memory between all processors, or it could be distributed between pairs of processors (as a hardware first-in-first-out (FIFO) queues or dual ported memory for example). Each inter-processor communication edge in our DFG thus translates into a buffer of a certain size (which we will discuss later) in shared memory.

Sender-receiver synchronization is also assumed to take place by setting flags in shared memory. Special hardware for synchronization (barriers, semaphores implemented in hardware, etc.) would be prohibitive for the embedded multiprocessor machines for applications such as DSP that we are considering.

Interfaces between hardware and software are typically implemented using memory-mapped registers in the address space of the programmable processor (again a kind of shared memory), and synchronization is achieved using flags that can be tested and set by the programmable component, and the same can be done by an interface controller on the hardware side [12].

Under the model above, the benefits that our proposed synchronization optimization techniques offer become abundantly clear. Each synchronization that we eliminate directly results in one less synchronization check, or a shared memory access. For example, where a processor would have to check a flag in shared memory before executing a *receive* primitive, eliminating that synchronization implies there is no longer need for such a check. This translates to one less shared memory read. Such a benefit is especially significant for simplifying interfaces between a programmable component and a hardware component: a *send* or a *receive* without the need for synchronization implies that the interface can be implemented in a non-blocking fashion, greatly simplifying the interface controller. As a result, eliminating a synchronization directly results in simpler hardware in this case.

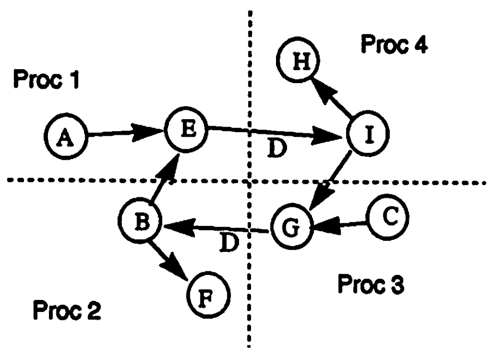
Thus the metric for the optimizations we present in this paper is the total number of accesses to shared memory that are needed for the purpose of synchronization in the final multiprocessor implementation of the self-timed schedule. This metric will be defined precisely in Sections 6 and 7.

5. Analysis of Self-Timed Execution

In this section we develop an analytical model to study the execution of a self-timed schedule. To motivate this section, let us consider the execution of the four-processor schedule in Figure 3. Inter-processor communication is ignored in the self-timed execution in Figure 3(c). If the timing estimates are accurate, the schedule execution settles into a repeating pattern spanning two iterations of G, and the average estimated iteration period is 7 time units.

We would like to model such a self-timed execution and determine the average iteration period, represent the sequential execution of actors assigned to a single processor, and represent dependencies across iterations of the DFG.

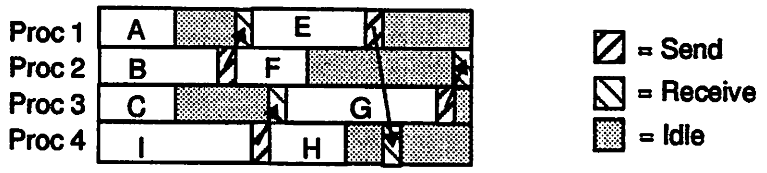
5.1 Inter-processor Communication Modelling Graph



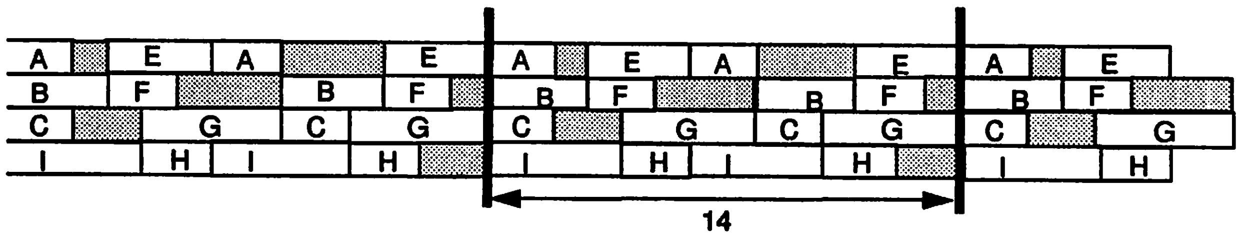
<u>Execution Time Estimates</u>

A, C, H, F	:2
B, E	:3
G, I	:4

(a) DFG "G"



(b) Schedule on four processors



(c) Self-timed execution

Figure 3. Self-timed execution.

$$G_{ipc} = (V, E_{ipc})$$

We model a self-timed schedule using a DFG $G_{ipc} = (V, E_{ipc})$ derived from the original SDF graph $G = (V, E)$ and the given self-timed schedule. The graph G_{ipc}, which we will refer to as the **inter-processor communication modelling graph**, or **IPC graph** for short, models the fact that actors of G assigned to the same processor execute sequentially, and it models constraints due to inter-processor communication. For example, the self-timed schedule in Figure 3 can be modelled by the IPC graph in Figure 4. The IPC edges are shown using dashed arrows. The rest of this subsection describes the construction of the IPC graph in detail.

The IPC graph has the same vertex set V as G, corresponding to the set of actors in G. The self-timed schedule specifies the actors assigned to each processor, and the order in which they execute. For example in Figure 3, processor 1 executes A and then E repeatedly. We model this in G_{ipc} by drawing a cycle around the vertices corresponding to A and E, and placing a delay on the edge from E to A. The delay-free edge from A to E represents the fact that the kth execution of A precedes the kth execution of E, and the edge from E to A with a delay represents the fact that the kth execution of A can occur only after the $(k-1)$th execution of E has

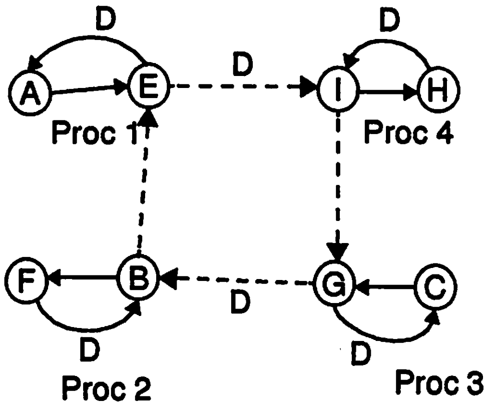


Figure 4. The IPC graph for the schedule in Figure 3.

completed. Thus if actors $v_1, v_2, \ldots, v_n$ are assigned to the same processor in that order, then G_{ipc} would have a cycle $((v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n), (v_n, v_1))$, with $delay((v_n, v_1)) = 1$. If there are P processors in the schedule, then we have P such cycles corresponding to each processor.

As mentioned before, edges in G that cross processor boundaries after scheduling represent inter-processor communication. We will call such edges IPC edges. Instead of explicitly introducing special *send* and *receive* primitives at the ends of the IPC edges, we will model these operations as part of the sending and receiving actors themselves. For example, in Figure 3, data produced by actor B is sent from processor 2 to processor 1; instead of inserting explicit communication primitives in the schedule, the send is modelled within actor B while the receive is modelled as part of actor E. This is done so as not to clutter G_{ipc} with extra communication actors. Even if the actual implementation uses explicit send and receive actors, communication can still be modelled in the above fashion because we are simply clustering the source of an IPC edge with the corresponding send actor and the sink with the receive actor.

For each IPC edge in G we add an IPC edge e in G_{ipc} between the same actors. We also set the delay on this edge equal to the delay, $delay(e)$, on the corresponding edge in G. Thus, we add an IPC edge from E to I in G_{ipc} with a single delay on it. The delay corresponds to the fact that execution of E is allowed to lag the execution of I by one iteration. An IPC edge represents a buffer implemented in shared memory, and initial tokens on the IPC edge are used to initialize the shared buffer. In a straightforward self-timed implementation, each such IPC edge would also be a synchronization point between the two communicating processors. Part of our goal is to identify IPC edges that do not require sender synchronization or receiver synchronization.

The IPC graph has the same semantics as a DFG, and its execution models the execution of the corresponding self-timed schedule. The following definitions are useful to formally state the constraints represented by the IPC graph. Time is modelled as an integer that can be viewed as a multiple of a base clock.

Definition 1: The function $start(v, k) \in Z^+$ (non-negative integer) represents the time at which the kth execution of the actor v starts in the self-timed schedule. The function $end(v, k) \in Z^+$ represents the time at which the kth execution of the actor v ends, and v produces data tokens at its output edges. Since we are interested in the kth execution of each actor for $k = 1, 2, 3, \ldots$, we set $start(v, k) = 0$ and $end(v, k) = 0$ for $k \le 0$ as the "initial conditions".

As per the semantics of a DFG, each edge (v_j, v_i) of G_{ipc} represents the following data dependency constraint:

$$start(v_i, k) \ge end(v_j, k - delay((v_j, v_i))),\ \forall (v_j, v_i) \in E_{ipc}, \forall k > delay(v_j, v_i). \qquad (1)$$

This is because each actor consumes one token from each of its input edges when it fires. Since there are already $delay(e)$ tokens on each incoming edge e of actor v, another $k - delay(e)$ tokens must be produced on e before the kth execution of v can begin. Thus the actor $src(e)$ must have completed its $(k - delay(e))$ th execution before v can begin its kth execution. The constraints in (1) are due both to IPC edges (representing synchronization between processors) and to edges that represent serialization of actors assigned to the same processor.

To model execution times of actors we associate execution time $t(v)$ with each vertex of the IPC graph; $t(v)$ assigns a positive integer execution time to each actor v (again, the actual execution time can be interpreted as $t(v)$ cycles of a base clock), and $t(v)$ includes the time taken to execute all IPC operations (*sends* and *receives*) that the actor v performs. Thus the IPC graph is $G_{ipc} = (V, E_{ipc}, t)$. Now, we can substitute

$$end(v_j, k) = start(v_j, k) + t(v_j)$$

in (1) to obtain

$$start(v_i, k) \ge start(v_j, k - delay((v_j, v_i))) + t(v_j) \text{ for each edge } (v_j, v_i) \text{ in } G_{ipc}. \qquad (2)$$

In the self-timed schedule, actors fire as soon as data is available at all their input edges.

Such an "as soon as possible" (ASAP) firing pattern implies:

$$start(v_i, k) = max\Big(\{s(v_j, k-delay((v_j, v_i))) + t(v_j) \mid (v_j, v_i) \in E_{ipc}\}\Big). \tag{3}$$

The IPC graph can also be looked upon as a Marked graph [24] or Reiter's computation graph [28]. The same properties hold for it, and we state some of the relevant properties here. Some of the proofs are omitted.

Lemma 1: [28] Every cycle C in the IPC graph has a path delay of at least one if and only if the static schedule it is constructed from is free of deadlock. That is, for each cycle C, $Delay(C) > 0$.

Lemma 2: The number of tokens in any cycle of the IPC graph is always conserved over all possible valid firings of actors in the graph, and is equal to the path delay of that cycle.

Proof: For each cycle C in the IPC graph, the number of tokens on C can only change when actors that are on it fire, because actors not on C remove and place tokens only on edges that are not part of C. If $C = ((v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n), (v_n, v_1))$, and any actor v_k $(1 \le k \le n)$ fires, then in a valid firing exactly one token is moved from the edge (v_{k-1}, v_k) to the edge (v_k, v_{k+1}), where $v_0 \equiv v_n$ and $v_{n+1} \equiv v_1$. This conserves the total number of tokens on C. *QED.*

Lemma 3: The asymptotic iteration period for a *strongly connected* IPC graph G when actors execute as soon as data is available at all inputs is given by [28]:

$$T = \max_{\text{cycle } C \text{ in } G} \left\{ \frac{\sum_{v \text{ is on } C} t(v)}{Delay(C)} \right\}. \tag{4}$$

Note that $Delay(C) > 0$ from Lemma 1.

The quotient in (4) is called the **cycle mean** of the cycle C. That is, the cycle mean of C is the sum of the execution times of all vertices on C divided by the path delay of C. The entire quantity on the RHS of (4) is called the "maximum cycle mean" of the strongly connected IPC graph G. If the IPC graph contains more than one SCC, then different SCCs may have different asymptotic iteration periods, depending on their individual maximum cycle means. In such a case, the iteration period of the overall graph (and hence the self-timed schedule) is the *maximum* over the maximum cycle means of all the SCCs of G_{ipc}. This is because the execution of the schedule is constrained by the slowest component in the system. Henceforth, we will use the following definition for the maximum cycle mean.

Definition 2: The **maximum cycle mean** of an IPC graph G_{ipc}, denoted by λ_{max}, is the maximal cycle mean over all strongly connected components of G_{ipc}: That is,

$$\lambda_{max} = \max_{\text{cycle } C \text{ in } G} \left\{ \frac{\sum_{v \text{ is on } C} t(v)}{Delay(C)} \right\}.$$

A fundamental cycle in G_{ipc} whose cycle mean is equal to λ_{max} is called a **critical cycle** of G_{ipc}. Thus the throughput of the system of processors executing a particular self-timed schedule is equal to the corresponding $\frac{1}{\lambda_{max}}$ value.

For example, in Figure 4, G_{ipc} has one SCC, and its maximal cycle mean is 7 time units. This corresponds to the critical cycle $((B, E), (E, I), (I, G), (G, B))$: $t(B) = t(E) = 3$ time units, $t(I) = t(G) = 4$ time units, so the total time along this cycle is 14 t.u., and there are two delays on this cycle. Thus the average iteration period for this schedule is 7 t.u. We have not included IPC costs in this calculation, but these can be included in a straightforward manner by adding the *send* and *receive* costs to the corresponding actors performing these operations.

The maximum cycle mean can be calculated in time $O(|V||E_{ipc}|\log_2(|V| + D + T))$ [17]

by repeated applications of the Bellman-Ford shortest-path algorithm. Here, D and T are such that $delay(e) \leq D \ \forall e \in E_{ipc}$ and $t(v) \leq T \ \forall v \in V$. If D and T are constants, the complexity of determining λ_{max} is simply $O(|V||E_{ipc}|\log_2(|V|))$.

5.2 Execution Time Estimates

If we only have execution time estimates available instead of exact values, and we set $t(v)$ in the previous section to be these estimated values, then we obtain the *estimated* iteration period by calculating λ_{max}. Henceforth we will assume that we know the **estimated throughput** $\frac{1}{\lambda_{max}}$ calculated by setting the $t(v)$ values to the available timing estimates.

In all the transformations that we present in the rest of the paper, we will preserve the estimated throughput by preserving the maximum cycle mean of G_{ipc}, with each $t(v)$ set to the estimated execution time of v. In the absence of more precise timing information, this is the best we can hope to do.

5.3 Strongly Connected Components and Buffer Size Bounds

In dataflow semantics, the edges between actors represent infinite buffers. Accordingly, the edges of the IPC graph are potentially buffers of infinite size. However, from Lemma 2, every feedback edge (an edge that belongs to a strongly connected component, and hence to some cycle) can only have a finite number of tokens at any time during the execution of the IPC graph. We will call this constant the **self-timed buffer bound** of that edge, and for a feedback edge e we will represent this bound by $B_{fb}(e)$. Lemma 2 yields the following self-timed buffer bound:

$$B_{fb}(e) = min(\{Delay(C) \mid C \text{ is a cycle that contains } e\}) \quad (5)$$

Feedforward edges have no such bound on buffer size; therefore for practical implementations we need to *impose* a bound on the sizes of these edges. For example, Figure 5(a) shows an

IPC graph where the IPC edge (A, B) could be unbounded when the execution time of A is less than that of B, for example. In practice, we need to bound the buffer size of such an edge; we will

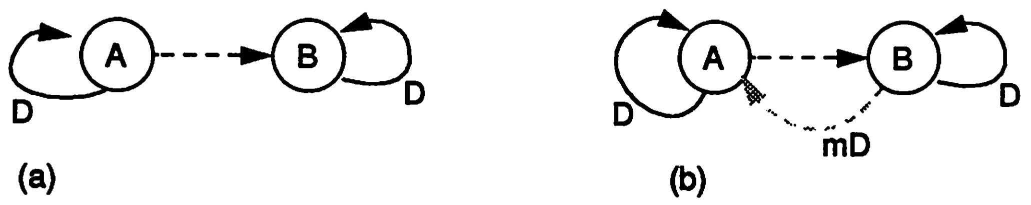


Figure 5. An IPC graph with a feedforward edge: (a). original graph (b). imposing bounded buffers.

denote such an "imposed" bound for a feedforward edge e by $B_{ff}(e)$. Since the effect of placing such a restriction includes "artificially" constraining $src(e)$ from getting more than $B_{ff}(e)$ invocations ahead of $snk(e)$, its effect on the estimated throughput can be modelled by adding the reverse edge $d_m(snk(e), src(e))$, where $m = B_{ff}(e) - delay(e)$, to G_{ipc} (grey edge in Figure 5(b)). Since the addition of this edge introduces a new cycle in G_{ipc}, it has the potential to reduce the estimated throughput; to prevent such a reduction, $B_{ff}(e)$ must be chosen to be large enough so that the maximum cycle mean remains unchanged upon adding $d_m(snk(e), src(e))$.

Sizing buffers optimally such that the maximum cycle mean remains unchanged has been studied by Kung, Lewis and Lo in [15], where the authors propose an integer linear programming formulation of the problem, with the number of constraints equal to the number of fundamental cycles in the DFG (potentially an exponential number of constraints).

An efficient albeit suboptimal procedure to determine B_{ff} is to note that if

$$B_{ff}(e) \ge \left\lceil \left(\sum_{x \in V} t(x) \right) / \lambda_{max} \right\rceil$$

holds for each feedforward edge e, then the maximum cycle mean of the resulting graph does not exceed λ_{max}.

Then, doing a binary search on $B_{ff}(e)$ for each feedforward edge, and computing the maximum cycle mean at each search step and ascertaining that it is less than λ_{max} results in a buffer assignment for the feedforward edges. Although this procedure is efficient, it is greedy (and suboptimal) because the order that the edges e are chosen is arbitrary and may effect the quality of the final solution.

However, as we will see in Section 10, imposing such a bound B_{ff} is a *naive* approach for bounding buffer sizes and, in terms of synchronization costs, there is a better technique for bounding buffers. Thus, in our final algorithm, we will not in fact find it necessary to use or compute these bounds B_{ff}.

6. Synchronization Model

6.1 Synchronization Protocols

We define two basic synchronization protocols for an IPC edge based on whether or not the length of the corresponding buffer is guaranteed to be bounded from the analysis presented in the previous section. Given an IPC graph G, and an IPC edge e in G, if the length of the corresponding buffer is not bounded — that is, if e is a feedforward edge of G — then we apply a synchronization protocol called **unbounded buffer synchronization (UBS)**, which guarantees that (a) an invocation of $snk(e)$ never attempts to read data from the buffer unless the buffer contains at least one token; *and* (b) an invocation of $src(e)$ never attempts to write data into the buffer unless the number of tokens in the buffer is less than some pre-specified limit $B_{ff}(e)$, which is the amount of memory allocated to the buffer as discussed in subsection 5.3.

On the other hand, if the topology of the IPC graph guarantees that the buffer length for e is bounded by some value $B_{fb}(e)$ (the self-timed buffer bound of e), then we use a simpler protocol, called **bounded buffer synchronization (BBS)**, that only explicitly ensures (a) above. Below, we outline the mechanics of the two synchronization protocols that we have defined.

BBS. In this mechanism, a *write pointer* $wr(e)$ for e is maintained on the processor that

executes $src(e)$; a *read pointer* $rd(e)$ for e is maintained on the processor that executes $snk(e)$; and a copy of $wr(e)$ is maintained in some shared memory location $sv(e)$. The pointers $rd(e)$ and $wr(e)$ are initialized to zero and $delay(e)$, respectively. Just after each execution of $src(e)$, the new data value produced onto e is written into the shared memory buffer for e at offset $wr(e)$; $wr(e)$ is updated by the following operation —
$wr(e) \leftarrow (wr(e) + 1) \bmod B_{fb}(e)$; and $sv(e)$ is updated to contain the new value of $wr(e)$. Just before each execution of $snk(e)$, the value contained in $sv(e)$ is repeatedly examined until it is found to be *not equal* to $rd(e)$; then the data value residing at offset $rd(e)$ of the shared memory buffer for e is read; and $rd(e)$ is updated by the operation
$rd(e) \leftarrow (rd(e) + 1) \bmod B_{fb}(e)$.

UBS. This mechanism also uses the read/write pointers $rd(e)$ and $wr(e)$, and these are initialized the same way; however, rather than maintaining a copy of $wr(e)$ in the shared memory location $sv(e)$, we maintain a count (initialized to $delay(e)$) of the number of unread tokens that currently reside in the buffer. Just after $src(e)$ executes, $sv(e)$ is repeatedly examined until its value is found to be less than $B_{ff}(e)$; then the new data value produced onto e is written into the shared memory buffer for e at offset $wr(e)$; $wr(e)$ is updated as in BBS (except that the new value is not written to shared memory); and the count in $sv(e)$ is incremented. Just before each execution of $snk(e)$, the value contained in $sv(e)$ is repeatedly examined until it is found to be nonzero; then the data value residing at offset $rd(e)$ of the shared memory buffer for e is read; the count in $sv(e)$ is decremented; and $rd(e)$ is updated as in BBS.

Note that we are assuming that there is enough shared memory to hold a separate buffer of size $B_{ff}(e)$ for each feedforward IPC edge e of G_{ipc} , and a separate buffer of size $B_{fb}(e)$ for each feedback IPC edge e . When this assumption does not hold, smaller bounds on some of the buffers must be imposed, possibly for feedback edges as well as for feedforward edges, and in general, this may require some sacrifice in estimated throughput. Note that whenever a buffer bound smaller than $B_{fb}(e)$ is imposed on a feedback edge e, then a protocol identical to UBS must be used. The problem of optimally choosing which edges should be subject to stricter buffer bounds when there is a shortage of shared memory, and the selection of these stricter bounds is an interesting area for further investigation.

6.2 The Synchronization Graph $G_s = (V, E_s)$

An IPC edge in G_{ipc} represents two functions: 1) reading and writing of data values into the buffer represented by that edge; and 2) synchronization between the sender and the receiver, which could be implemented with the UBS protocol or with the BBS protocol. We find it useful to differentiate these two functions by creating another graph called the **synchronization graph** (G_s), in which edges between actors assigned to different processors, called **synchronization edges**, represent *synchronization constraints only*. Recall from Subsection 5.1 that an IPC edge (v_j, v_i) of G_{ipc} represents the **synchronization constraint**:

$$start(v_i, k) \geq end(v_j, k - delay((v_j, v_i)))\ \forall k > delay(v_j, v_i)\ . \qquad (6)$$

Thus, before we perform any optimization on synchronizations, the synchronization graph is identical to the IPC graph, because every IPC edge represents a synchronization point. However, we will modify the synchronization graph in certain "valid" ways (which will be defined shortly) by adding some edges and deleting some others. Thus, at the end of our optimizations, the synchronization graph may look very different from the IPC graph: it is of the form $(V, (E_{ipc} - F + F'))$, where F is the set of edges deleted from the IPC graph and F' is the set of edges added to it. At this point the IPC edges in G_{ipc} represent buffer activity, and must be implemented as buffers in shared memory, whereas the synchronization edges represent synchronization constraints, and are implemented using the UBS and BBS protocols introduced in the previous section. If there is an IPC edge as well as a synchronization edge between the same pair of actors, then the synchronization protocol is executed before the buffers corresponding to the IPC edge are accessed so as to ensure sender-receiver synchronization. On the other hand, if there is an IPC edge between two actors in the IPC graph, but there is no synchronization edge between the two, then no synchronization needs to be done before accessing the shared buffer. If there is a synchronization edge between two actors but no IPC edge, then no shared buffer is allocated between the two actors; only the corresponding synchronization protocol is invoked.

Thus, initially, the synchronization graph G_s is identical to G_{ipc}. Then we perform transformations on the synchronization graph in order to reduce synchronization costs. The cost measure and the transformations will be discussed in the following sections of this paper. All of these transformations must respect the synchronization constraints implied by G_{ipc}. If we ensure this, then we only need to implement the synchronization edges of the optimized synchronization graph. The following theorem is useful to formalize the concept of when the synchronization constraints represented by one synchronization graph G_1 imply the synchronization constraints of another graph G_2. This theorem provides a useful constraint for synchronization optimization, and it underlies the validity of the main techniques that we will present in this paper.

Theorem 1: The synchronization constraints in a synchronization graph $G_1 = (V, E_1)$ imply the synchronization constraints of the synchronization graph $G_2 = (V, E_2)$ if the following condition holds: $\forall \varepsilon \in E_2, \varepsilon \notin E_1, \rho_{G_1}(src(\varepsilon), snk(\varepsilon)) \le delay(\varepsilon)$, that is, if for each edge ε that is present in G_2 but not in G_1 there is a minimum delay path from $src(\varepsilon)$ to $snk(\varepsilon)$ in G_1 that has total delay of at most $delay(\varepsilon)$ (number of delays on edge ε).

(Note that since the vertex sets for the two graphs are identical, it is meaningful to refer to $src(\varepsilon)$ and $snk(\varepsilon)$ as being vertices of G_1 even though $\varepsilon \in E_2, \varepsilon \notin E_1$.)
First we prove the following lemma.

Lemma 4: If there is a path $p = (e_1, e_2, e_3, \ldots, e_n)$ in G_1, then

$$start(snk(e_n), k) \ge end(src(e_1), k - Delay(p)).$$

Proof of Lemma 4:
The following constraints hold along such a path p (as per (6))

$$start(snk(e_1), k) \geq end(src(e_1), k - delay(e_1)) . \tag{7}$$

Similarly,

$$start(snk(e_2), k) \geq end(src(e_2), k - delay(e_2)) .$$

Noting that $src(e_2)$ is the same as $snk(e_1)$, we get

$$start(snk(e_2), k) \geq end(snk(e_1), k - delay(e_2)) .$$

Causality implies $end(v, k) \geq start(v, k)$, so we get

$$start(snk(e_2), k) \geq start(snk(e_1), k - delay(e_2)) . \tag{8}$$

Substituting (7) in (8),

$$start(snk(e_2), k) \geq end(src(e_1), k - delay(e_2) - delay(e_1)) .$$

Continuing along p in this manner, it can easily be verified that

$$start(snk(e_n), k) \geq end(src(e_1), k - delay(e_n) - delay(e_{n-1}) - \ldots - delay(e_1)) ;$$

that is,

$$start((snk(e_n), k) \geq end(src(e_1), k - Delay(p))) . QED.$$

Proof of Theorem 1: If $\varepsilon \in E_2, \varepsilon \in E_1$, then the synchronization constraint due to the edge ε holds in both graphs. But for each $\varepsilon \in E_2, \varepsilon \notin E_1$ we need to show that the constraint due to ε:

$$start(snk(\varepsilon), k) > end(src(\varepsilon), k - delay(\varepsilon)) \tag{9}$$

holds in G_1 provided $\rho_{G_1}(src(\varepsilon), snk(\varepsilon)) \leq delay(\varepsilon)$, which implies there is at least one path $p = (e_1, e_2, e_3, \ldots, e_n)$ from $src(\varepsilon)$ to $snk(\varepsilon)$ in G_1 ($src(e_1) = src(\varepsilon)$ and $snk(e_n) = snk(\varepsilon)$) such that $Delay(p) \leq delay(\varepsilon)$.

From Lemma 4, existence of such a path p implies

$$start((snk(e_n), k) \geq end(src(e_1), k - Delay(p))) .$$

that is,

$$start((snk(\varepsilon), k) \geq end(src(\varepsilon), k - Delay(p))) \quad . \tag{10}$$

If $Delay(p) \leq delay(\varepsilon)$, then $end(src(\varepsilon), k - Delay(p)) \geq end(src(\varepsilon), k - delay(\varepsilon))$. Substituting this in (10) we get

$$start((snk(\varepsilon), k) \geq end(src(\varepsilon), k - delay(\varepsilon))) .$$

The above relation is identical to (9), and this proves the Theorem. *QED.*

The above theorem motivates the following definition.

Definition 3: If $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$ are synchronization graphs with the same vertex-set, we say that G_1 **preserves** G_2 if $\forall \varepsilon \in E_2, \varepsilon \notin E_1$, we have $\rho_{G_1}(src(\varepsilon), snk(\varepsilon)) \leq delay(\varepsilon)$.

Thus, Theorem 1 states that the synchronization constraints of (V, E_1) imply the synchronization constraints of (V, E_2) if (V, E_1) preserves (V, E_2).

Observation 1: Given an IPC graph G_{ipc}, and a synchronization graph G_s such that G_s preserves G_{ipc}, suppose that we implement the synchronizations corresponding to the synchronization edges of G_s. Then, the iteration period of the resulting system is determined by the maximum cycle mean of G_s. This is because the synchronization edges alone determine the interaction between processors; an IPC edge without synchronization does not constrain the execution of the corresponding processors in any way.

6.3 Computing Buffer Bounds from G_s and G_{ipc}

After all the optimizations are complete we have a final synchronization graph $G_s = (V, (E_{ipc} - F + F'))$ that preserves G_{ipc}. Since the synchronization edges in G_s are the ones that are finally implemented, it is advantageous to calculate the self-timed buffer bound B_{fb}

as a final step after all the transformations on G_s are complete, instead of using G_{ipc} itself to calculate these bounds. This is because addition of the edges F' may reduce these buffer bounds. It is easily verified that removal of the edges (F) cannot change the buffer bounds in (5) as long as the synchronizations in G_{ipc} are preserved. Thus, in the interest of obtaining minimum possible shared buffer sizes, we compute the bounds using the optimized synchronization graph. The following theorem tells us how to compute the self-timed buffer bounds from G_s.

Theorem 2: If G_s preserves G_{ipc} and the synchronization edges in G_s are implemented, then for each feedback IPC edge e in G_{ipc}, the self-timed buffer bound of e ($B_{fb}(e)$) — an upper bound on the number of data tokens that can ever be present on e — is given by:

$$B_{fb}(e) = \rho_{G_s}(snk(e), src(e)) + delay(e),$$

Proof: By Lemma 4, if there is a path p from $snk(e)$ to $src(e)$ in G_s, then

$$start(src(e), k) \geq end(snk(e), k - Delay(p)).$$

Taking p to be an arbitrary minimum-delay path from $snk(e)$ to $src(e)$ in G_s, we get

$$start(src(e), k) \geq end(snk(e), k - \rho_{G_s}(snk(e), src(e))).$$

That is, $src(e)$ cannot be more that $\rho_{G_s}(snk(e), src(e))$ iterations "ahead" of $snk(e)$. Thus there can never be more that $\rho_{G_s}(snk(e), src(e))$ tokens more than the initial number of tokens on e — $delay(e)$. Since the initial number of tokens on e was $delay(e)$, the size of the buffer corresponding to e is bounded above by $B_{fb}(e) = \rho_{G_s}(snk(e), src(e)) + delay(e)$. *QED.*

The quantities $\rho_{G_s}(snk(e), src(e))$ can be computed using Dijkstra's algorithm [7] to solve the all-pairs shortest path problem on the synchronization graph in time $O(|V|^3)$. Thus the problem of determining the $B_{fb}(e)$ values has complexity at most cubic in the size of the number of actors in the schedule.

7. Problem Statement

We refer to each access of the shared memory "synchronization variable" $sv(e)$ by $src(e)$ and $snk(e)$ as a **synchronization access**[1] to shared memory. If synchronization for e is implemented using UBS, then we see that on average, 4 synchronization accesses are required for e in each iteration period, while BBS implies 2 synchronization accesses per iteration period. We define the **synchronization cost** of a synchronization graph G_s to be the average number of synchronization accesses required per iteration period. Thus, if n_{ff} denotes the number of synchronization edges in G_s that are feedforward edges, and n_{fb} denotes the number of synchronization edges that are feedback edges, then the synchronization cost of G_s can be expressed as $(4n_{ff} + 2n_{fb})$. In the remainder of this paper we will develop techniques that apply the results and the analysis framework developed in Sections 4-6 to minimize the synchronization cost of a self-timed implementation of a DFG without sacrificing the integrity of any inter-processor data transfer or reducing the estimated throughput.

We will explore three mechanisms for reducing synchronization accesses. The first is the detection and removal of *redundant* synchronization edges, which are synchronization edges whose respective synchronization functions are subsumed by other synchronization edges, and thus need not be implemented explicitly. The second mechanism is the insertion of new synchronization edges in such a way that the number of original synchronization edges that become redundant exceeds the number of new edges added. This mechanism, which we call *resynchroni-*

1. Note that in our measure of the number of shared memory accesses required for synchronization, we neglect the accesses to shared memory that are performed while the sink actor is waiting for the required data to become available, or the source actor is waiting for an "empty slot" in the buffer. The number of accesses required to perform these "busy-wait" or "spin-lock" operations is dependent on the exact relative execution times of the actor invocations. Since in our problem context, this information is not generally available to us, we use the *best case* number of accesses — the number of shared memory accesses required for synchronization assuming that IPC data on an edge is always produced before the corresponding sink invocation attempts to execute — as an approximation.

zation, is explored in sections 9 and in the appendix. Finally, in Section 10, we examine the utility of adding additional synchronization edges to convert a synchronization graph that is not strongly connected into a strongly connected graph. Such a conversion allows us to implement all synchronization edges with BBS. We address optimization criteria in performing such a conversion, and we will show that the extra synchronization accesses required for such a conversion are always (at least) compensated by the number of synchronization accesses that are saved (by the UBSs that get converted to BBSs).

8. Removing Redundant Synchronizations

The first technique that we explore for reducing synchronization overhead is the removal of *redundant synchronization edges* of the synchronization graph. Formally, a synchronization edge is **redundant** in a synchronization graph G if its removal yields a synchronization graph that preserves G. Equivalently, from definition 3, a synchronization edge e is redundant in the synchronization graph G if there is a path $p \neq (e)$ in G directed from $src(e)$ to $snk(e)$ such that $Delay(p) \leq delay(e)$.

Thus, the synchronization function associated with a redundant synchronization edge "comes for free" as a by product of other synchronizations. Figure 6 shows an example of a redundant synchronization edge. Here, before executing actor D, the processor that executes

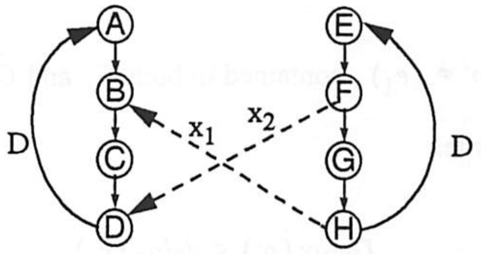


Figure 6. An example of a redundant synchronization edge.

$\{A, B, C, D\}$ does not need to synchronize with the processor that executes $\{E, F, G, H\}$ because due to the synchronization edge x_1, the corresponding invocation of F is guaranteed to complete before each invocation of D is begun. Thus, x_2 is redundant in Figure 6. It is easily verified that the path $p = ((F, G), (G, H), x_1, (B, C), (C, D))$ is directed from $src(x_2)$ to $snk(x_2)$, and has a path delay (zero) that is equal to the delay on x_2.

In this section we develop an efficient algorithm to optimally remove redundant synchronization edges from a synchronization graph.

8.1 The Independence of Redundant Synchronizations

The following theorem establishes that the order in which we remove redundant synchronization edges is not important, and thus, we need not implement synchronization for any of the redundant synchronization edges in a synchronization graph.

Theorem 3: Suppose that $G_s = (V, E)$ is a synchronization graph, e_1 and e_2 are distinct redundant synchronization edges in G_s, and $\tilde{G}_s = \left(V, E - \{e_1\}\right)$. Then e_2 is redundant in $\tilde{G}_s$.

Proof: Since e_2 is redundant in G_s, there is a path $p \neq (e_2)$ in G_s directed from $src(e_2)$ to $snk(e_2)$ such that

$$Delay(p) \le delay(e_2) . \tag{11}$$

Similarly, there is a path $p' \neq (e_1)$, contained in both G_s and $\tilde{G}_s$, that is directed from $src(e_1)$ to $snk(e_1)$, and that satisfies

$$Delay(p') \le delay(e_1) . \tag{12}$$

Now, if p does not contain e_1, then p exists in $\tilde{G}_s$, and we are done. Otherwise, let $p' = (x_1, x_2, \ldots, x_n)$; observe that p is of the form $p = (y_1, y_2, \ldots, y_{k-1}, e_1, y_k, y_{k+1}, \ldots, y_m)$; and define

$$p'' \equiv (y_1, y_2, \ldots, y_{k-1}, x_1, x_2, \ldots, x_n, y_k, y_{k+1}, \ldots, y_m) .$$

Clearly, p'' is a path from $src(e_2)$ to $snk(e_2)$ in $\tilde{G}_s$. Also,

$$\begin{aligned} Delay(p'') &= \sum delay(x_i) + \sum delay(y_i) \\ &= Delay(p') + (Delay(p) - delay(e_1)) \\ &\le Delay(p) && \text{(from (12))} \\ &\le delay(e_2) && \text{(from (11)).} \end{aligned}$$

QED.

Theorem 3 tells us that we can avoid implementing synchronization for *all* redundant synchronization edges since the "redundancies" are not interdependent. Thus, an optimal removal of redundant synchronizations can be obtained by applying a straightforward algorithm that successively tests the synchronization edges for redundancy in some arbitrary sequence, and since *shortest path* computation is a tractable problem, we can expect such a solution to be practical.

8.2 An Algorithm for Removing Redundant Synchronizations

Figure 7 presents an efficient algorithm, based on the ideas presented in the previous subsection, for optimal removal of redundant synchronization edges. In this algorithm, we first compute the path delay of a minimum-delay path from x to y for each ordered pair of vertices (x, y) ; here, we assign a path delay of ∞ whenever there is no path from x to y. This computation is equivalent to solving an instance of the well known *all points shortest paths problem* [7]. Then, we examine each synchronization edge e — in some arbitrary sequence — and determine whether or not there is a path from $src(e)$ to $snk(e)$ that does not contain e, and that has a path delay that does not exceed $delay(e)$. Now, at first, it may not be obvious that this check for redundancy is equivalent to the check that is performed by the *if* statement in *RemoveRedundantSynchs*: one might ask "What if e_0 satisfies the inequality in the *if* statement, but all of the minimum-delay paths from $snk(e_0)$ to $snk(e)$ contain e?" To see that the *if* statement is indeed equivalent to checking the redundancy of e, observe that if p is a path from $src(e)$ to $snk(e)$ that contains more than one edge and that contains e, then p must contain a cycle c such that c does not contain e; and since all cycles (from Lemma 1) must have positive path delay, the path delay of such a path p must exceed $delay(e)$. Thus, if e_0 satisfies the inequality in the *if* statement of *RemoveRedundantSynchs*, and p^* is a path from $snk(e_0)$ to $snk(e)$ such that $Delay(p^*) = \rho(snk(e_0), snk(e))$, then p^* cannot contain e.

```
Function RemoveRedundantSynchs
Input: A synchronization graph $G_s = (V, E)$ such that $I \subseteq E$ is the set of synchro-
nization edges.
Output: The synchronization graph $G_s^* = (V, (E - E_r))$, where $E_r$ is the set of
redundant synchronization edges in $G_s$.

1. Compute $\rho(x, y)$ for each ordered pair of vertices in $G_s$.
2. Initialize: $E_r = \emptyset$.
3. For each $e \in I$
     For each output edge $e_o$ of $src(e)$ except for $e$
         If $delay(e_o) + \rho(snk(e_o), snk(e)) \le delay(e)$
         Then
             $E_r = E_r \cup \{e\}$
             Break          /* exit the innermost enclosing For loop */
         End If
     End For
End For
4. Return $(V, (E - E_r))$.
```

Figure 7. An algorithm that optimally removes redundant synchronization edges.

From the definition of a redundant synchronization edge, it is easily verified that the removal of a redundant synchronization edge does not alter any of the minimum-delay path values (path delays). That is, given a redundant synchronization edge e_r in G_s, and two arbitrary vertices $x, y \in V$, if we let $\hat{G}_s = \left(V, \left(E - \{e_r\}\right)\right)$, then $\rho_{\hat{G}_s}(x, y) = \rho_{G_s}(x, y)$. Thus, none of the minimum-delay path values computed in Step 1 need to be recalculated after removing a redundant synchronization edge in Step 3.

Observe that the complexity of Function *RemoveRedundantSynchs* is dominated by Step 1 and Step 3. Since all edge delays are non-negative, we can repeatedly apply Dijkstra's algorithm (once for each vertex) to carry out Step 1 in $O\left(|V|^3\right)$ time; a modification of Dijkstra's algorithm can be used to reduce the complexity of Step 1 to $O\left(|V|^2\log_2(|V|) + |V||E|\right)$ [7]. In Step 3, $|E|$ is an upper bound for the number of synchronization edges, and in the worst case, each vertex has all members of V in its set of successors. Thus, the time complexity of Step 3 is $O(|V||E|)$, and if we use the modification to Dijkstra's algorithm mentioned above for Step 1, then the time complexity of *RemoveRedundantSynchs* is

$$O\left(|V|^2\log_2(|V|) + |V||E| + |V||E|\right) = O\left(|V|^2\log_2(|V|) + |V||E|\right).$$

8.3 Comparison with Shaffer's Approach

In [30], Shaffer presents an algorithm that minimizes the number of directed synchronizations in the self-timed execution of a dataflow graph under the (implicit) assumption that the execution of successive iterations of the dataflow graph are not allowed to overlap. In Shaffer's technique, a construction identical to our synchronization graph is used with the exception that there is no feedback edge that connects the last actor executed on a processor to the first actor executed on the same processor. Also, in Shaffer's construction, edges that have delay are ignored since only dependences within the same graph iteration are significant. Thus, Shaffer's synchronization graph can be assumed to be acyclic.

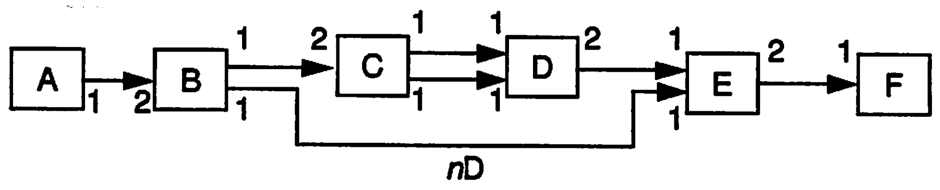
In the context of Shaffer's problem, a synchronization edge is redundant if and only if there is a path from the source of the edge to the sink — here we only need "reachability" information; no notion of path delay is required. As in the context of our problem, the removal of a redundant synchronization edge in Shaffer's synchronization graph cannot negate the redundancy of another redundant synchronization edge, and consequently, the order in which synchronization edges are tested for redundancy is not significant. Shaffer's algorithm begins by computing a boolean value $r(x, y)$ for each ordered pair of vertices (x, y) that is set to *true* if and only if there is a path directed from x to y. Then, the algorithm proceeds in a manner equivalent to Step 3 of *RemoveRedundantSynchs*, with the exception that the predicate of the *if* statement is changed from $(delay(e_o) + \rho(snk(e_o), snk(e)) \le delay(e))$ to $(r(snk(e_o), snk(e)))$. Thus, *RemoveRedundantSynchs* can be viewed as a direct extension of Shaffer's algorithm to handle pure self-timed, iterative execution of a DFG; Shaffer's algorithm accounts for self-timed execution only within a graph iteration, and in general, it can be applied to iterative dataflow programs only if all processors are forced to synchronize between graph iterations.

Shaffer states that the complexity of his algorithm is $O\left(|V|^3\right)$; however, the complexity can be improved (at least for sparse graphs) by using a more efficient technique to compute the function r. The function r in Shaffer's method can be computed in $O(|V||E|)$ time [7], and using this method, Shaffer's algorithm achieves a time complexity of $O(|V||E|)$. Thus, in exchange for its dependence on a less flexible execution model, Shaffer's solution, with appropriate choice of r, attains a slightly more favorable asymptotic complexity than our *RemoveRedundantSynchs*.

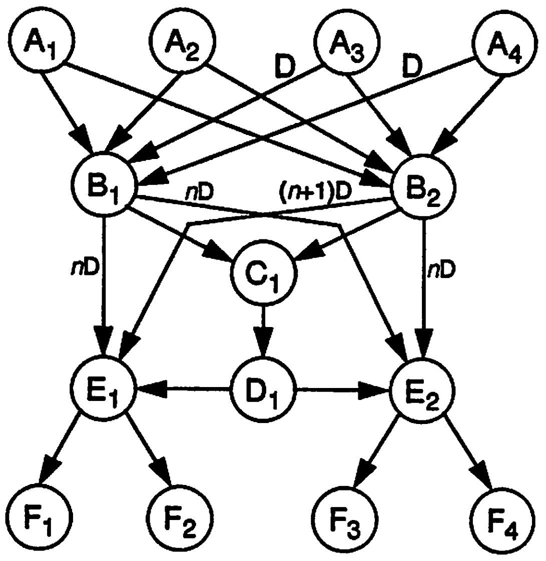
8.4 An Example

In this subsection, we illustrate the benefits of removing redundant synchronizations through a practical example. Figure 8(a) shows an abstraction of a three channel, multi-resolution quadrature mirror (QMF) filter bank, which has applications in signal compression [34]. This representation is based on the general (not homogeneous) SDF model, and accordingly, each edge is annotated with the number of tokens produced and consumed by its source and sink actors. For

(a)



(b)



Proc. 1 $A_1, A_2, B_1, C_1, D_1, E_1, F_1, F_2$

Proc. 2 $A_3, A_4, B_2, E_2, F_3, F_4$

(c)

(d)

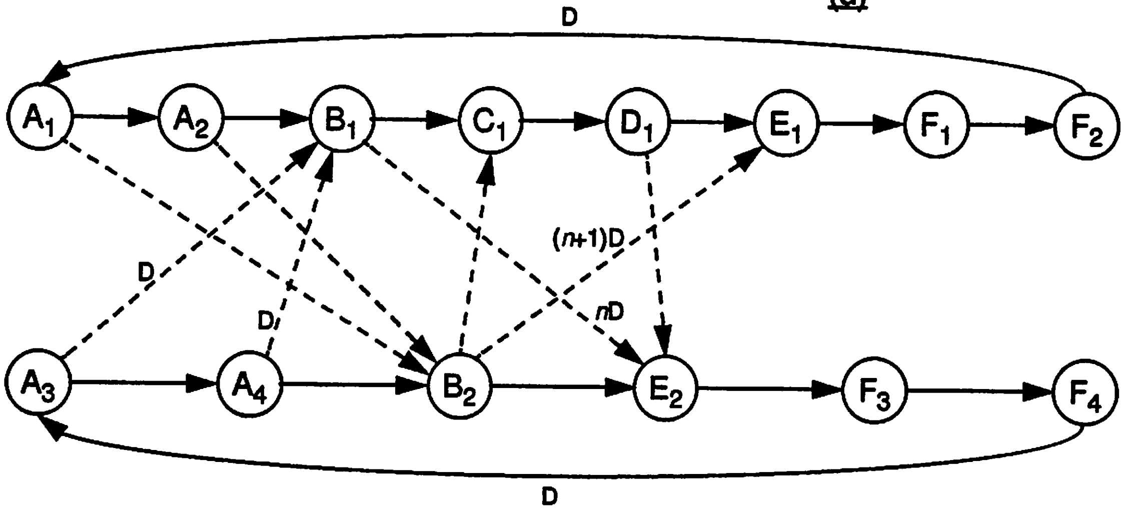


Figure 8. (a). A multi-resolution QMF filter bank used to illustrate the benefits of removing redundant synchronizations. (b). The precedence graph for (a). (c). A self-timed, two-processor, parallel schedule for (a). (d). The initial synchronization graph for (c).

clarity, the actors are drawn as boxes, rather than circles. Actors A and F represent the subsystems that, respectively, supply and consume data to/from the filter bank system; B and C each represents a parallel combination of decimating high and low pass FIR analysis filters; D and E represent the corresponding pairs of interpolating synthesis filters. The amount of delay on the edge directed from B to E is equal to the sum of the filter orders of C and D. For more details on the application represented by Figure 8(a), we refer the reader to [34].

To construct a periodic, parallel schedule we must first determine the number of times $\mathbf{q}(N)$ that each actor N must be invoked in the periodic schedule. Systematic techniques to compute these values are presented in [18]. Next, we must determine the precedence relationships between the actor invocations. In determining the exact precedence relationships, we must take into account the dependence of a given filter invocation on not only the invocation that produces the token that is "consumed" by the filter, but also on the invocations that produce the n preceding tokens, where n is the order of the filter. Such dependence can easily be evaluated with an additional dataflow parameter on each actor input that specifies the number of *past tokens* that are accessed [27][1]. Using this information, together with the invocation counts specified by $\mathbf{q}$, we obtain the precedence relationships specified by the graph of Figure 8(b), in which the ith invocation of actor N is labeled N_i, and each edge e specifies that invocation $snk(e)$ requires data produced by invocation $src(e)$ $delay(e)$ iteration periods after the iteration period in which the data is produced.

A self-timed schedule for Figure 8(b) that can be obtained from Hu's well-known list scheduling method [11] is specified in Figure 8(c), and the synchronization graph that corresponds to the IPC graph of Figure 8(b) and Figure 8(c) is shown in Figure 8(d). All of the dashed edges in Figure 8(d) are synchronization edges. If we apply Shaffer's method, which considers only those synchronization edges that do not have delay, we can eliminate the need for explicit

1. It should be noted that some SDF-based design environments choose to forego parallelization across multiple invocations of an actor in favor of simplified code generation and scheduling. For example, in the GRAPE system, this restriction has been justified on the grounds that it simplifies inter-processor data management, reduces code duplication, and allows the derivation of efficient scheduling algorithms that operate directly on general SDF graphs without requiring the use of the acyclic precedence graph (APG) [3].

synchronization along only one of the 8 synchronization edges — edge (A_1, B_2). In contrast, if we apply *RemoveRedundantSynchs*, we can detect the redundancy of (A_1, B_2) as well as four additional redundant synchronization edges — (A_3, B_1), (A_4, B_1), (B_2, E_1), and (B_1, E_2). Thus, *RemoveRedundantSynchs* reduces the number of synchronizations from 8 down to 3 — a reduction of 62%. Figure 9 shows the synchronization graph of Figure 8 (d) after all redundant synchronization edges are removed. It is easily verified that the synchronization edges that remain in this graph are not redundant; explicit synchronizations need only be implemented for these edges.

9. Resynchronization

It is sometimes possible to reduce the total number of irredundant synchronization edges by adding new synchronization edges to a synchronization graph. We refer to the process of adding one or more new synchronization edges and removing the redundant edges that result as *resynchronization* (defined more precisely below). Figure 10(a) illustrates this concept. Here, the dashed edges represent synchronization edges. Observe that if we insert the new synchronization

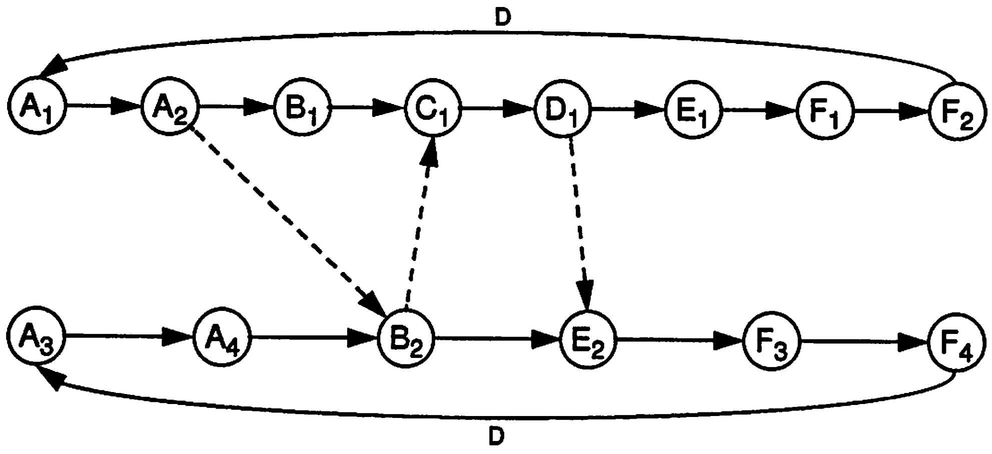


Figure 9. The synchronization graph of Figure 8(d) after all redundant synchronization edges are removed.

edge $d_0(C, H)$, then two of the original synchronization edges — (B, G) and (E, J) — become redundant. Since redundant synchronization edges can be removed from the synchronization graph to yield an equivalent synchronization graph, we see that the net effect of adding the synchronization edge $d_0(C, H)$ is to reduce the number of synchronization edges that need to be implemented by 1. In Figure 10(b), we show the synchronization graph that results from inserting the *resynchronization edge* $d_0(C, H)$ into Figure 10(a), and then removing the redundant synchronization edges that result.

Definition 4 gives a formal definition of resynchronization that we will use throughout the remainder of this paper. This considers resynchronization only "across" feedforward edges. Resynchronization that includes inserting edges into the SCCs is also possible; however, for our objectives, it must be verified that each new synchronization edge introduced in an SCC does not decrease the estimated throughput. To avoid this complication, which requires a check of significant complexity $(O(|V||E|\log_2(|V|))$, where (V, E) is the modified synchronization graph — this is using the Bellman Ford algorithm described in [17]) *for each* candidate resynchronization edge, we focus only on feedforward resynchronization in this paper.

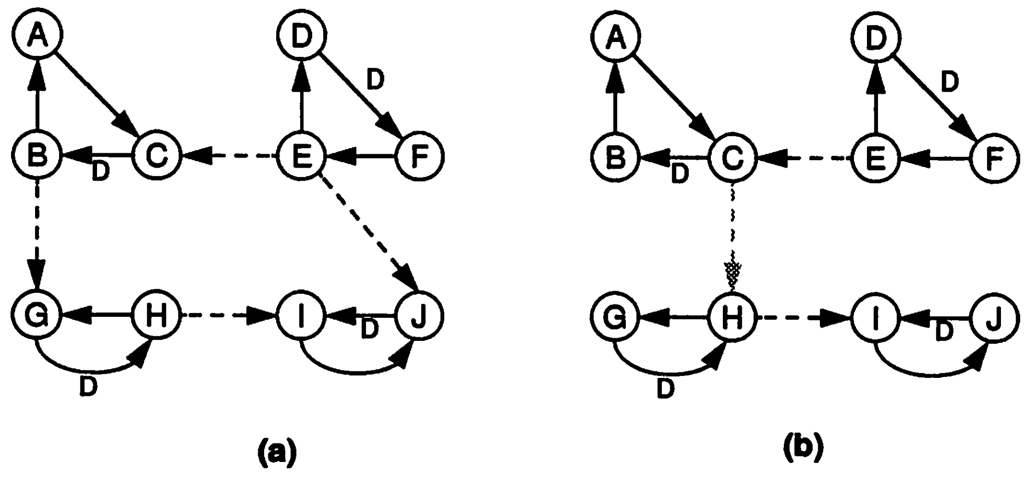


Figure 10. An example of resynchronization.

Definition 4: Given a synchronization graph $G = (V, E)$ consisting of feedforward edges $F \equiv \{e_1, e_2, \ldots, e_n\}$, a **resynchronization** of G is a finite set $F' \equiv \{e_1', e_2', \ldots, e_m'\}$ of edges that are not necessarily contained in E, but whose source and sink vertices are in V, such that (a). $e_1', e_2', \ldots, e_m'$ are feedforward edges in the DFG $G^* \equiv (V, ((E - F) + F'))$; and (b). G^* preserves G — that is, $\rho_{G^*}(src(e_i), snk(e_i)) \le delay(e_i)$ for all $i \in \{1, 2, \ldots, n\}$.

If we let G denote the graph in Figure 10, then the set of feedforward edges is $F = \{(B, G), (E, J)\}$; $F' = \{d_0(C, H)\}$ is a resynchronization of G; Figure 10(b) shows the DFG $G^* = (V, ((E - F) + F'))$; and from Figure 10(b), it is easily verified that F, F', and G^* satisfy conditions (a) and (b) of Definition 4.

We refer to the problem of finding a resynchronization with the fewest number of elements as the **resynchronization problem**. In the appendix, we formally show that the resynchronization problem is NP-hard, and in this section, we explain the intuition behind this result. To establish the NP-hardness of the resynchronization problem, we examine a special case of the problem that occurs when there are exactly two SCCs, which we call the **pairwise resynchronization problem**, and we derive a polynomial-time reduction from the classic *set covering problem* [7], a well-known NP-hard problem, to the pairwise resynchronization problem. In the set covering problem, one is given a finite set X and a family T of subsets of X, and asked to find a minimal (fewest number of members) subfamily $T_s \subseteq T$ such that $\bigcup_{t \in T_s} t = X$. A subfamily of T is said to *cover* X if each member of X is contained in some member of the subfamily. Thus, the set covering problem is the problem of finding a minimal cover.

Although the correspondence that we establish between the resynchronization problem and set covering shows that the resynchronization problem probably cannot be attacked optimally with a polynomial-time algorithm, we will show that the correspondence allows any heuristic for set covering to be adapted easily into a heuristic for the pairwise resynchronization problem, and applying such a heuristic to each pair of SCCs in a general synchronization graph yields a heuristic for the general (not just pairwise) resynchronization problem. This is fortunate since the set covering problem has been studied in great depth, and efficient heuristic methods have been devised [7].

The following definition facilitates the developments of this section and the appendix.

Definition 5: Given a synchronization graph G, let (x_1, x_2) and (y_1, y_2) be two ordered pairs of vertices in G. We say that (y_1, y_2) **subsumes** (x_1, x_2) in G if $\rho(x_1, y_1) = \rho(y_2, x_2) = 0$. We may omit the qualification "in G" if the graph in question is understood from context.

Intuitively, every ordered pair of vertices subsumes itself, and if (x_1, x_2) and (y_1, y_2) are distinct, then (y_1, y_2) subsumes (x_1, x_2) if a zero-delay synchronization edge directed from y_1 to y_2 would make a synchronization edge (regardless of its delay) directed from x_1 to x_2 redundant.

The following fact is easily verified from Definitions 4 and 5.

Fact 1: Suppose that G is a synchronization graph that contains exactly two SCCs, F is the set of feedforward edges in G, and F' is a resynchronization of G. Then for each $e \in F$, there exists $e' \in F'$ such that $(src(e'), snk(e'))$ subsumes $(src(e), snk(e))$ in G.

An intuitive correspondence between the pairwise resynchronization problem and the set covering problem can be derived from Fact 1. Suppose that G is a synchronization graph with exactly two SCCs C_1 and C_2 such that each feedforward edge is directed from a member of C_1 to a member of C_2. We start by viewing the set F of feedforward edges in G as the finite set that we wish to cover, and with each member p of $\{(x, y) \mid (x \in C_1, y \in C_2)\}$, we associate the subset of F defined by $\chi(p) \equiv \{e \in F \mid (p\ subsumes\ (src(e), snk(e)))\}$. Thus, $\chi(p)$ is the set of feedforward edges of G whose corresponding synchronizations can be eliminated if we implement a zero-delay synchronization edge directed from the first vertex of the ordered pair p to the second vertex of p. Clearly then, $\{e_1', e_2', \ldots, e_n'\}$ is a resynchronization if and only if each $e \in F$ is contained in at least one $\chi((src(e_i'), snk(e_i')))$ — that is, if and only if $\{\chi((src(e_i'), snk(e_i'))) \mid 1 \le i \le n\}$ covers F. Thus, solving the pairwise resynchronization problem for G is equivalent to finding a minimal cover for F given the family of subsets $\{\chi(x, y) \mid (x \in C_1, y \in C_2)\}$.

Figure 11 helps to illustrate this intuition and our method (defined formally in the appendix) for converting an instance of the set covering problem to an instance of pairwise resynchronization. Suppose that we are given the set $X = \{x_1, x_2, x_3, x_4\}$, and the family of subsets $T = \{t_1, t_2, t_3\}$, where $t_1 = \{x_1, x_3\}$, $t_2 = \{x_1, x_2\}$, and $t_3 = \{x_2, x_4\}$. To construct an instance of the pairwise resynchronization problem, we first create two vertices and an edge directed between these vertices *for each* member of X; we label each of the edges created in this step with the corresponding member of X. Then for each $t \in T$, we create two vertices $vsrc(t)$ and $vsnk(t)$. Next, for each relation $x_i \in t_j$ (there are six such relations in this example), we create two zero-delay edges — one directed from the source of the edge corresponding x_i to $vsrc(t_j)$, and another directed from $vsnk(t_j)$ to the sink of the edge corresponding to x_i. This last step has the effect of making each pair $(vsrc(t_i), vsnk(t_i))$ preserve exactly those edges that correspond to members of t_i; in other words, after this construction, $\chi((vsrc(t_i), vsnk(t_i))) = t_i$, for each i. Finally, for each edge created in the previous step, we create a corresponding feedback edge oriented in the opposite direction, and having a unit delay[1].

Figure 11 shows the graph that results from this construction process. Observe that the graph contains two SCCs — $\left(\{src(x_i)\} \cup \{vsrc(t_i)\}\right)$ and $\left(\{snk(x_i)\} \cup \{vsnk(t_i)\}\right)$

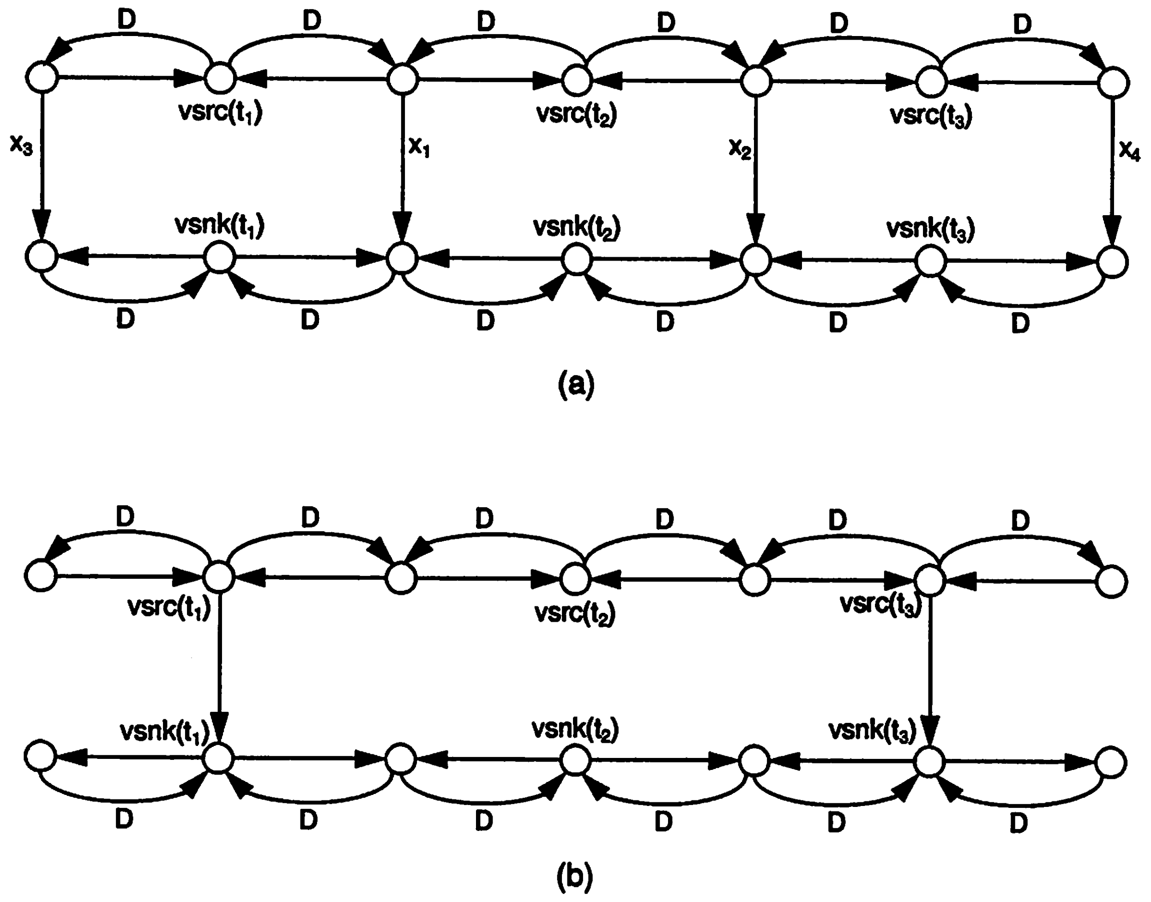


Figure 11. (a). An instance of the pairwise resynchronization problem that is derived from an instance of the set covering problem.
(b). The DFG that results from a solution to this instance.

1. In general, these edges will not be sufficient to ensure that the resulting graph has exactly two SCCs. In the appendix, we will show that for our reduction of set covering to pairwise resynchronization, we can assume without loss of generality that the family T is such that the construction outlined here guarantees a graph with exactly two SCCs.

— and that the set of feedforward edges is the set of edges that correspond to members of X. Now, recall that a major correspondence between the given instance of set covering and the instance of pairwise resynchronization defined by Figure 11(a) is that

$\chi((vsrc(t_i), vsnk(t_i))) = t_i$, for each i. Thus, if we can find a minimal resynchronization of Figure 11(a) such that each edge in this resynchronization is directed from some $vsrc(t_k)$ to the corresponding $vsnk(t_k)$, then the associated t_k's form a minimum cover of X. For example, it is easy, albeit tedious, to verify that the resynchronization illustrated in Figure 11(b),

$\{d_0(vsrc(t_1), vsnk(t_1)), d_0(vsrc(t_3), vsnk(t_3))\}$, is a minimal resynchronization of Figure 11(a), and from this, we can conclude that $\{t_1, t_3\}$ is a minimal cover for X. From inspection of the given sets X and T, it is easily verified that this conclusion is correct.

This example illustrates how an instance of pairwise resynchronization can be constructed (in polynomial time) from an instance of set covering, and how a solution to this instance of pairwise resynchronization can easily be converted into a solution of the set covering instance. Our proof of the NP-hardness of pairwise resynchronization, presented in the appendix, is a formalized generalization of this example. We summarize with the following theorem.

Theorem 4: The pairwise resynchronization problem is NP-hard, and thus, the resynchronization problem is NP-hard.

Proof: A formal proof is given in the appendix.

Two natural questions that arise when studying the example of Figure 11 are "How do we know that a minimal resynchronization exists such that each edge is directed from a $vsrc(t_k)$ to the corresponding $vsnk(t_k)$?" and "If such a minimal resynchronization exists, how can we obtain one efficiently from an arbitrary minimal resynchronization?" In the appendix, we will show that such a minimal synchronization always exists, and that we can always derive (in polynomial time) such a minimal resynchronization from an arbitrary minimal synchronization. The key here is that if e' is a member of a minimal resynchronization R, then there is always a member $p^* = (x^*, y^*)$ of $\{(vsrc(t_i), vsnk(t_i))\}$ such that $\chi(snk(e'), snk(e')) \subseteq \chi(p^*)$, and thus, replacing e' with $d_0(x^*, y^*)$ in R yields a minimal resynchronization.

We have pointed out that the correspondence we have established between set-covering and pairwise resynchronization allows us to adapt any heuristic for set-covering into a heuristic for pairwise resynchronization. Furthermore applying such a heuristic for pairwise resynchronization to each pair of SCCs in a general synchronization graph gives a heuristic for the general resynchronization problem. Figure 12 below shows how any algorithm *Cover* that solves the set covering problem can be applied to derive a heuristic algorithm for resynchronization.

10. Making the Synchronization Graph Strongly Connected

In Section 6, we defined two different synchronization protocols — bounded buffer synchronization (BBS), which has a cost of 2 synchronization accesses per iteration period, and can be used whenever the associated edge is contained in a strongly connected component of the synchronization graph; and unbounded buffer synchronization (UBS), which has a cost of 4 synchronization accesses per iteration period. We pay the increased overhead of UBS whenever the associated edge is a feedforward edge of the synchronization graph.

One alternative to implementing UBS for a feedforward edge e is to add synchronization edges to the synchronization graph so that e becomes encapsulated in a strongly connected component; such a transformation would allow e to be implemented with BBS. However, extra synchronization accesses will be required to implement the new synchronization edges that are inserted. In this section, we show that by adding synchronization edges through a certain simple procedure, the synchronization graph can be transformed into a strongly connected graph in such a way that the overhead of implementing the extra synchronization edges is always at least compensated by the savings attained by being able to avoid the use of UBS. That is, the total number

```
Function Resynchronize
Input: A synchronization graph $G = (V, E)$.
Output: A synchronization graph $\tilde{G}$ that preserves $G$.

$\tilde{E} = E$
Compute $\rho_G(x, y)$ for each ordered pair of vertices in $G$.  /* used in Pairwise */
For each SCC $C_a$ of $G$
    For each SCC $C_d$ of $G$
        If $C_a$ is a predecessor SCC of $C_d$ Then
            Compute $E_f = \{e \in E \mid (src(e) \in C_a) \text{ and } (snk(e) \in C_d)\}$
            $F = Pairwise(subgraph(C_a), subgraph(C_d), E_f)$
            $\tilde{E} = ((\tilde{E} - E_f) \cup F)$
        End If
    End For
End For
Return $(V, \tilde{E})$
```

```
Function Pairwise($G_1$, $G_2$, $F$)
Input: Two strongly connected synchronization graphs $G_1$ and $G_2$, and a set $F$ of edges whose source vertices are all in $G_1$ and whose sink vertices are all in $G_2$.
Output: A resynchronization $F'$.

For each vertex $u$ in $G_1$
    For each vertex $v$ in $G_2$
        $\chi((u, v)) = \{e \in F \mid (\rho_G(src(e), u) = 0) \text{ and } (\rho_G(v, snk(e)) = 0)\}$
    End For
End For
$T = \{\chi((u, v)) \mid (u \text{ is in } G_1 \text{ and } v \text{ is in } G_2)\}$
$\Xi = Cover(F, T)$
Return $\{d_0(u, v) \mid \chi((u, v)) \in \Xi\}$
```

Figure 12. An algorithm for resynchronization that is derived from an arbitrary algorithm *Cover* for the set covering problem

of synchronization accesses required (per iteration period) for the transformed graph is less than or equal to the number of synchronization accesses required for the original synchronization graph. Through a practical example, we show that this transformation can significantly reduce the number of required synchronization accesses. Also, we develop a technique to compute the delay that should be added to each of the new edges added in the conversion to a strongly connected graph. This technique computes the delays in such a way that the estimated throughput of the IPC graph is preserved with minimal increase in the shared memory storage cost required to implement the IPC edges.

10.1 Adding Edges to the Synchronization Graph

Figure 13 presents our algorithm for transforming a synchronization graph that is not strongly connected into a strongly connected graph. This algorithm simply "chains together" the source SCCs, and similarly, chains together the sink SCCs. The construction is completed by connecting the first SCC of the "source chain" to the last SCC of the sink chain with an edge that we call the **sink-source edge**. From each source or sink SCC, the algorithm selects a vertex that has minimum execution time to be the chain "link" corresponding to that SCC. Minimum execution time vertices are chosen in an attempt to minimize the amount of delay that must be inserted on the new edges to preserve the estimated throughput of the original graph. In Subsection 10.2, We discuss in detail the selection of delays for the edges introduced by *Convert-to-SC-graph*.

It is easily verified that algorithm *Convert-to-SC-graph* always produces a strongly connected graph, and that a conversion to a strongly connected graph cannot be attained by adding fewer edges than the number of edges added by *Convert-to-SC-graph*. Figure 14 illustrates a possible solution obtained by algorithm *Convert-to-SC-graph*. Here, the black dashed edges are the synchronization edges contained in the original synchronization graph, and the grey dashed edges are the edges that are added by *Convert-to-SC-graph*. The dashed edge labeled e_s is the sink-source edge.

Assuming the synchronization graph is connected, the number of feedforward edges n_f must satisfy $(n_f > n_c - 2)$, where n_c is the number of SCCs. This follows from the fundamental graph theoretic fact that in a connected graph (V^*, E^*), $|E^*|$ must exceed $(|V^*| - 2)$. Now, it is easily verified that the number of new edges introduced by *Convert-to-SC-graph* is equal to $(n_{src} + n_{snk} - 1)$, where n_{src} is the number of source SCCs, and n_{snk} is the number of sink SCCs. Thus, the number of synchronization accesses per iteration period, S_+, that is required to implement the edges introduced by *Convert-to-SC-graph* is $(2 \times (n_{src} + n_{snk} - 1))$, while the number of synchronization accesses, S_-, eliminated by *Convert-to-SC-graph* (by allowing the feedforward edges of the original synchronization graph to be implemented with BBS rather than UBS) equals $2n_f$. It follows that the net change $(S_+ - S_-)$ in the number of synchronization

```
Function Convert-to-SC-graph
Input: A synchronization graph $G$ that is not strongly connected.
Output: A strongly connected graph obtained by adding edges between the
SCCs of $G$.

1. Generate an ordering $C_1, C_2, \ldots, C_m$ of the source SCCs of $G$, and similarly,
generate an ordering $D_1, D_2, \ldots, D_n$ of the sink SCCs of $G$.
2. Select a vertex $v_1 \in C_1$ that minimizes $t(*)$ over $C_1$.
3. For $i = 2, 3 \ldots, m$
    • Select a vertex $v_i \in C_i$ that minimizes $t(*)$ over $C_i$.
    • Instantiate the edge $d_0(v_{i-1}, v_i)$.
End For
4. Select a vertex $w_1 \in D_1$ that minimizes $t(*)$ over $D_1$.
5. For $i = 2, 3 \ldots, n$
    • Select a vertex $w_i \in D_i$ that minimizes $t(*)$ over $D_i$.
    • Instantiate the edge $d_0(w_{i-1}, w_i)$.
End For
6. Instantiate the edge $d_0(w_m, v_1)$.
```

Figure 13. An algorithm for converting a synchronization graph that is not strongly connected into a strongly connected graph.

accesses satisfies

$$(S_+ - S_-) = 2(n_{src} + n_{snk} - 1) - 2n_f \le 2(n_c - 1 - n_f) \le 2(n_c - 1 - (n_c - 1)) ,$$

and thus, $(S_+ - S_-) \le 0$. We have established the following result.

Theorem 5: Suppose that G is a synchronization graph, and $\hat{G}$ is the graph that results from applying algorithm *Convert-to-SC-graph* to G. Then the synchronization cost of $\hat{G}$ is less than or equal to the synchronization cost of G.

For example, without the edges added by *Convert-to-SC-graph* (the dashed grey edges) in Figure 14, there are 6 feedforward edges, which require 24 synchronization accesses per iteration period to implement. The addition of the 4 dashed edges requires 8 synchronization accesses to implement these new edges, but allows us to use UBS for the original feedforward edges, which leads to a savings of 12 synchronization accesses for the original feedforward edges. Thus, the net effect achieved by *Convert-to-SC-graph* in this example is a reduction of the total number of synchronization accesses by $(12 - 8) = 4$. As another example, consider Figure 15, which shows the synchronization graph topology (after redundant synchronization edges are removed) that

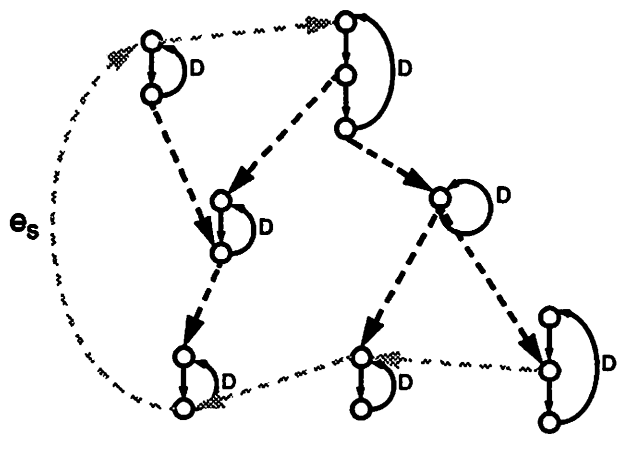


Figure 14. An illustration of a possible solution obtained by algorithm *Convert-to-SC-graph.*

results from a four-processor schedule of a synthesizer for plucked-string musical instruments in seven voices based on the Karplus-Strong technique. This graph contains $n_i = 6$ synchronization edges (the dashed edges), all of which are feedforward edges, so the synchronization cost is $4n_i = 24$ synchronization access per iteration period. Since the graph has one source SCC and one sink SCC, only one edge is added by *Convert-to-SC-graph*, and adding this edge reduces the synchronization cost to $2n_i + 2 = 14$ — a 42% savings.

Figure 16 shows the topology of a possible solution computed by *Convert-to-SC-graph* on

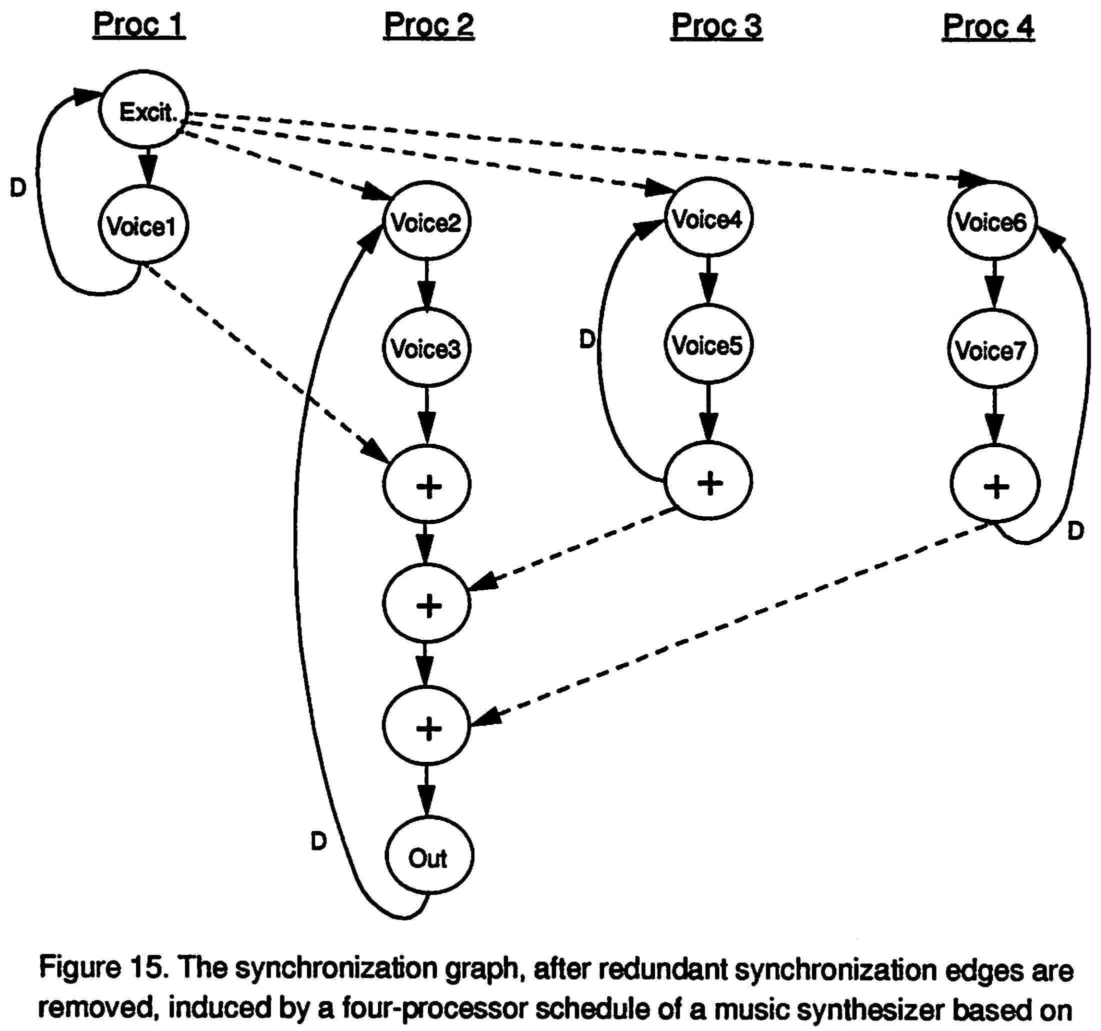


Figure 15. The synchronization graph, after redundant synchronization edges are removed, induced by a four-processor schedule of a music synthesizer based on the Karplus-Strong algorithm.

this example. Here, the dashed edges represent the synchronization edges in the synchronization graph returned by *Convert-to-SC-graph*. The actual solution computed by a given implementation of *Convert-to-SC-graph* will depend on exactly how the ordering in Step 1 is constructed, and thus may differ from the one shown here. However, any solution for Figure 15 generated from an implementation of *Convert-to-SC-graph* will have six synchronization edges in the result, as shown in Figure 16.

10.2 Insertion of Delays

One issue remains to be addressed in the conversion of a synchronization graph G_s into a strongly connected graph $\hat{G}_s$ — the proper insertion of delays so that $\hat{G}_s$ is not deadlocked, and does not have lower estimated throughput than G_s. The potential for deadlock and reduced estimated throughput arise because the conversion to a strongly connected graph necessarily must introduce one or more new fundamental cycles. In general, a new cycle may be delay-free, or its

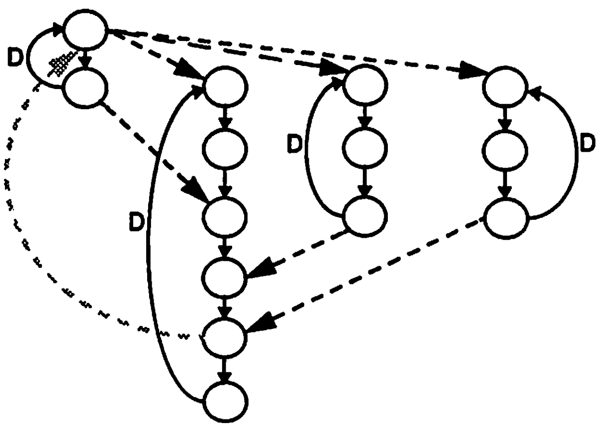


Figure 16. A possible solution obtained by applying *Convert-to-SC-graph* to the example of Figure 15.

cycle mean may exceed that of the critical cycle in G_s. Thus, we may have to insert delays on the edges added by *Convert-to-SC-graph*. The location (edge) and magnitude of the delays that we add are significant since (from Theorem 2) they affect the self-timed buffer bounds of the IPC edges. Since the self-timed buffer bounds determine the amount of memory that we allocate for the corresponding buffers, it is desirable to prevent deadlock and decrease in estimated throughput in such a way that we minimize the sum of the self-timed buffer bounds over all IPC edges. In this subsection, we present a simple and efficient algorithm for addressing this goal. Our algorithm produces an optimal result if G_s has only one source SCC or only one sink SCC; in other cases, the algorithm must be viewed as a heuristic. In practice, the assumptions under which we can expect an optimal result are frequently satisfied.

For simplicity in explaining our optimality result, we first specify a restricted version of the algorithm that assumes only one sink SCC. After explaining the optimality of this restricted algorithm, we discuss how it can be modified to yield an optimal algorithm for the general single-source-SCC case, and finally, we discuss how it can be extended to provide a heuristic for arbitrary synchronization graphs.

We will use the following notation in the remainder of this section: if $G = (V, E)$ is a DFG; $(e_0, e_1, \ldots, e_{n-1})$ is a sequence of distinct members of E; and $\Delta_0, \Delta_1, \ldots, \Delta_{n-1} \in \{0, 1, \ldots, \infty\}$, then $G[e_0 \rightarrow \Delta_0, \ldots, e_{n-1} \rightarrow \Delta_{n-1}]$ denotes the DFG $\left(V, \left(\left(E - \{e_0, e_1, \ldots, e_{n-1}\}\right) \cup \{e_0', e_1', \ldots, e_{n-1}'\}\right)\right)$, where each e_i' is defined by $src(e_i') = src(e_i)$, $snk(e_i') = snk(e_i)$, and $delay(e_i') = \Delta_i$. Thus, $G[e_0 \rightarrow \Delta_0, \ldots, e_{n-1} \rightarrow \Delta_{n-1}]$ is simply the DFG that results from "changing the delay" on each e_i to the corresponding new delay value Δ_i.

Definition 6: Suppose that G is a synchronization graph that preserves G_{ipc}. An **IPC sink-source path** in G is a minimum-delay path in G directed from $snk(e)$ to $src(e)$, where e is an IPC edge (in G_{ipc}). The existence of such a path is guaranteed by Definition 3.

Figure 17 outlines the restricted version of our algorithm that applies when the synchronization graph G_s has exactly one source SCC. Here, *BellmanFord* is assumed to be an algorithm that takes a synchronization graph Z as input, and applies the Bellman-Ford algorithm discussed in pp. 94-97 of [17] to return the cycle mean of the critical cycle in Z; if one or more cycles exist that have zero path delay, then *BellmanFord* returns ∞.

Algorithm *DetermineDelays* is based on the observations that the set of IPC sink-source paths introduced by *Convert-to-SC-graph* can be partitioned into m nonempty subsets $P_0, P_1, \ldots, P_{m-1}$ such that each member of P_i contains $e_0, e_1, \ldots, e_i$[1] and contains no other members of $\{e_0, e_1, \ldots, e_{m-1}\}$, and similarly, the set of fundamental cycles introduced by *DetermineDelays* can be partitioned into $W_0, W_1, \ldots, W_{m-1}$ such that each member of W_i contains $e_0, e_1, \ldots, e_i$ and contains no other members of $\{e_0, e_1, \ldots, e_{m-1}\}$.

By construction, a nonzero delay on any of the edges $e_0, e_1, \ldots, e_i$ "contributes to reducing the cycle means of all members of W_i". Algorithm *DetermineDelays* starts (iteration $i = 0$ of the *For* loop) by determining the minimum delay δ_0 on e_0 that is required to ensure that none of the cycles in W_0 has a cycle mean that exceeds the maximum cycle mean λ_{max} of G_s. Then (in iteration $i = 1$) the algorithm determines the minimum delay δ_1 on e_1 that is required to guarantee that no member of W_1 has a cycle mean that exceeds λ_{max}, assuming that $delay(e_0) = \delta_0$.

Now, if $delay(e_0) = \delta_0$, $delay(e_1) = \delta_1$, and $\delta_1 > 0$, then for any positive integer $k \le \delta_1$, k units of delay can be "transferred from e_1 to e_0" without violating the property that no member of $(W_0 \cup W_1)$ contains a cycle whose cycle mean exceeds λ_{max}. However, such a

1. See Figure 17 for the specification of what the e_is represent.

Function *DetermineDelays*

Input: Synchronization graphs $G_s = (V, E)$ and $\hat{G}_s$, where $\hat{G}_s$ is the graph computed by *Convert-to-SC-graph* when applied to G_s. The ordering of source SCCs generated in Step 2 of *Convert-to-SC-graph* is denoted $C_1, C_2, \ldots, C_m$. For $i = 1, 2, \ldots m - 1$, e_i denotes the edge instantiated by *Convert-to-SC-graph* from a vertex in C_i to a vertex in C_{i+1}. The sink-source edge instantiated by *Convert-to-SC-graph* is denoted e_0.

Output: Non-negative integers $d_o, d_1, \ldots, d_{m-1}$ such that the estimated throughput of $\hat{G}_s[e_0 \to d_0, \ldots, e_{m-1} \to d_{m-1}]$ equals the estimated throughput of G_s.

```
$X_0 = \hat{G}_s[e_0 \to \infty, \ldots, e_{m-1} \to \infty]$
$\lambda_{max}$= BellmanFord($X_0$)          /* compute the max. cycle mean of $G_s$ */
$d_{ub} = \left\lceil \left( \sum_{x \in V} t(x) \right) / \lambda_{max} \right\rceil$          /* an upper bound on the delay required for any $e_i$ */
For $i = 0, 1, \ldots, m - 1$
    $\delta_i = MinDelay(X_i, e_i, \lambda_{max}, d_{ub})$
    $X_{i+1} = X_i[e_i \to \delta_i]$          /* fix the delay on $e_i$ to be $\delta_i$ */
End For
Return $\delta_o, \delta_1, \ldots, \delta_{m-1}$.
```

Function *MinDelay*(X, e, λ, B)

Input: A synchronization graph X, an edge e in X, a positive real number λ, and a positive integer B.

Output: Assuming $X[e \to B]$ has estimated throughput no less than λ^{-1}, determine the minimum $d \in \{0, 1, \ldots, B\}$ such that the estimated throughput of $X[e \to d]$ is no less than λ^{-1}.

Perform a binary search in the range $[0, 1, \ldots, B]$ to find the minimum value of $r \in \{0, 1, \ldots, B\}$ such that *BellmanFord*($X[e \to r]$) returns a value less than or equal to λ. Return this minimum value of r.

Figure 17. An algorithm for determining the delays on the edges introduced by algorithm *Convert-to-SC-graph*. This algorithm assumes that the original synchronization graph (G_s) has only one sink SCC.

transformation increases the path delay of each member of P_0 while leaving the path delay of each member of P_1 unchanged, and thus, from Theorem 2, such a transformation cannot reduce the self-timed buffer bound of any IPC edge. Furthermore, apart from transferring delay from e_1 to e_0, the only other change that can be made to $delay(e_0)$ or $delay(e_1)$ — without introducing a member of $(W_0 \cup W_1)$ whose cycle mean exceeds λ_{max} — is to increase one or both of these values by some positive integer amount(s). Clearly, such a change cannot reduce the self-timed buffer bound on any IPC edge.

Thus, we see that the values δ_0 and δ_1 computed by *DetermineDelays* for $delay(e_0)$ and $delay(e_1)$, respectively, optimally ensure that no member of $(W_0 \cup W_1)$ has a cycle mean that exceeds λ_{max}. After computing these values, *DetermineDelays* computes the minimum delay δ_2 on e_2 that is required for all members of W_2 to have cycle means less than or equal to λ_{max}, assuming that $delay(e_0) = \delta_0$ and $delay(e_1) = \delta_1$. Given the "configuration" $(delay(e_0) = \delta_0, delay(e_1) = \delta_1, delay(e_2) = \delta_2)$, transferring delay from e_2 to e_1 increases the path delay of all members of P_1, while leaving the path delay of each member of $(P_0 \cup P_2)$ unchanged; and transferring delay from e_2 to e_0 increases the path delay across $(P_0 \cup P_1)$, while leaving the path delay across P_2 unchanged. Thus, by an argument similar to that given to establish the optimality of (δ_0, δ_1) with respect to $(W_0 \cup W_1)$, we can deduce that (1). The values computed by *DetermineDelays* for the delays on e_0, e_1, e_2 guarantee that no member of $(W_0 \cup W_1 \cup W_2)$ has a cycle mean that exceeds λ_{max}; and (2). For any other assignment of delays $(\delta_0', \delta_1', \delta_2')$ to (e_0, e_1, e_2) that preserves the estimated throughput across $(W_0 \cup W_1 \cup W_2)$, and for any IPC edge e such that an IPC sink-source path of e is contained in $(P_0 \cup P_1 \cup P_2)$, the self-timed buffer bound of e under the assignment $(\delta_0', \delta_1', \delta_2')$ is greater than or equal to self-timed buffer bound of e under the assignment $(\delta_0, \delta_1, \delta_2)$ computed by iterations $i = 0, 1, 2$ of *DetermineDelays*.

After extending this analysis successively to each of the remaining iterations $i = 3, 4, \ldots, m-1$ of the *for* loop in *DetermineDelays*, we arrive at the following result.

Theorem 6: Suppose that G_s is a synchronization graph that has exactly one sink SCC; let $\hat{G}_s$ and $(e_0, e_1, \ldots, e_{m-1})$ be as in Figure 17; let $(d_0, d_1, \ldots, d_{m-1})$ be the result of applying *DetermineDelays* to G_s and $\hat{G}_s$; and let $(d_0', d_1', \ldots, d_{m-1}')$ be any sequence of m non-negative integers such that $\hat{G}_s[e_0 \rightarrow d_0', \ldots, e_{m-1} \rightarrow d_{m-1}']$ has the same estimated throughput as G_s. Then $\Phi\left(\hat{G}_s[e_0 \rightarrow d_0', \ldots, e_{m-1} \rightarrow d_{m-1}']\right) \geq \Phi\left(\hat{G}_s[e_0 \rightarrow d_0, \ldots, e_{m-1} \rightarrow d_{m-1}]\right)$, where $\Phi(X)$ denotes the sum of the self-timed buffer bounds over all IPC edges in G_{ipc} induced by the synchronization graph X.

Figure 18 illustrates a solution obtained from *DetermineDelays*. Here we assume that $t(v) = 1$, for each vertex v, and we assume that the set of IPC edges is $\{e_a, e_b\}$ (for clarity, we are assuming in this example that the IPC edges are present in the given synchronization graph). The grey dashed edges are the edges added by *Convert-to-SC-graph*. We see that λ_{max} is determined by the cycle in the sink SCC of the original graph, and inspection of this cycle yields

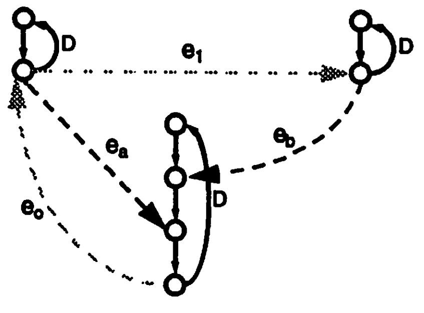


Figure 18. An example used to illustrate a solution obtained by algorithm *DetermineDelays*.

$\lambda_{max} = 4$. Also, we see that the set W_0 — the set of fundamental cycles that contain e_0, and do not contain e_1 — consists of a single cycle c_0 that contains three edges. By inspection of this cycle, we see that the minimum delay on e_0 required to guarantee that its cycle mean does not exceed λ_{max} is 1. Thus, the $i = 0$ iteration of the *For* loop in *DetermineDelays* computes $\delta_0 = 1$. Next, we see that W_1 consists of a single cycle that contains five edges, and we see that two delays must be present on this cycle for its cycle mean to be less than or equal to λ_{max}. Since one delay has been placed on e_0, *DetermineDelays* computes $\delta_1 = 1$ in the $i = 1$ iteration of the *For* loop. Thus, the solution determined by *DetermineDelays* for Figure 18 is $(\delta_0, \delta_1) = (1, 1)$; the resulting self-timed buffer bounds of e_a and e_b are, respectively, 1 and 2; and $\Phi = 2 + 1 = 3$.

Now $(2, 0)$ is an alternative assignment of delays on (e_0, e_1) that preserves the estimated throughput of the original graph. However, in this assignment, we see that the self-timed buffer bounds of e_a and e_b are identically equal to 2, and thus, $\Phi = 4$, one greater than the corresponding sum from the delay assignment $(1, 1)$ computed by *DetermineDelays*. Thus, if $\hat{G}_s$ denotes the graph returned by *Convert-to-SC-graph* for the example of Figure 18, we have that $\Phi\left(\hat{G}_s[e_0 \to \delta_0, e_1 \to \delta_1]\right) < \Phi\left(\hat{G}_s[e_0 \to 2, e_1 \to 0]\right)$, where $\Phi(X)$ denotes the sum of the self-timed buffer bounds over all IPC edges in X.

Algorithm *DetermineDelays* can easily be modified to optimally handle general graphs that have only one *source* SCC. Here, the algorithm specification remains essentially the same, with the exception that for $i = 1, 2, \ldots, (m-1)$, e_i denotes the edge directed from a vertex in D_{m-i} to a vertex in D_{m-i+1}, where $D_1, D_2, \ldots, D_m$ is the ordering of sink SCCs generated in Step 2 of the corresponding invocation of *Convert-to-SC-graph* (e_0 still denotes the sink-source edge instantiated by *Convert-to-SC-graph*). By adapting the reasoning behind Theorem 6, it is easily verified that when it is applicable, this modified algorithm always yields an optimal solution.

As far as we are aware, there is no straightforward extension of *DetermineDelays* to general graphs (multiple source SCCs and multiple sink SCCs) that is guaranteed to yield optimal solutions. The fundamental problem for the general case is the inability to derive the partitions $W_0, W_1, \ldots, W_{m-1}$ $(P_0, P_1, \ldots, P_{m-1})$ of the fundamental cycles (IPC sink-source paths) introduced by *Convert-to-SC-graph* such that each W_i (P_i) contains $e_0, e_1, \ldots, e_i$, and contains no other members of $E_s \equiv \{e_0, e_1, \ldots, e_{m-1}\}$, where E_s is the set of edges added by *Convert-to-SC-graph*. The existence of such partitions was crucial to our development of Theorem 6 because it implied that once the minimum values for $e_0, e_1, \ldots, e_i$ are successively computed, "transferring" delay from some e_i to some e_j, $j < i$, is never beneficial. Figure 19 shows an example of a synchronization graph that has multiple source SCCs and multiple sink SCCs, and that does not induce a partition of the desired form for the fundamental cycles.

However, *DetermineDelays* can be extended to yield heuristics for the general case in which the original synchronization graph G_s contains more than one source SCC *and* more than one sink SCC. For example, if $(a_1, a_2, \ldots, a_k)$ denote edges that were instantiated by *Convert-to-SC-graph* "between" the source SCCs — with each a_i representing the ith edge created — and similarly, $(b_1, b_2, \ldots, b_l)$ denote the sequence of edges instantiated between the sink SCCs, then algorithm *DetermineDelays* can be applied with the modification that $m = k + l + 1$, and $(e_0, e_1, \ldots, e_{m-1}) \equiv (e_s, a_1, a_2, \ldots, a_k, b_l, b_{l-1}, \ldots, b_1)$, where e_s is the sink-source edge from *Convert-to-SC-graph*.

The derivation of alternative heuristics for general synchronization graphs appears to be an interesting direction for further research. It should be noted, though, that practical synchronization graphs frequently contain either a single source SCC or a single SCC, or both — such as the example of Figure 15 — so that algorithm *DetermineDelays*, together with its counterpart for graphs that have a single source SCC, form a widely-applicable solution for optimally determining the delays on the edges created by *Convert-to-SC-graph.*

If we assume that there exist constants T and D such that $t(v) \leq T$, for all v, and $delay(e) \leq D$ for all edges e, then the complexity of *BellmanFord* is $O(|V||E|\log_2(|V|))$ [17]; and we have $\lambda_{max} \geq \frac{1}{D}$ and $\sum t(v) \leq T|V|$, so that $d_{ub} \leq DT|V|$. Thus, each invocation of *MinDelay* runs in $O(\log_2(DT|V|)|V||E|\log_2(|V|)) = O\left(|V||E|(\log_2(|V|))^2\right)$ time. It follows that *DetermineDelays* — and any of the variations of *DetermineDelays* defined above — is $O\left(m|V||E|(\log_2(|V|))^2\right)$, where m is the number of edges instantiated by *Convert-to-SC-graph.* Since $m = (n_{src} + n_{snk} - 1)$, where n_{src} is the number of source SCCs, and n_{snk} is the

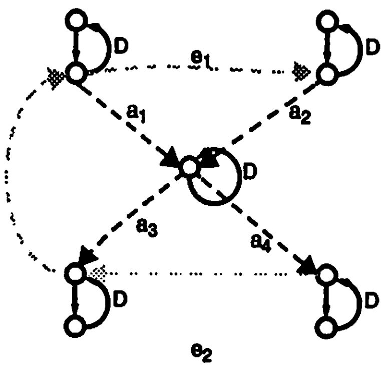


Figure 19. A synchronization graph, after processing by *Convert-to-SC-graph*, such that there is no m-way partition $W_0, W_1, \ldots, W_{m-1}$ of the fundamental cycles introduced by *Convert-to-SC-graph* that satisfies both (1). Each W_i contains $e_0, e_1, \ldots, e_i$ *and* (2). Each W_i does not contain any member of $e_{i+1}, e_{i+2}, \ldots, e_{m-1}$. Here, the fundamental cycles introduced by *Convert-to-SC-graph* (the grey dashed edges are the edges instantiated by *Convert-to-SC-graph*) are (e_0, a_1, a_3), (e_0, e_1, a_2, a_3), $(e_0, e_1, a_2, a_4, e_2)$, and (e_0, a_1, a_4, a_2). It is easily verified that these cycles cannot be decomposed into a partition of the above form even if we are allowed to reorder the e_i's.

number of sink SCCs, it is obvious that $m < |V|$. With this observation, and the observation that $|E| \le |V|^2$, we have that *DetermineDelays* and its variations are $O\left(|V|^4 (\log_2 (|V|))^2\right)$. Furthermore, it is easily verified that the time complexity of *DetermineDelays* dominates that of *Convert-to-SC-graph,* so the time complexity of applying *Convert-to-SC-graph* and *DetermineDelays* in succession is also $O\left(|V|^4 (\log_2 (|V|))^2\right)$.

Although the issue of deadlock does not explicitly arise in algorithm *DetermineDelays,* the algorithm does guarantee that the output graph is not deadlocked, assuming that the input graph is not deadlocked. This is because (from Lemma 1) deadlock is equivalent to the existence of a cycle that has zero path delay, and is thus equivalent to an infinite maximum cycle mean. Since *DetermineDelays* does not increase the maximum cycle mean, it follows that the algorithm cannot convert a graph that is not deadlocked into a deadlocked graph.

10.3 Related Work

Converting a mixed grain DFG that contains feedforward edges into a strongly connected graph has been studied by Zivojnovic [35] in the context of retiming when the assignment of actors to processors is fixed beforehand. In this case, the objective is to retime the input graph so that the number of IPC edges that have nonzero delay is maximized, and the conversion is performed to constrain the set of possible retimings in such a way that an integer linear programming formulation can be developed. The technique generates two dummy vertices that are connected by an edge; the sink vertices of the original graph are connected to one of the dummy vertices, while the other dummy vertex is connected to each source. It is easily verified that in a self-timed execution, this scheme requires at least four more synchronization accesses per graph iteration than the method that we have proposed. We can obtain further relative savings if we succeed in detecting one or more beneficial resynchronization opportunities. The effect of Zivojnovic's retiming algorithm on synchronization overhead is unpredictable since one hand an IPC edge becomes "easier to make redundant" when its delay increases, while on the other hand, the edge becomes less useful in making other IPC edges redundant since the path delay of all paths that contain the edge increase.

11. Complete Algorithm

In this section we outline our complete synchronization optimization algorithm. The input to the algorithm is a DFG and a parallel schedule for it. The output from the algorithm is an IPC graph $G_{ipc} = (V, E_{ipc})$, which represents buffers as IPC edges; a strongly connected synchronization graph $G_s = (V, E_s)$, which represents synchronization constraints; and a set of shared-

```
Function SynchronizationOptimize
Input: A DFG $G$ and a self-timed schedule for this DFG.
Output: $G_{ipc}$, $G_s$, and $\{B_{fb}(e) \mid e \text{ is an IPC edge in } G_{ipc}\}$.

1. Extract $G_{ipc}$ from $G$ and the given parallel schedule (which specifies actor assignment to
processors and the order in which each actor executes on a processor)

2. Set $G_s = G_{ipc}$                      /* Each IPC edge is also a synchronization
                                              edge to begin with */

3. $G_s = RemoveRedundantSynchs(G_s)$

4. $G_s = Resynchronize(G_s)$

5. $G_s = Convert\text{-}to\text{-}SC\text{-}graph(G_s)$

6. $G_s = DetermineDelays(G_s)$

/* Remove any synchronization edges that have become redundant as a result of the appli-
cation of Convert-to-SC-graph. */
7. $G_s = RemoveRedundantSynchs(G_s)$

8. Calculate buffer sizes $B_{fb}(e)$ for each IPC edge $e$ in $G_{ipc}$. (to be used for implementing
the BBS protocol)
        a) Compute $\rho_{G_s}(src(e), snk(e))$, the path delay of a minimum-delay
        path in $G_s$ directed from $src(e)$ to $snk(e)$
        b) Set $B_{fb}(e) = \rho_{G_s}(src(e), snk(e)) + delay(e)$
```

Figure 20. The complete synchronization optimization algorithm.

memory buffer sizes $\{B_{fb}(e) \mid e \text{ is an IPC edge in } G_{ipc}\}$, which specifies the amount of memory to allocate in shared memory for each IPC edge.

The pseudocode for the complete algorithm is given in Figure 20. Here, *RemoveRedundantSynchs* is invoked twice, once at the beginning, and once again after *Convert-to-SC-graph* and *DetermineDelays*. It is possible that the edge(s) added by *Convert-to-SC-graph* can make some of the existing synchronization edges redundant, and thus, applying *RemoveRedundantSynchs* after *Convert-to-SC-graph* may be beneficial.

A code generator can then accept G_{ipc} and G_s, and allocate a buffer in shared memory for each IPC edge e specified by G_{ipc} of size $B_{fb}(e)$, and generate synchronization code for the synchronization edges represented in G_s. These synchronizations may be implemented using the BBS protocol described in Subsection 6.1. The synchronization cost in the final implementation is thus equal to $2n_s$, where n_s is the number of synchronization edges in G_s.

12. Summary

We have addressed the problem of minimizing synchronization overhead when implementing self-timed, iterative dataflow programs. We have introduced a graph-theoretic analysis framework that allows us to determine the effects on throughput and buffer sizes of modifying the points in the target program at which synchronization functions are carried out, and we have used this framework to extend an existing technique — removal of redundant synchronization edges — for noniterative programs to the iterative case, and to develop two new methods for reducing synchronization overhead — resynchronization and the conversion of the synchronization graph into a strongly connected graph. Finally, we have shown how our techniques can be combined, and how the result can be post processed to yield a format from which IPC code can easily be generated.

The premise of our work is that estimates are available for the execution times of actors such that the actual execution time of an actor exhibits large variation from its corresponding estimate only with very low frequency. Accordingly our techniques have been devised to guarantee that if the actual execution time of each actor invocation is always equal to the corresponding execution time estimate, then the throughput of an implementation that incorporates our synchronization minimization techniques is never less than the throughput of a corresponding unoptimized implementation — that is, we never accept an opportunity to reduce synchronization overhead if it constrains execution in such a way that throughput is increased. Thus, our work is particularly relevant to embedded DSP applications, where the price of synchronization is high, and accurate execution time estimates are often available, but guarantees on these execution times do no exist due to infrequent events such as cache misses and error handling.

13. Further Work

Several directions for further work emerge from the study presented in this paper. Perhaps the most significant is the incorporation of timing guarantees — for example, hard upper and lower execution time bounds, as Dietz, Zaafrani, and O'keefe use in [8]; and handling of a mix of actors some of which have guaranteed execution time bounds, and some that have no such guarantees, as Filo, Ku, Coelho Jr., and De Micheli do in [9]. Such guarantees could be used to detect situations in which IPC data will always be available (produced) before it is needed for consumption. Upper and lower bounds also make it an interesting issue to define what the objective of "preserving estimated throughput" means — for example: How can we formulate a constraint, incorporating guaranteed execution time upper and lower bounds, to efficiently prevent synchronization optimization from introducing cycles that can significantly degrade the throughput?

Also, execution time guarantees can be used to compute tighter buffer size bounds. As a simple example, consider Figure 21. Here, the analysis of Section 5.3 yields a buffer size $B_{fb}((A, B)) = 3$, since 3 is the minimum path delay of a cycle that contains (A, B). However, if $t(A)$ and $t(B)$ are guaranteed to be equal to the same constant, then it is easily verified that a buffer size of 1 will suffice for (A, B). Systematically applying execution time guarantees to derive lower buffer size bounds appears to be a promising direction for further work.

We have shown that pairwise resynchronization can be attacked with arbitrary heuristics for set covering. It would be useful to study which of the existing set covering heuristics are best suited to addressing resynchronization in practical applications. Conceivably, there is also opportunity to devise new heuristics that exploit certain properties of applications with regards to resynchronization that are not taken into account by existing set covering heuristics. We have shown that a heuristic for general (not just pairwise) resynchronization can be derived from any given heuristic for pairwise resynchronization by simply applying the pairwise resynchronization heuristic to each pair of distinct SCCs. It appears to be a significant challenge to devise a more global approach to the general (not just pairwise) resynchronization problem.

Finally, there is considerable room for refinement in our techniques for converting the synchronization graph into a strongly connected graph. For example, currently the ordering of SCCs in the source and sink chains is performed arbitrarily. However, their ordering can impact both the total shared memory requirement (self-timed buffer bounds), and the number of redundant synchronizations introduced by the new edges added by *Convert-to-SC-graph*. Thus, it would be useful to study techniques to optimize the ordering of the source and sink SCCs with regard to one or both of these criteria.

Our technique for computing the delays on the edges introduced by *Convert-to-SC-graph* is optimal under the assumption that there is one source SCC or one sink SCC. Although this assumption is frequently satisfied in practice, it may be interesting to examine whether or not an efficient scheme can be devised to determine the delays optimally for general synchronization graphs.

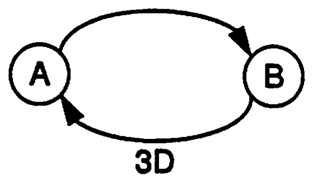


Figure 21. An example of how execution time guarantees can be used to reduce buffer size bounds.

Acknowledgment

Tom Parks at the University of California at Berkeley designed and implemented the original version of the Karplus-Strong system in the Ptolemy design environment [5], from which the synchronization graph of Figure 15 was obtained.

Appendix

In this appendix, we establish the NP completeness of the resynchronization problem, which was defined in Section 9. We establish this by reducing an arbitrary instance of the set-covering problem, a well-known NP-hard problem, to an instance of the pairwise resynchronization problem, which is a special case of the resynchronization problem that occurs when there are exactly two SCCs. The intuition behind this reduction is explained in Section 9.

Suppose that we are given an instance (X, T) of set covering, where X is a finite set, and T is a family of subsets of X that covers X. Without loss of generality, we assume that

T does *not* contain a proper nonempty subset T' that satisfies $\left(\bigcup_{t \in (T - T')} t\right) \cap \left(\bigcup_{t \in T'} t\right) = \emptyset$. (13)

We can assume this without loss of generality because if this assumption does not hold, then we can apply the construction below to each "independent subfamily" separately, and then combine the results to get a minimal cover for X.

The following steps specify how we construct a DFG from (X, T). Except where stated otherwise, no delay is placed on the edges that are instantiated.

1. For each $x \in X$, instantiate two vertices $vsrc(x)$ and $vsnk(x)$, and instantiate an edge $e(x)$ directed from $vsrc(x)$ to $vsnk(x)$.

2. For each $t \in T$

   (a). Instantiate two vertices $vsrc(t)$ and $vsnk(t)$.

(b). For each $x \in t$

- Instantiate an edge directed from $vsrc(x)$ to $vsrc(t)$.
- Instantiate an edge directed from $vsrc(t)$ to $vsrc(x)$, and place one delay on this edge.
- Instantiate an edge directed from $vsnk(t)$ to $vsnk(x)$.
- Instantiate an edge directed from $vsnk(x)$ to $vsnk(t)$, and place one delay on this edge.

3. For each vertex v that has been instantiated, instantiate an edge directed from v to itself, and place one delay on this edge.

Observe from our construction, that whenever $x \in X$ is contained in $t \in T$, there is an edge directed from $vsrc(x)$ ($vsnk(t)$) to $vsrc(t)$ ($vsnk(x)$), and there is also an edge (having unit delay) directed from $vsrc(t)$ ($vsnk(x)$) to $vsrc(x)$ ($vsnk(t)$). Thus, from the assumption stated in (13), it follows that $\{vsrc(z) | z \in (X \cup T)\}$ forms one SCC, $\{vsnk(z) | z \in (X \cup T)\}$ forms another SCC, and $F \equiv \{e(x) | x \in X\}$ is the set of feedforward edges.

Let G denote the DFG that we have constructed, and as in Section 9, define $\chi(p) \equiv \{e \in F | (p \; subsumes \; (src(e), snk(e)))\}$ for each ordered pair of vertices $p = (y_1, y_2)$ such that y_1 is contained in the source SCC of G, and y_2 is contained in the sink SCC of G. Clearly, G gives an instance of the pairwise resynchronization problem.

Observation 2: By construction of G, observe that $\{x \in X | ((vsrc(t), vsnk(t)) \; subsumes \; (vsrc(x), vsnk(x)))\} = t$, for all $t \in T$. Thus, for all $t \in T$, $\chi(vsrc(t), vsnk(t)) = \{e(x) | x \in t\}$.

Observation 3: For each $x \in X$, all input edges of $vsrc(x)$ have unit delay on them. It follows that for any vertex y in the sink SCC of G,

$\chi(vsrc(x), y) \subseteq \{e \in F \mid src(e) = vsrc(x)\} = \{e(x)\}$.

Observation 4: For each $t \in T$, the only vertices in G that have a delay-free path to $vsrc(t)$ are those vertices contained in $\{vsrc(x) \mid x \in t\}$. It follows that for any vertex y in the sink SCC of G, $\chi(vsrc(t), y) \subseteq \chi(vsrc(t), vsnk(t)) = \{e(x) \mid x \in t\}$.

Now suppose that $F' = \{f_1, f_2, \ldots, f_m\}$ is a minimal resynchronization of G. For each $i \in \{1, 2, \ldots, m\}$, exactly one of the following two cases must apply

Case 1: $vsrc(f_i) = vsrc(x)$ for some $x \in X$. In this case, we pick an arbitrary $t \in T$ that contains x, and we set $v_i = vsrc(t)$ and $w_i = vsnk(t)$. From Observation 3, it follows that $\chi((src(f_i), snk(f_i))) \subseteq \{e(x)\} \subseteq \chi(v_i, w_i)$.

Case 2: $vsrc(f_i) = vsrc(t)$ for some $t \in T$. We set $v_i = vsrc(t)$ and $w_i = vsnk(t)$. From Observation 4, we have $\chi((src(f_i), snk(f_i))) \subseteq \chi(v_i, w_i)$.

Observation 5: From our definition of the v_is and w_is, $\{d_0(v_i, w_i) \mid (i \in \{1, 2, \ldots, m\})\}$ is a minimal resynchronization of G. Also, each (v_i, w_i) is of the form $(vsrc(t), vsnk(t))$, where $t \in T$.

Now, for each $i \in \{1, 2, \ldots, m\}$, we define

$$Z_i \equiv \{x \in X \mid (v_i, w_i) \textit{ subsumes } (vsrc(x), vsnk(x))\} .$$

Proposition 1: $\{Z_1, Z_2, \ldots, Z_m\}$ covers X.

Proof: From Observation 5, we have that for each Z_i, there exists a $t \in T$ such that

$Z_i = \{x \in X | (vsrc(t), vsnk(t)) \textit{ subsumes } (vsrc(x), vsnk(x))\}$. Thus, each Z_i is a member of T. Also, since $\{d_o(v_i, w_i) | (i \in \{1, 2, ..., m\})\}$ is a resynchronization of G, each member of $\{(vsrc(x), vsnk(x)) | x \in X\}$ must be preserved by some (v_i, w_i), and thus each $x \in X$ must be contained in some Z_i. *QED.*

Proposition 2: $\{Z_1, Z_2, ..., Z_m\}$ is a minimal cover for X.

Proof: (By contraposition). Suppose there exists a cover $\{Y_1, Y_2, ..., Y_{m'}\}$ (among the members of T) for X, with $m' < m$. Then, each $x \in X$ is contained in some Y_j, and from Observation 2, $(vsrc(Y_j), vsnk(Y_j))$ subsumes $e(x)$. Thus, $\{(vsrc(Y_i), vsnk(Y_i)) | (i \in \{1, 2, ..., m'\})\}$ is a resynchronization of G. Since $m' < m$, it follows that $F' = \{f_1, f_2, ..., f_m\}$ is not a minimal resynchronization of G. *QED.*

In summary, we have shown how to convert an arbitrary instance (X, T) of the set covering problem into an instance G of the pairwise resynchronization problem, and we have shown how to convert a solution $F' = \{f_1, f_2, ..., f_m\}$ of this instance of pairwise resynchronization into a solution $\{Z_1, Z_2, ..., Z_m\}$ of (X, T). It is easily verified that all of the steps involved in deriving G from (X, T), and in deriving $\{Z_1, Z_2, ..., Z_m\}$ from F' can be performed in polynomial time. Thus, from the NP hardness of set covering [7], we can conclude that the pairwise resynchronization problem is NP hard.

Glossary

$\rho(x, y)$	Same as ρ_G with the DFG G understood from context.
$\rho_G(x, y)$	If there is no path in G from x to y, then $\rho_G(x, y) = \infty$; otherwise, $\rho_G(x, y) = Delay(p)$, where p is any minimum-delay path from x to y.
$Delay(p)$	Given a path p, $Delay(p)$ is the sum of the edge delays over all edges in p.
$d_n(u, v)$	Represents an edge whose source and sink vertices are u and v, respectively, and whose delay is equal to n.
λ_{max}	Represents the maximum cycle mean of a DFG.
BBS	Bounded buffer synchronization. A synchronization protocol that may be used for feedback edges in a synchronization graph. This protocol requires two synchronization accesses per schedule period.
critical cycle	A fundamental cycle in a DFG whose cycle mean is equal to the maximum cycle mean of the DFG.
cycle mean	The cycle mean of a cycle C in a DFG is equal to T/D, where T is the sum of the execution times of all vertices on C, and D is the sum of delays of all edges in C.
estimated throughput	Given a DFG with execution time estimates for the actors, the estimated throughput is the reciprocal of the maximum cycle mean.
feedback edge	An edge that is contained in at least one cycle.
feedforward edge	An edge that is not contained in a cycle.
maximum cycle mean	Given a DFG, the maximum cycle mean is the largest cycle mean over all fundamental cycles in the DFG.
SCC	Strongly connected component.
self-timed buffer bound	Given a feedback edge e in a synchronization graph, the self-timed buffer bound is an upper bound on the number of tokens that can simultaneously reside on e (the buffer size).
synchronization access	An access to shared memory that used to update or examine the status of a synchronization variable.
synchronization cost	The average number of synchronization accesses that must be performed per iteration period in the self timed implementation of a

DFG.

$t(v)$	The execution time or estimated execution time of actor v.
UBS	Unbounded buffer synchronization. A synchronization protocol that must be used for feedforward edges of the synchronization graph. This protocol requires four synchronization accesses per iteration period.

References

[1] S. Banerjee, D. Picker, D. Fellman, and P. M. Chau, “Improved Scheduling of Signal Flow Graphs onto Multiprocessor Systems Through an Accurate Network Modelling Technique,” *VLSI Signal Processing VII*, IEEE Press, 1994.

[2] A. Benveniste and G. Berry, “The Synchronous Approach to Reactive and Real-Time Systems,” *Proceedings of the IEEE*, **Vol. 79, No. 9**, 1991, pp.1270-1282.

[3] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete, “Static Scheduling of Multi-Rate and Cyclo-Static DSP-Applications,” *VLSI Signal Processing VII*, IEEE Press, 1994.

[4] S. Borkar *et. al.*, “iWarp: An Integrated Solution to High-Speed Parallel Computing”, *Proceedings of Supercomputing 1988 Conference*, Orlando, Florida, 1988.

[5] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, “Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems,” *International Journal of Computer Simulation*, 1994.

[6] L-F. Chao and E. H-M. Sha, *Static Scheduling for Synthesis of DSP Algorithms on Various Models*, technical report, Department of Computer Science, Princeton University, 1993.

[7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, McGraw-Hill, 1990.

[8] H. G. Dietz, A. Zaafrani, and M. T. O’keefe, “Static Scheduling for Barrier MIMD Architectures,” *Journal of Supercomputing*, **Vol. 5, No. 4**, 1992.

[9] D. Filo, D. C. Ku, C. N. Coelho Jr., and G. De Micheli, “Interface Optimization for Concurrent Systems Under Timing Constraints,” *IEEE Transactions on Very Large Scale Integration*, **Vol. 1, No. 3**, September, 1993.

[10] R. Govindarajan, G. R. Gao, and P. Desai, “Minimizing Memory Requirements in Rate-Optimal Schedules,” *Proceedings of the International Conference on Application Specific Array Processors*, San Francisco, August, 1994.

[11] T. C. Hu, “Parallel Sequencing and Assembly Line Problems,” *Operations Research*, **Vol. 9**, 1961.

[12] J. A. Huisken et. al., “Synthesis of Synchronous Communication Hardware in a Multiprocessor Architecture,” *Journal of VLSI Signal Processing*, **Vol. 6**, pp.289-299, 1993.

[13] A. Kalavade, and E. A. Lee, "A Hardware/Software Codesign Methodology for DSP Applications," *IEEE Design and Test*, September 1993, vol. 10, no. 3, pp. 16-28.

[14] W. Koh, "A Reconfigurable Multiprocessor System for DSP Behavioral Simulation", Ph.D. Thesis, Memorandum No. UCB/ERL M90/53, Electronics Research Laboratory, University of California at Berkeley, June, 1990.

[15] S. Y. Kung, P. S. Lewis, and S. C. Lo, "Performance Analysis and Optimization of VLSI Dataflow Arrays," *Journal of Parallel and Distributed Computing*, Vol. 4, 1987.

[16] R. Lauwereins, M. Engels, J.A. Peperstraete, E. Steegmans, and J. Van Ginderdeuren, "GRAPE: A CASE Tool for Digital Signal Parallel Processing," *IEEE ASSP Magazine*, Vol. 7, No. 2, April, 1990.

[17] E. Lawler, *Combinatorial Optimization: Networks and Matroids*," Holt, Rinehart and Winston, pp. 65-80, 1976.

[18] E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Dataflow Programs for Digital Signal Processing," *IEEE Transactions on Computers*, February, 1987.

[19] E. A. Lee, and S. Ha, "Scheduling Strategies for Multiprocessor Real-Time DSP," *Globecom*, November 1989.

[20] E. A. Lee, and J. C. Bier, "Architectures for Statically Scheduled Dataflow," *Journal of Parallel and Distributed Computing*, December 1990.

[21] G. Liao, G. R. Gao, E. Altman, and V. K. Agarwal, *A Comparative Study of DSP Multiprocessor List Scheduling Heuristics*, technical report, School of Computer Science, McGill University.

[22] D. R. O'Hallaron, *The Assign Parallel Program Generator*, Memorandum CMU-CS-91-141, School of Computer Science, Carnegie Mellon University, May, 1991.

[23] K. K. Parhi and D. G. Messerschmitt, "Static Rate-Optimal Scheduling of Iterative Data-Flow Programs via Optimum Unfolding," *IEEE Transactions on Computers*, Vol. 40, No. 2, February, 1991.

[24] J. L. Peterson, Petri Net Theory and the Modelling of Systems, Prentice-Hall Inc., 1981.

[25] J. Pino, S. Ha, E. A. Lee, and J. T. Buck, "Software Synthesis for DSP Using Ptolemy," *Journal of VLSI Signal Processing*, Vol. 9, No. 1, January, 1995, to appear.

[26] D. B. Powell, E. A. Lee, and W. C. Newman, "Direct Synthesis of Optimized DSP Assembly Code from Signal Flow Block Diagrams," *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, San Francisco, March, 1992.

[27] H. Printz, *Automatic Mapping of Large Signal Processing Systems to a Parallel Machine*, Ph.D. thesis, Memorandum CMU-CS-91-101, School of Computer Science, Carnegie Mellon University, May, 1991.

[28] R. Reiter, Scheduling Parallel Computations, *Journal of the Association for Computing*

Machinery, October 1968.

[29] S. Ritz, M. Pankert, and H. Meyr, "High Level Software Synthesis for Signal Processing Systems," *Proceedings of the International Conference on Application Specific Array Processors*, Berkeley, August, 1992.

[30] P. L. Shaffer, "Minimization of Interprocessor Synchronization in Multiprocessors with Shared and Private Memory," *International Conference on Parallel Processing*, 1989.

[31] G. C. Sih and E. A. Lee, "Scheduling to Account for Interprocessor Communication Within Interconnection-Constrained Processor Networks, *International Conference on Parallel Processing*, 1990.

[32] S. Sriram and E. A. Lee, "Statically Scheduling Communication Resources in Multiprocessor DSP architectures," *Proceedings of the Asilomar Conference on Signals, Systems, and Computers*, November, 1994.

[33] S. Sriram, E. A. Lee, "Design and Implementation of an Ordered Memory Access Architecture," *Proceedings of the International Conference on Acoustics Speech and Signal Processing*, April, 1993.

[34] P. P. Vaidyanathan, *Multirate Systems and Filter Banks*, Prentice Hall, 1993.

[35] V. Zivojnovic, H. Koerner, and H. Meyr, "Multiprocessor Scheduling with A-priori Node Assignment," *VLSI Signal Processing VII*, IEEE Press, 1994.