— *IEEE Transactions on Signal Processing*, Vol. 45, No. 6, June 1997 —

OPTIMIZING SYNCHRONIZATION IN MULTIPROCESSOR DSP SYSTEMS

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

February 11, 1997

ABSTRACT

This paper is concerned with multiprocessor implementations of embedded applications specified as iterative dataflow programs, in which synchronization overhead can 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, based on a data structure called the *synchronization graph*, for analyzing and optimizing synchronization overhead in self-timed, iterative dataflow programs. 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 present an optimization that converts a feedforward dataflow graph into a strongly connected graph in such a way as to reduce synchronization overhead without slowing down execution.

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, 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., 201 East Tasman Drive., San Jose, California 95134, USA. Phone: (408)922-4121, FAX: (408)954-8907, Electronic mail: shuvra@hmsi.hitachi.com.

S. Sriram is with the DSP R&D Centre at Texas Instruments, MS446, 13510 North Central Expressway, Dallas, TX75265-5474, USA. Phone: (972)995-0336, FAX: (972)995-6194, Electronic mail: sriram@hc.ti.com,.

E. A. Lee is with the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, California 94720, USA, Phone: (510)642-0455, FAX: (510)642-2739, Electronic mail: eal@eecs.berkeley.edu.

1. Introduction

Inter-processor synchronization overhead can severely limit the speedup of a multiprocessor implementation. This paper develops techniques to minimize synchronization overhead in shared-memory multiprocessor implementations of iterative synchronous dataflow (SDF) programs. Our study is motivated by the widespread popularity of the SDF model in DSP design environments and the suitability of this model for exploiting parallelism. 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 arbitrary magnitude can occasionally occur due to phenomena such as cache misses or error handling.

SDF and closely related models have been used widely in DSP design environments, such as those described in [14, 19, 22, 25]. In SDF, 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 for each firing of that actor, and is known at compile time [16] 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 [16], 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.

Delays on DFG edges represent initial tokens, and specify dependencies between iterations of the actors in iterative execution. 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 (A, B) contains two delays. We represent an edge with n delays by annotating it with the symbol "nD" (see Fig. 1).

Multiprocessor implementation of an algorithm specified as a DFG involves scheduling the actors. By "scheduling" we collectively refer to the tasks 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. In [17] the authors propose a scheduling taxonomy based on which of these tasks are performed at compile time (static strategy) and which at run time (dynamic strategy); in this paper we will use the same terminology that was introduced there.

In the **fully-static** scheduling strategy of [17], all three scheduling tasks are performed at compile time. This strategy involves the least possible runtime overhead. All 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 generally not practical. A more realistic assumption for DSP algorithms is that good estimates for the execution times of actors can be obtained.

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 [17]. 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 is ensured. As a result, the self-timed strategy incurs greater run-time cost than the fully-static case because of the synchronization overhead.

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 suspend execution until the appropriate condition is met. In each kind of platform, every IPC that requires such synchronization checks 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 [10].

The main goal of this paper is to present 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. 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. As a performance metric for evaluating DFG implementations we use the average iteration period T, (or equivalently the throughput T^{-1}) which is the average time that it takes for all the actors in the graph to be executed once. Thus an optimal schedule is one that minimizes T.

2. Related Work

Numerous research efforts have focused on constructing efficient parallel schedules for DFGs. For example in [5, 20], techniques are developed for exploiting overlapped execution to optimize throughput, assuming zero cost for IPC. Other work has focused on taking IPC costs into account during scheduling [1, 18, 23, 27], while not explicitly addressing overlapped execution. Similarly, in [9], techniques are developed 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.

Among the prior work that is most relevant to this paper is the *barrier-MIMD* concept, discussed in [7]. However, the techniques of barrier MIMD do not apply to our problem context because they assume a hardware barrier mechanism; 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 DFG can overlap; and because even for non-iterative execution, there appears to be no obvious correspondence [3] 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**).

In [26], Shaffer presents an algorithm that minimizes the number of directed synchronizations in the self-timed execution of a DFG. However, this work, like that of Dietz et al., does not allow the execution of successive iterations of the DFG to overlap. It also avoids having to consider dataflow edges that have delay. The technique that we present for removing redundant synchronizations generalizes Shaffer's algorithm to handle delays and overlapped, iterative execution. The other major technique that we present for optimizing synchronization — handling the feedforward edges of the *synchronization graph* — is fundamentally different from Shaffer's technique since it addresses issues that are specific to our more general context of overlapped, iterative execution.

3. Terminology

We represent a DFG by an ordered pair (V, E), where V is the set of vertices and E is the set of edges. The source vertex, sink vertex and delay of an edge e are denoted $src(e)$, $snk(e)$ and $delay(e)$.

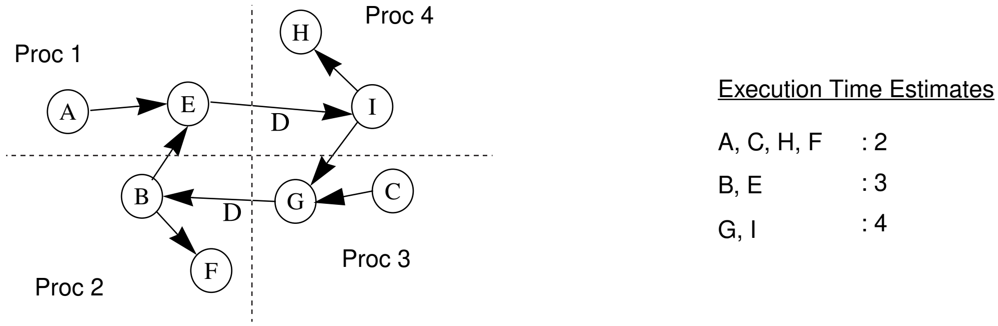
A **path** in (V, E) is a finite, nonempty sequence $(e_1, e_2, \ldots, e_n)$, where each e_i is a member of E, and $snk(e_1) = src(e_2)$, $snk(e_2) = src(e_3)$, ..., $snk(e_{n-1}) = src(e_n)$. A path that is directed from some vertex to itself is called a **cycle**, and a **fundamental cycle** is a cycle of which no proper subsequence is a cycle. If $p = (e_1, e_2, \ldots, e_n)$ is a path in (V, E), we define the **path delay** of p, denoted $Delay(p)$, by $Delay(p) = \sum_{i=1}^{n} delay(e_i)$. Between any two vertices $x, y \in V$, either there is no path from x to y, or there exists a **minimum-delay path** from x to y. That is, if there is a path from x to y, then there exists a path p from x to y such that $Delay(p') \ge Delay(p)$, for all paths p' directed from x to y. Given a DFG G, and vertices x, y, we define $\rho_G(x, y)$ to be ∞ 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 exists a path from x to y.

A DFG (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. 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, its associated subgraph is also called an SCC. An SCC V' of a DFG (V, E) is a **source SCC** if $\forall e \in E$, $(snk(e) \in V') \Rightarrow (src(e) \in V')$; V' is a **sink SCC** if $(src(e) \in V') \Rightarrow (snk(e) \in V')$. An edge e is a **feedforward** edge of (V, E) if it is not contained in an SCC; an edge that is contained in an SCC is called a **feedback** edge.

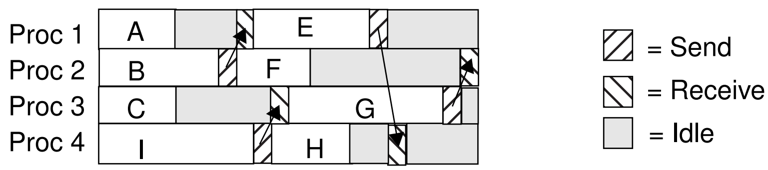
We denote the number of elements in a finite set S by $|S|$. Also, if r is a real number, then we denote the smallest integer that is greater than or equal to r by $\lceil r \rceil$. Finally, if x, y are vertices in (V, E), we define $d_n(x, y)$ to represent an edge (that is not necessarily in E) whose source and sink vertices are x and y, respectively, and whose delay is n.

4. Analysis of Self-Timed Execution

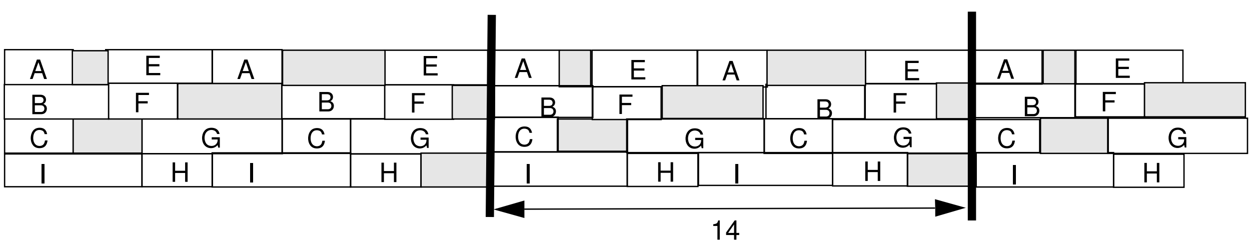
Fig. 1(c) illustrates the self-timed execution of the four-processor schedule in Fig. 1(a&b) (IPC is ignored here). If the timing estimates are accurate, the schedule execution settles into a repeating pattern spanning two iterations of G, and the average estimated iteration period is 7 time units. In this section we develop an analytical model to study such an execution of a self-timed schedule.



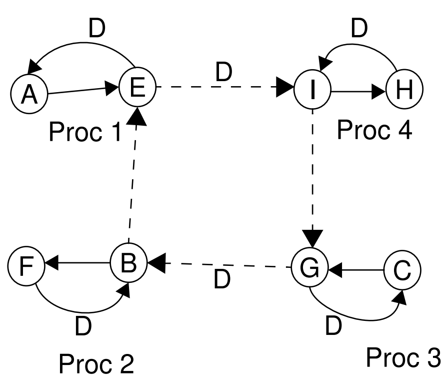
(a) DFG “G”



(b) Schedule on four processors



(c) Self-timed execution



(d) The IPC graph

Figure 1. Self-timed execution.

4.1 Inter-processor Communication Modelling Graph

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 Fig. 1 (b) can be modelled by the IPC graph in Fig. 1 (d). 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 Fig. 1, 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 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 Fig. 1, data produced by actor B is sent from processor 2 to processor 1; instead of inserting explicit communication primitives in the schedule, we model the send within actor B and we model the receive as part of actor E.

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. 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.

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:

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

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 (*send*s and *receive*s) that the actor v performs. Now, we can substitute $end(v_j, k) = start(v_j, k) + t(v_j)$ in (1) to obtain

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

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:

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

The IPC graph can also be looked upon as a timed marked graph [21] or Reiter's computation graph [24]. The same properties hold for it, and we state some of the relevant properties here. See [24] for proofs of Lemmas 1 and 3, and [3] for a proof of Lemma 2.

Lemma 1: [24] 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: [3] 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.

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 [24]:

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

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

The quotient in (4) is called the **cycle mean** of the cycle C. 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 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}, because the execution of the schedule is constrained by the slowest component in the system. Henceforth, we will define the maximum cycle mean as follows.

Definition 2: The **maximum cycle mean** of an IPC graph G_{ipc}, denoted by λ_{max}, is the maximal cycle mean over all SCCs 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 cycle in G_{ipc} whose cycle mean is 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 Fig. 1(d), G_{ipc} has one SCC, and its maximal cycle mean is 7 time units. This corresponds to the critical cycle $((B, E), (E, I), (I, G), (G, B))$. 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 $O(|V||E_{ipc}|\log_2(|V| + D + T))$, where D and T are such that $delay(e) \le D \ \ \forall e \in E_{ipc}$ and $t(v) \le T \ \ \forall v \in V$ [15].

4.2 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 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.

4.3 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, the number of tokens on each feedback edge (an edge that belongs to an SCC, and hence to some cycle) during the execution of the IPC graph is bounded above by a constant. 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:

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

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, Fig. 2(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 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

Fig. 2. An IPC graph with a feedforward edge: (a). original graph (b). imposing bounded buffers.

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 Fig. 2(b)). Since adding this edge introduces a new cycle in G_{ipc}, it may reduce the estimated throughput; to prevent such a reduction, $B_{ff}(e)$ must be chosen 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 [13], 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

$$B_{ff}(e) \geq \left\lceil \left(\sum_{x \in V} t(x) \right) \Big/ \lambda_{max} \right\rceil \tag{6}$$

holds for each feedforward edge e, then the maximum cycle mean of the resulting graph does not exceed λ_{max}. This is because the reverse edge that gets added as a result of imposing a buffer bound on e introduces new cycles; the maximum execution time along any such newly introduced cycle can be at most $\sum_{x \in V} t(x)$; and hence adding the number of delays given by (6) guarantees no change in the maximum cycle mean.

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 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 9, 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}.

5. Synchronization Model

5.1 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 4.3.

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 in the case of edges for which $B_{fb}(e)$ is too large to be practically implementable, smaller bounds must be imposed, using a protocol identical to UBS.

5.2 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 UBS or BBS. 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 4.1 that an IPC edge (v_j, v_i) of G_{ipc} represents the **synchronization constraint**:

$$start(v_i, k) \geq end(v_j, k - delay((v_j, v_i))) \quad \forall k > delay(v_j, v_i). \tag{7}$$

Initially, 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. 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 UBS and BBS. 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.

All transformations that we perform on G_s 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 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 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 $delay(\varepsilon)$, that is the following condition holds: $\forall \varepsilon \in E_2, \varepsilon \notin E_1$, $\rho_{G_1}(src(\varepsilon), snk(\varepsilon)) \le delay(\varepsilon)$.

(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 $p = (e_1, e_2, \ldots, e_n)$ is a path in G_1, then

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

Proof: The following constraints hold along such a path p (as per (7))

$$start(snk(e_1), k) \ge end(src(e_1), k - delay(e_1)) . \tag{8}$$

Similarly,

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

Noting that $src(e_2) = snk(e_1)$, we obtain $start(snk(e_2), k) \ge end(snk(e_1), k - delay(e_2))$.

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

$$start(snk(e_2), k) \ge start(snk(e_1), k - delay(e_2)) . \tag{9}$$

Substituting (8) in (9),

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

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

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

that is,

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

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:

$$start(snk(\varepsilon), k) > end(src(\varepsilon), k - delay(\varepsilon)) \tag{10}$$

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

From Lemma 4, the existence of such a path p implies

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

That is,

$$start(snk(\varepsilon), k) \geq end(src(\varepsilon), k - Delay(p)) . \tag{11}$$

If $Delay(p) \leq delay(\varepsilon)$, then $end(src(\varepsilon), k - Delay(p)) \geq end(src(\varepsilon), k - delay(\varepsilon))$. Substituting this in (11) we obtain

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

The above relation is identical to (10), and this proves the theorem. *Q. E. D.*

Theorem 1 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).

Given an IPC graph G_{ipc}, and a synchronization graph G_s such that G_s preserves G_{ipc}, if we implement the synchronizations corresponding to the synchronization edges of G_s, then, because the synchronization edges alone determine the interaction between processors, the iteration period of the resulting system is determined by the maximal cycle mean of G_s.

5.3 Computing Buffer Bounds from G_s and G_{ipc}

After all the optimizations are complete we have a final synchronization graph 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 bounds as a final step after all the transformations on G_s are complete, instead of deriving the bounds from G_{ipc}. 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. 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. Since the initial number of tokens on e is $delay(e)$, the size of the buffer corresponding to e is bounded above by $B_{fb}(e) = \rho_{G_s}(snk(e), src(e)) + delay(e)$. *Q. E. D.*

The quantities $\rho_{G_s}(snk(e), src(e))$ can be computed using Dijkstra's algorithm [6] to solve the all-pairs shortest path problem on the synchronization graph in time $O(|V|^3)$. Thus the $B_{fb}(e)$ values can be computed in $O(|V|^3)$ time.

6. 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 DFG 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 present two mechanisms to minimize the synchronization cost — removal of redundant synchronization edges, and conversion of a synchronization graph that is not strongly connected into one that is strongly connected

7. Removing Redundant Synchronizations

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. Fig. 3 shows an example of a redundant synchronization edge. Here, before executing actor D, the processor that executes $\{A, B, C, D\}$ 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 must complete before each invocation of D begins. Thus, x_2 is redundant.

The following theorem establishes that the order in which we remove redundant synchronization

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.

edges is not important.

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 $\tilde{G}_s = (V, E - \{e_1\})$. 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

$$Delay(p) \leq delay(e_2). \tag{12}$$

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

$$Delay(p') \leq delay(e_1). \tag{13}$$

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,

$$Delay(p'') = \sum delay(x_i) + \sum delay(y_i)$$

$$= Delay(p') + (Delay(p) - delay(e_1))$$

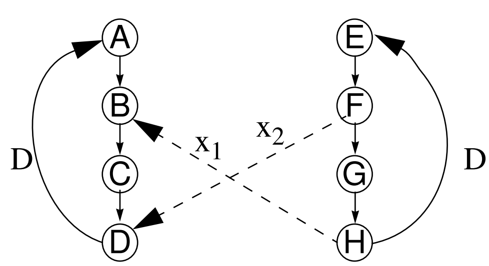


Fig. 3. An example of a redundant synchronization edge.

$\leq Delay(p)$ (from (13))

$\leq delay(e_2)$ (from (12)). *Q. E. D.*

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.

Fig. 4 presents an efficient algorithm, based on the ideas presented above, for optimal removal of redundant synchronization edges. In this algorithm, we first compute the path delay of a minimum-delay

Function RemoveRedundantSynchs
Input: A synchronization graph $G_s = (V, E)$ such that $I \subseteq E$ is the set of synchronization 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 $\rho_{G_s}(x, y)$ 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 $delay(e_o) + \rho_{G_s}(snk(e_o), snk(e)) \leq delay(e)$
        Then
            $E_r = E_r \cup \{e\}$
            Break /* exit the innermost enclosing **For** loop */
        End If
    End For
End For
4. **Return** $(V, (E - E_r))$.

Fig. 4. An algorithm that optimally removes redundant synchronization edges.

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* [6]. Then, we examine each synchronization edge e — in some arbitrary sequence — and determine whether or not there is a path from some successor v of $src(e)$ (other than $snk(e)$) to $snk(e)$ that has a path delay that does not exceed $(delay(e) - delay(src(e), v))$. It is easily verified that this check is equivalent to checking whether or not e is redundant [3].

From the definition of a redundant synchronization edge, it is easily verified that given a redundant synchronization edge e_r in G_s, and two arbitrary vertices $x, y \in V$, if we let $\hat{G}_s = (V, (E - \{e_r\}))$, 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.

In [3], it is shown that *RemoveRedundantSynchs* attains a time complexity of $O(|V|^2 \log_2(|V|) + |V||E|)$ if we use a modification of Dijkstra's algorithm described in [6] for Step 1.

8. Comparison with Shaffer's Approach

In [26], Shaffer presents an algorithm that minimizes the number of directed synchronizations in the self-timed execution of a DFG under the (implicit) assumption that the execution of successive iterations of the DFG 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 connecting the last actor executed on a processor to the first actor executed on the same processor, and edges that have delay are ignored since only intra-iteration dependencies are significant. Thus, Shaffer's synchronization graph is acyclic. *RemoveRedundantSynchs* can be viewed as an extension of Shaffer's algorithm to handle self-timed, iterative execution of a DFG.

Fig. 5 shows a DFG that arises from a four-channel multiresolution QMF filter bank, and Fig. 5(b)

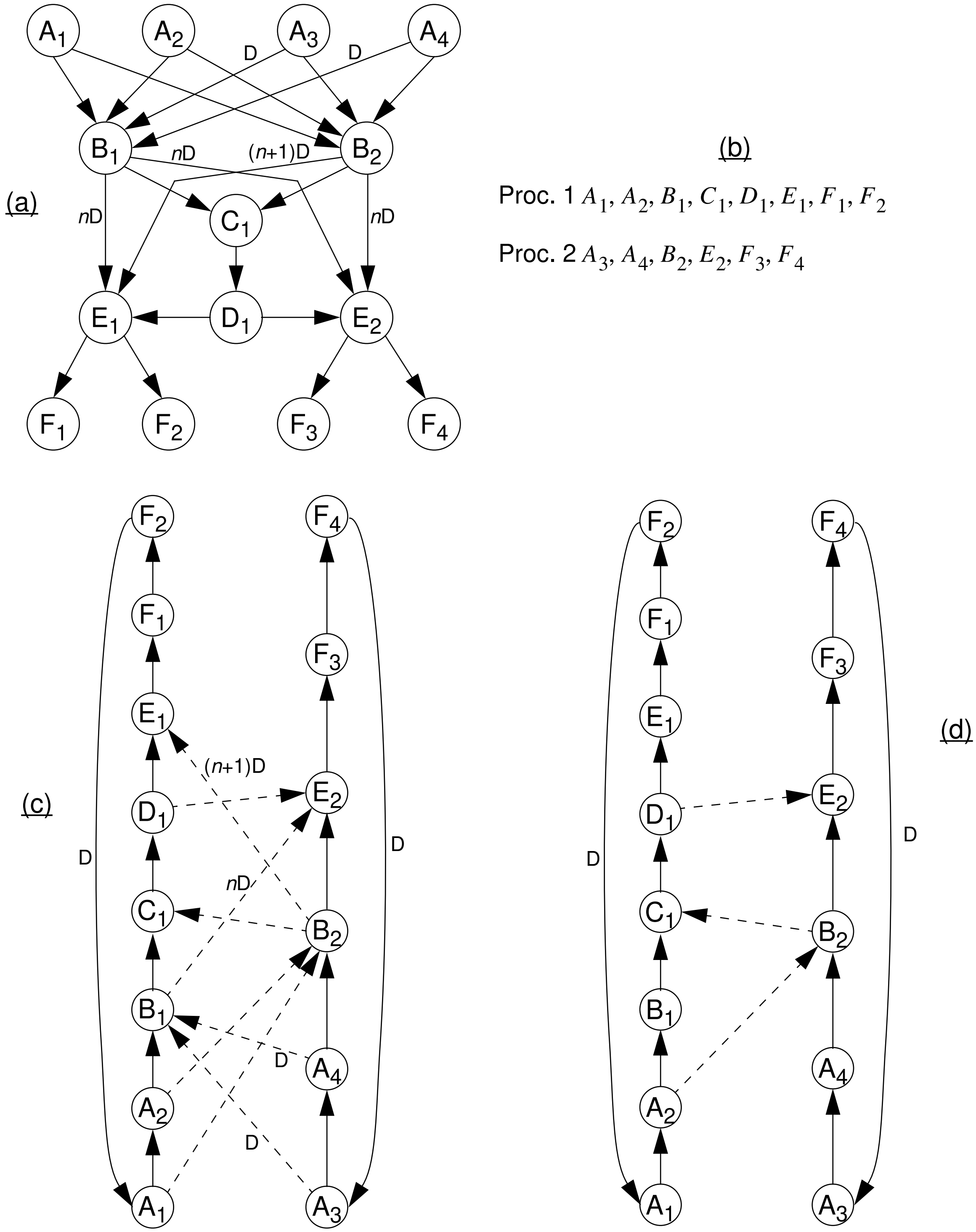


Fig. 5. Application of *RemoveRedundantSynchs* to a multiresolution QMF filter bank

shows a self timed schedule for this DFG. For elaboration on the derivation of this DFG from the original SDF graph see [3, 16]. The synchronization graph that corresponds to Figs. 5(a&b) is shown in Fig. 5(c). The dashed edges 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 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 edges — (A_3, B_1), (A_4, B_1), (B_2, E_1), and (B_1, E_2). The synchronization graph that results from applying *RemoveRedundantSynchs* is shown in Fig. 5(d). The number of synchronization edges is reduced from 8 to 3.

9. Deriving a Strongly Connected Synchronization Graph

Earlier, we defined two synchronization protocols — BBS, which has a cost of 2 synchronization accesses per iteration period, and UBS, which has a cost of 4 synchronization accesses. We pay the increased overhead of UBS whenever the associated edge is a feedforward edge of the synchronization graph G_s.

One alternative to implementing UBS for a feedforward edge e is to add synchronization edges to G_s so that e becomes encapsulated in an SCC; such a transformation would allow e to be implemented with BBS. We have developed an efficient technique to perform such a graph transformation in such a way that the net synchronization cost is minimized, the impact on the self-timed buffer bounds of the IPC edges is optimized, and the estimated throughput is not degraded. This technique is similar in spirit to the one in [30], where the concept of converting a DFG that contains feedforward edges into a strongly connected graph has been studied in the context of retiming.

Fig. 6 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.

The following theorem establishes that a solution computed by *Convert-to-SC-graph* always has a synchronization cost that is no greater than that of the original synchronization graph:

Theorem 4: 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.

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 $d_0(w_m, v_1)$.

Fig. 6. An algorithm for converting a synchronization graph that is not strongly connected into a strongly connected graph.

Proof: Recall that in a connected graph (V^*, E^*), $|E^*|$ must exceed $(|V^*| - 2)$ [6]. Thus, the number of feedforward edges n_f must satisfy $(n_f > n_c - 2)$, where n_c is the number of SCCs. Now, 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, and consequently, 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 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, $(S_+ - S\_) \le 0$. *Q. E. D.*

Fig. 7 shows the synchronization graph topology that 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 black, dashed edges), all of which are feedforward edges, so the synchronization cost is $4n_i = 24$. Since the graph has one source SCC and one sink SCC, only one edge is added by *Convert-to-SC-graph* (shown by the grey, dashed edge), and adding this edge reduces the synchronization cost to $2n_i + 2 = 14$ — a 42% savings.

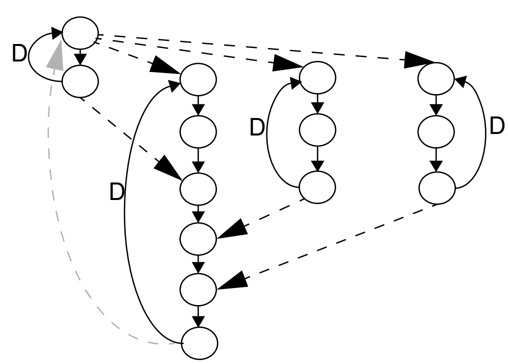


Fig. 7. A solution obtained by *Convert-to-SC-graph* when applied to a 4-processor schedule of a synthesizer for musical instruments based on the Karplus-Strong technique.

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 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 an 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.

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 $G[e_0 \to \Delta_0, \ldots, e_{n-1} \to \Delta_{n-1}]$ denotes the DFG

$(V, ((E - \{e_0, e_1, \ldots, e_{n-1}\}) \cup \{e_0', e_1', \ldots, e_{n-1}'\}))$, 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, $G[e_0 \to \Delta_0, \ldots, e_{n-1} \to \Delta_{n-1}]$ is simply the DFG that results from "changing the delay" on each e_i to the corresponding new delay value Δ_i. Also, if G is a strongly connected 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}).

Fig. 8 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 [15] to return the cycle mean of the critical cycle in Z; if one or more cycles exist that have zero path delay, then *Bell-*

manFord returns ∞.

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 $d_0, d_1, \ldots, d_{m-1}$ 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.

$X_0 = \hat{G}_s[e_0 \to \infty, \ldots, e_{m-1} \to \infty]$

λ_{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 δ_i */

End For

Return $\delta_0, \delta_1, \ldots, \delta_{m-1}$.

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.

Fig. 8. An algorithm for determining the delays on the edges introduced by *Convert-to-SC-graph*. This algorithm assumes the original synchronization graph has only one sink SCC.

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 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

1. See Fig. 8 for the specification of what the e_i s represent.

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 5: 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 Fig. 8; 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 $\hat{G}_s[e_0 \to d_0', \ldots, e_{m-1} \to d_{m-1}']$ has the same estimated throughput as G_s. Then $\Phi(\hat{G}_s[e_0 \to d_0', \ldots, e_{m-1} \to d_{m-1}']) \geq \Phi(\hat{G}_s[e_0 \to d_0, \ldots, e_{m-1} \to d_{m-1}])$, where $\Phi(X)$ is the sum of the self-timed buffer bounds over all IPC edges in G_{ipc} induced by the synchronization graph X.

Fig. 9 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 $\{e_a, e_b\}$. 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; inspection of this cycle yields $\lambda_{max} = 4$. Also, 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 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 Fig. 9 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 $\Phi = 2 + 1 = 3$.

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 argument of Theorem 5, it is easily verified that when it is applicable, this modified algorithm always yields an optimal solution.

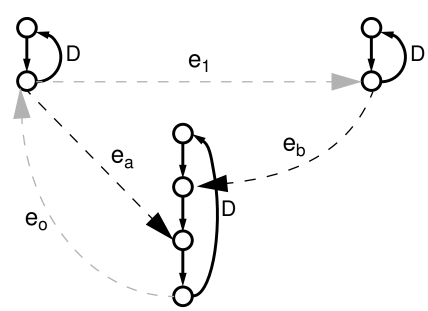


Fig. 9. An example used to illustrate a solution obtained by algorithm *DetermineDelays*.

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. Some fundamental difficulties in deriving such an extension are explained in [3].

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 i th 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*.

It should be noted that practical synchronization graphs frequently contain either a single source SCC or a single SCC, or both — such as the example of Fig. 7. Thus, *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 it can be shown that *DetermineDelays* — and any of the variations of *DetermineDelays* defined above — has $O(|V|^4 (\log_2(|V|))^2)$ time complexity.

Although the issue of deadlock does not explicitly arise in *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, the algorithm cannot convert a graph that is not deadlocked into a deadlocked graph.

10. Complete Algorithm

In this section we outline our complete synchronization optimization algorithm. The input is a DFG and a parallel schedule for it, and the output 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-memory buffer sizes $\{B_{fb}(e) | e \text{ is an IPC edge in } G_{ipc}\}$, which specifies the amount of memory to allocate in shared memory for each IPC edge.

The pseudocode for the complete algorithm is given in Fig. 10. 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.

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

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 $G_s = G_{ipc}$ /* Initially, each IPC edge is also a synchronization edge */

3. $G_s = RemoveRedundantSynchs(G_s)$

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

5. $G_s = DetermineDelays(G_s)$

/* Remove the synchronization edges that have become redundant as a result of Step 4. */
6. $G_s = RemoveRedundantSynchs(G_s)$

7. Calculate buffer sizes $B_{fb}(e)$ for each IPC edge e in G_{ipc} (to be used for BBS):
— Compute $\rho_{G_s}(snk(e), src(e))$, and set $B_{fb}(e) = \rho_{G_s}(snk(e), src(e)) + delay(e)$.

Fig. 10. The complete synchronization optimization algorithm.

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 BBS. The synchronization cost in the final implementation is equal to $2n_s$, where n_s is the number of synchronization edges in G_s.

11. Conclusions

We have presented techniques to reduce synchronization overhead in self-timed, multiprocessor implementations of 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 a new method for reducing synchronization overhead that converts a feedforward DFG into a strongly connected graph in such a way as to reduce synchronization overhead without slowing down execution. 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.

Perhaps the most significant direction for further work is the incorporation of timing guarantees — for example, hard upper and lower execution time bounds, as Dietz, Zaafrani, and O'keefe use in [7]; 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 [8].

References

[1] S. Banerjee, D. Picker, D. Fellman, 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, G. Berry, "The Synchronous Approach to Reactive and Real-Time Systems," *Proceedings of the IEEE*, September, 1991.

[3] S. S. Bhattacharyya, S. Sriram, E. A. Lee, *Optimizing Synchronization in Multiprocessor Implementations of Iterative Dataflow Programs*, Memorandum No. UCB/ERL 95/2, University of California at Berkeley, January, 1995. WWW URL: http://ptolemy.eecs.berkeley.edu/~ptdesign/Ptolemy/papers/synch_optimization.ps.Z.

[4] J. T. Buck, S. Ha, E. A. Lee, D. G. Messerschmitt, "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems," *Intl. Jo. of Computer Simulation*, 1994.

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

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

[7] H. G. Dietz, A. Zaafrani, M. T. O'keefe, "Static Scheduling for Barrier MIMD Architectures," *Jo. of Supercomputing*, February, 1992.

[8] D. Filo, D. C. Ku, C. N. Coelho Jr., G. De Micheli, "Interface Optimization for Concurrent Systems Under Timing Constraints," *IEEE Trans. on VLSI Systems*, September, 1993.

[9] R. Govindarajan, G. R. Gao, P. Desai, "Minimizing Memory Requirements in Rate-Optimal Schedules," *Proc. of the Intl. Conf. on Application Specific Array Processors*, August, 1994.

[10] J. A. Huisken et al., "Synthesis of Synchronous Communication Hardware in a Multiprocessor Architecture," *Jo. of VLSI Signal Processing*, December, 1993.

[11] A. Kalavade, E. A. Lee, "A Hardware/Software Codesign Methodology for DSP Applications," *IEEE Design and Test*, September 1993.

[12] 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.

[13] S. Y. Kung, P. S. Lewis, S. C. Lo, "Performance Analysis and Optimization of VLSI Dataflow Arrays," *Jo. of Parallel and Distributed Computing*, December, 1987.

[14] R. Lauwereins, M. Engels, J.A. Peperstraete, E. Steegmans, J. Van Ginderdeuren, "GRAPE: A CASE Tool for Digital Signal Parallel Processing," *IEEE ASSP Magazine*, April, 1990.

[15] E. Lawler, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, 1976.

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

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

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

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

[20] K. K. Parhi, D. G. Messerschmitt, "Static Rate-Optimal Scheduling of Iterative Data-Flow Programs via Optimum Unfolding," *IEEE Trans. on Computers*, February, 1991.

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

[22] J. Pino, S. Ha, E. A. Lee, J. T. Buck, "Software Synthesis for DSP Using Ptolemy," *Jo. of VLSI Signal Processing*, January, 1995.

[23] 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.

[24] R. Reiter, "Scheduling Parallel Computations," *Jo. of the Association for Computing Machinery*, October 1968.

[25] S. Ritz, M. Pankert, H. Meyr, "High Level Software Synthesis for Signal Processing Systems," *Proc. of the Intl. Conf. on Application Specific Array Processors*, August, 1992.

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

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

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

[29] S. Sriram, E. A. Lee, "Design and Implementation of an Ordered Memory Access Architecture," *Proc. of the Intl. Conf. on Acoustics Speech and Signal Processing*, April, 1993.

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