— Appeared in the *Proceedings of Intl. Conference on Application Specific Array Processors*, 1995 —

# Minimizing Synchronization Overhead in Statically Scheduled Multiprocessor Systems

Shuvra S. Bhattacharyya†, Sundararajan Sriram‡, and Edward A. Lee‡
April 27, 1995

## Abstract

Synchronization overhead can significantly degrade performance in embedded multiprocessor systems. In this paper, we develop techniques to determine a minimal set of processor synchronizations that are essential for correct execution in an embedded multiprocessor implementation. Our study is based in the context of *self-timed* execution of *iterative dataflow* programs; dataflow programming in this form has been 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 also present an optimization that involves converting a synchronization graph that is not strongly connected into a strongly connected graph.

## 1. Introduction

This paper addresses 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 popularity of the synchronous dataflow (SDF) model in DSP design environments [5, 10, 11, 13]; the suitability of this model for exploiting parallelism; and the high overhead of run-time synchronization in embedded multiprocessor implementations. Our work is particularly relevant when estimates are available for the task execution times, and actual execution times are close to the corresponding estimates with high frequency, but deviations from the estimates of (effectively) arbitrary magnitude can occasionally occur due to phenomena such as cache misses or error handling.

In SDF, a program is represented as a directed graph in which the vertices (**actors**) rep-

---

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.

†Semiconductor Research Laboratory, Hitachi America, Ltd., 201 East Tasman Drive., San Jose, California 95134, USA.

‡Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, California 94720, USA.

resent computations, and the edges specify data dependences. In SDF the number of data values (**tokens**) produced (consumed) by each actor onto (from) each of its output (input) edges is fixed and known at compile time. The techniques developed in this paper assume that the input SDF graph is *homogeneous*, which means that the numbers of tokens 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 [6], 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 $k$th 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 Figure 1).

Multiprocessor implementation of an algorithm specified as a DFG involves scheduling the actors. 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. In [7] 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.

As a performance metric for evaluating schedules 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 $T$.

In the **fully-static** scheduling strategy of [7], 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. This results in the **self-timed** scheduling strategy [7]. 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.

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 this paper, we 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.

Numerous research efforts have focused on constructing efficient parallel schedules for DFGs. Some of these develop systematic techniques for exploiting overlapped execution [8]; other work has focused on taking IPC costs into account during scheduling [7, 11, 15]; and in [4], the authors 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* concept, discussed in [3]. 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 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**) [1].

In [14], Shaffer presents an algorithm that minimizes the number of directed synchronizations in the self-timed execution of a DFG. However, this work 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.

## 2. Background Terminology and Notation

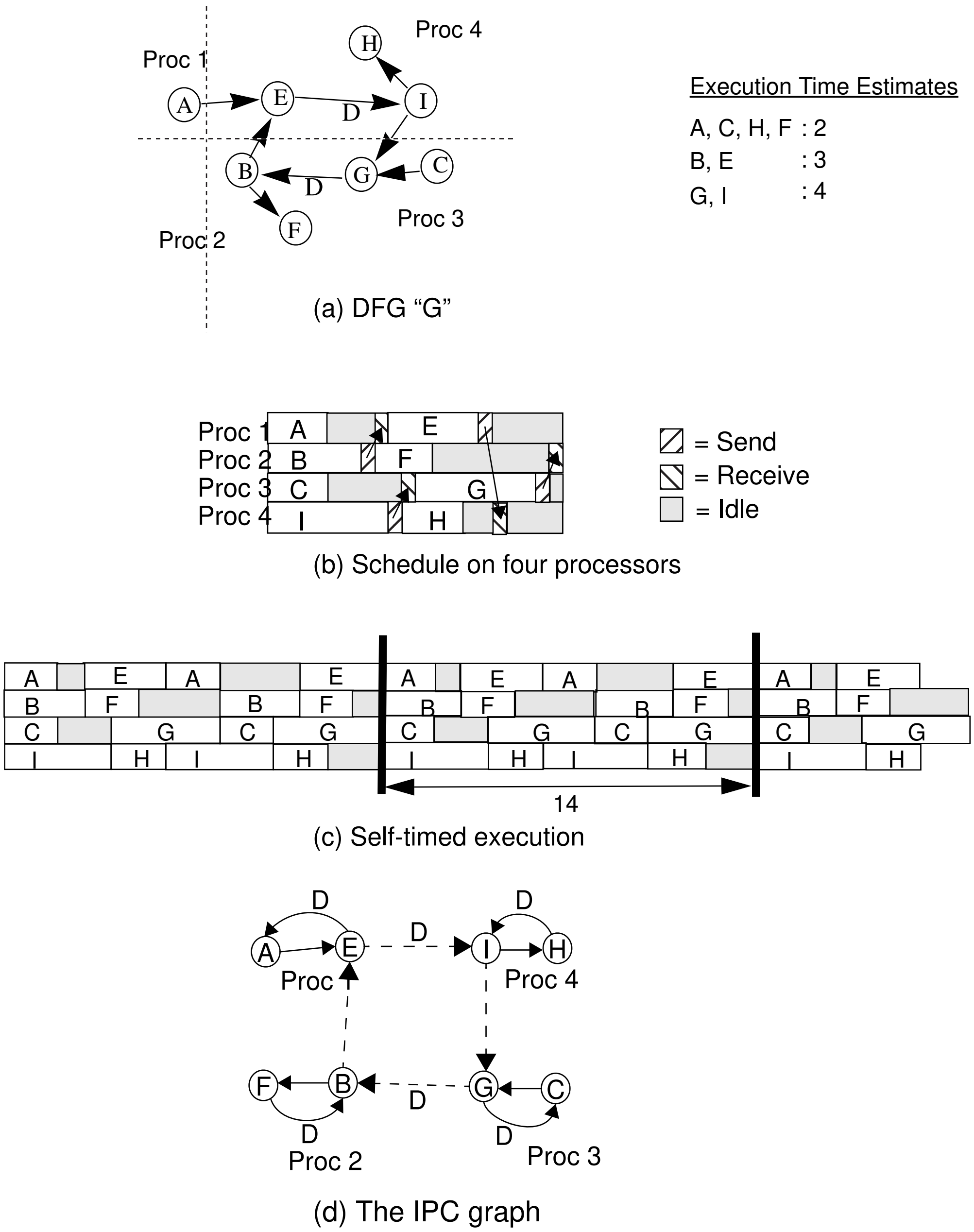
We represent a DFG by an ordered pair $(V, E)$, where $V$ is the set of vertices and $E$ is the set of edges. We denote the source vertex, sink vertex, and the delay of an edge $e$ by $src(e)$, $snk(e)$, and $delay(e)$. A **path** in $(V, E)$ is a finite, nonempty sequence $(e_1, e_2, \ldots, e_n)$, 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 a **cycle**.

The **path delay** of $p = (e_1, e_2, \ldots, e_n)$, denoted $Delay(p)$, is defined by $Delay(p) = \sum_{i=1}^{n} delay(e_i)$. For two vertices $x, y \in V$, either there is no path directed from $x$ to $y$, or there exists a **minimum-delay path** from $x$ to $y$. Given a DFG $G$, and vertices $x, y$ in $G$, we define $\rho_G(x, y)$ to be equal to $\infty$ 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 $y$.

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.

## 3. Analysis of Self-Timed Execution

Figure 1(c) illustrates the self-timed execution of the four-processor schedule in Figure 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.

### 3.1 IPC Graph $G_{ipc} = (V, E_{ipc})$

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 **IPC graph**, models the sequential execution of the actors of $G$ assigned to the same processor, and it models constraints due to IPC.

The IPC graph has the same vertex set $V$ as $G$. The self-timed schedule specifies the actors assigned to each processor, and the order in which they execute. For example in Figure 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 $k$th execution of $A$ precedes the $k$th execution of $E$, and the edge from $E$ to $A$ with a delay represents the fact that the $k$th 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 $delay((v_n, v_1)) = 1$. For example, the self-timed schedule in Figure 1(c) can be modelled by the IPC graph in Figure 1(d). The IPC edges are shown using dashed arrows.

Edges in $G$ that cross processor boundaries after scheduling represent IPC; we call such edges **IPC edges**. Instead of explicitly introducing special *send* and *receive* primitives at the ends of IPC edges, we will model these operations as part of the sending and receiving actors themselves. This is done to avoid cluttering $G_{ipc}$ 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 $delay(e)$ equal to the delay on the corresponding edge in $G$. Thus in Figure 1 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.

By the function $start(v, k) \in Z^+$ (non-negative integer) we represent the time at which the $k$th execution of the actor $v$ starts in the self-timed schedule; and by $end(v, k) \in Z^+$ we represent the time at which the $k$th execution of $v$ ends. We set $start(v, k) = 0$ and $end(v, k) = 0$ for $k \le 0$. Here, time is modelled as an integer that can be considered a multiple of a base clock.

The edges $(v_j, v_i)$ of $G_{ipc}$ represent the following constraints:

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

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 an execution time $t(v) \in Z^+$ with each $v \in V$; $t(v)$ includes the time taken to execute all IPC operations (*send*s and *receive*s) that the actor $v$ performs.

The IPC graph can also be considered to be a timed Marked graph [9] or Reiter's computation graph [12].

**Lemma 1:** 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 [12]:

$$T = \max_{\text{cycle } C \text{ in } G} \left\{ \frac{\sum_{v \text{ is on } C} t(v)}{Delay(C)} \right\}. \quad (2)$$

Note that $Delay(C) > 0$ if the given schedule is deadlock free [12].

The quantity on the RHS of (2) 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 SCC in the system. Henceforth, the **maximum cycle mean** of an IPC graph $G_{ipc}$, denoted by $\lambda_{max}$, will refer to the maximal cycle mean over all SCCs of $G_{ipc}$. For example, in Figure 2, $G_{ipc}$ has $\lambda_{max} = 7$.

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 $\lambda_{max}$. In the transformations that we present in the rest of the paper, we ensure that we do not alter the **estimated throughput** $\frac{1}{\lambda_{max}}$. We ensure this by preserving the maximum cycle mean of $G_{ipc}$.

### 3.2 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. The following lemma states that each feedback edge has a buffer requirement that is bounded by some 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:** [1] The number of tokens on a feedback edge $e$ of $G_{ipc}$ is bounded; an upper bound is given by

$$B_{fb}(e) = min(\{Delay(C) \mid C \text{ is a cycle that contains } e\}). \quad (3)$$

We show how to calculate these buffer bounds efficiently later in the paper.

A feedforward edge, however, has no such bound on the buffer size, because it is not contained in any cycle. In Section 6, we present a method for making a DFG strongly connected, which makes all buffers bounded because each edge becomes a feedback edge.

## 4. Synchronization Model

### 4.1 Synchronization Protocols

In [1], we define two synchronization protocols for an IPC edge. Given an IPC graph $(V, E)$, and an IPC edge $e \in E$, if $e$ is a feedforward edge then we apply a synchronization protocol called **unbounded buffer synchronization (UBS)**, which guarantees that $snk(e)$ never attempts to read data from an empty buffer (to prevent underflow), and $src(e)$ never attempts to write data into the buffer unless the number of tokens already in the buffer is less than some pre-specified limit, which is the amount of memory allocated to that buffer (to prevent overflow). This involves maintaining a count of the number of tokens currently in the buffer in a shared memory location. This count must be examined and updated by each invocation of $src(e)$ and $snk(e)$, and thus in each graph iteration period, UBS requires an average of four accesses to shared memory (two read accesses and two write accesses)[1]. We refer to these accesses to shared memory, which are performed solely for the purpose of synchronization, as **synchronization accesses**.

If $e$ is a feedback edge, then we use a simpler protocol, called **bounded buffer synchronization (BBS)**, that only explicitly ensures that overflow does not occur. In this protocol, rather than maintaining the token count in shared memory, we maintain a copy of the *write pointer* into the buffer (of the source actor). After each invocation of $src(e)$, the write pointer is updated locally (on the processor that executes $src(e)$), and the new value is written to shared memory. It is easily verified that to prevent underflow, it suffices to block each invocation of the sink actor until the *read pointer* (maintained locally on the processor that executes $snk(e)$) is found to be not equal to the current value of the write pointer. Thus, in BBS, $src(e)$ never reads the shared memory location associated with the write pointer for $e$, and $snk(e)$ never writes to this location, and consequently, only two synchronization accesses are required per iteration period.

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.

## 4.2 The Synchronization Graph $G_s = (V, E_s)$

An IPC edge in $G_{ipc}$ represents two functions: reading and writing of tokens into the buffer represented by that edge, and synchronization between the sender and the receiver, which could be implemented with UBS or BBS. To differentiate these two functions, we define 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 3.1 that an IPC edge $(v_j, v_i)$ of $G_{ipc}$ represents the **synchronization constraint**:

$$start(v_i, k) \geq end(v_j, k - delay((v_j, v_i)))\ \forall k > delay(v_j, v_i)\ . \tag{4}$$

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 (defined shortly) by adding/deleting edges. At the end of our optimizations, the synchronization graph 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 corresponding buffer is 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.

The following theorem underlies the validity of our synchronization optimizations.

---

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 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.

**Theorem 1:** [1] The synchronization constraints in a synchronization graph $G_1 = (V, E_1)$ imply the constraints of the graph $G_2 = (V, E_2)$ if $\forall \varepsilon \in E_2, \varepsilon \notin E_1$, $\rho_{G_1}(src(\varepsilon), snk(\varepsilon)) \le delay(\varepsilon)$, that is if for each edge $\varepsilon$ 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 $delay(\varepsilon)$.

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 $\rho_{G_1}(src(\varepsilon), snk(\varepsilon)) \le delay(\varepsilon)$.

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 maximum cycle mean of $G_s$.

### 4.3 Problem Statement

Recall that if synchronization for an edge $e$ is implemented using UBS, then $4$ synchronization accesses are required per iteration period, while BBS implies $2$ synchronization accesses. The **synchronization cost** of a synchronization graph $G_s$ is the 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 feedback synchronization edges, then the synchronization cost of $G_s$ is $(4n_{ff} + 2n_{fb})$.

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.

## 5. Removing Redundant Synchronizations

Formally, a synchronization edge is **redundant** in a synchronization graph $G$ if its removal yields a graph that preserves $G$. Equivalently, a synchronization edge $e$ is redundant if there is a path $p \ne (e)$ from $src(e)$ to $snk(e)$ such that $Delay(p) \le delay(e)$.

Thus, the synchronization function associated with a redundant synchronization edge "comes for free" as a by product of other synchronizations. Figure 2 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 $F$ is guaranteed to

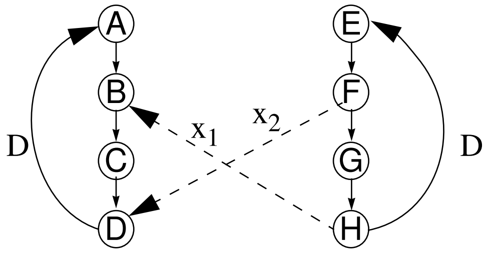



Figure 2. An example of a redundant synchronization edge.

complete before each invocation of $D$ is begun. Thus $x_2$ is redundant.

## 5.1 Algorithm for Removing Redundant Synchronization Edges

The following result establishes that the order in which we remove redundant synchronization edges is not important.

**Theorem 2:** [1] Suppose $G_s = (V, E)$ 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 $\tilde{G}_s$.

Theorem 2 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.

Figure 3 presents an efficient algorithm for optimal removal of redundant synchronization edges. In this algorithm, we first compute $\rho_{G_s}(x, y)$ for each ordered pair of vertices $(x, y)$. Then, we examine each synchronization edge $e$ 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 $delay(e)$. It can be shown that such a path exists iff $e$ is a redundant edge.

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

**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. $E_r \leftarrow \varnothing$
3. **For** each $e \in I$

    **For** each output edge $e_o$ of $src(e)$ except for $e$

        **If** $delay(e_o) + \rho_{G_s}(snk(e_o), snk(e)) \le delay(e)$

        **Then**

            $E_r \leftarrow E_r \cup \{e\}$

            Break /* exit the innermost enclosing **For** loop */

        **Endif**

    **Endfor**

**Endfor**

4. **Return** $(V, (E - E_r))$.

Figure 3. An algorithm that optimally removes redundant synchronization edges.

$\hat{G}_s = \left( V, \left( E - \{ e_r \} \right) \right)$, 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.

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

### 5.2 Comparison with Shaffer's Approach

In [14], 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; 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. We present an example next.

Figure 4 (a) shows a DFG that arises from a four-channel multiresolution QMF filter bank, and Figure 4(b) shows a self timed schedule for this DFG. For elaboration on the derivation of this DFG from the original SDF graph see [1, 6]. The synchronization graph that corresponds to Figures 4(a&b) is shown in Figure 4(c). The dashed edges are synchronization edges. If we apply *RemoveRedundantSynchs*, we obtain the synchronization graph in Figure 4(d): the edges $(A_1, B_2)$, $(A_3, B_1)$, $(A_4, B_1)$, $(B_2, E_1)$, and $(B_1, E_2)$ are detected to be redundant, and the number of synchronization edges is reduced from $8$ to $3$ as a result.

## 6. Making the Synchronization Graph Strongly Connected

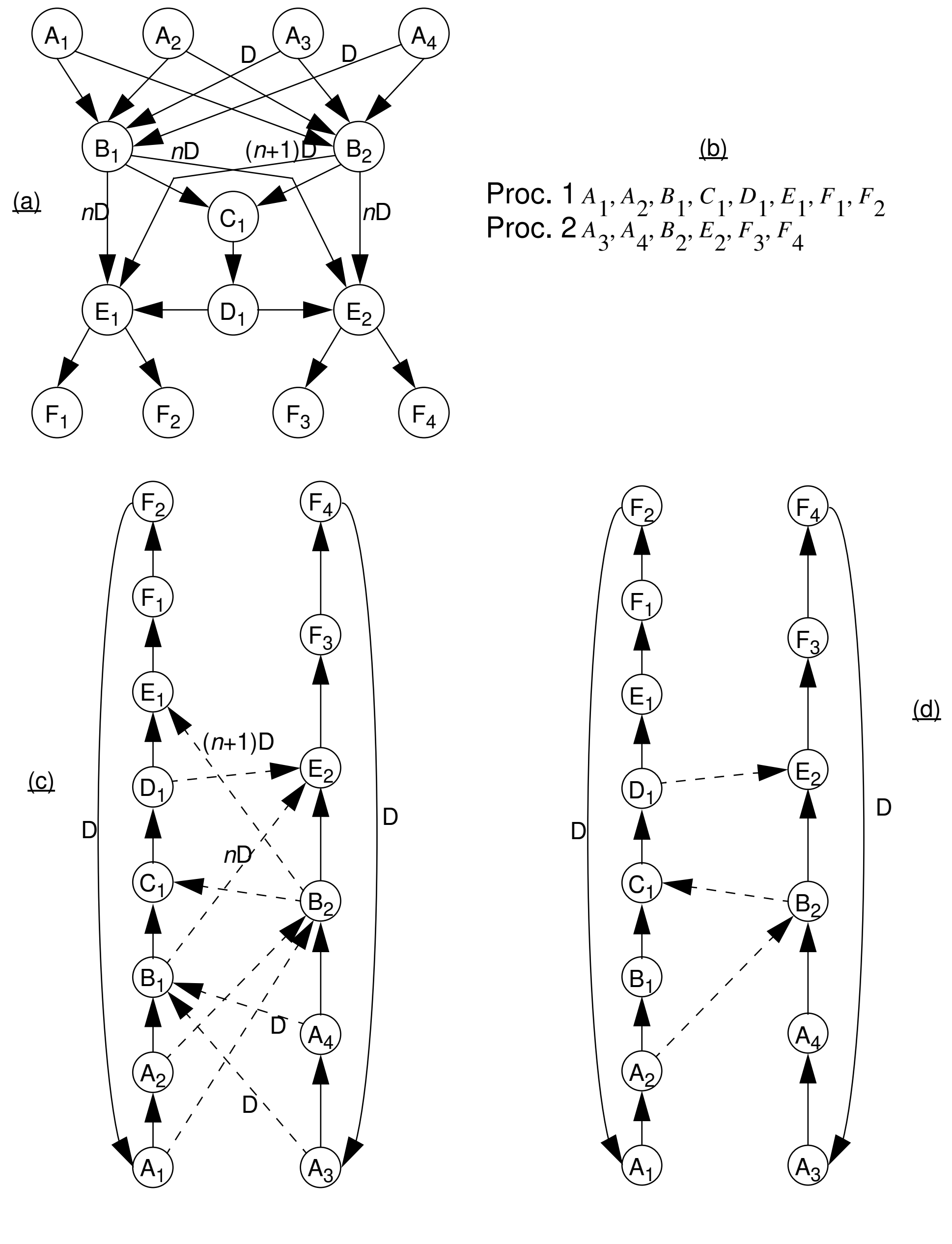
Earlier, we defined two synchronization protocols — BBS, which has a cost of 2 synchronization accesses per iteration period, and UBS, which has a cost of 4 synchronization accesses. We pay the increased overhead of UBS whenever the associated edge is a feedforward edge of the synchronization graph $G_s$.

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 [17], where the concept of converting a DFG that contains feedforward edges into a strongly connected graph has been studied in the context of retiming.

Our algorithm for transforming a synchronization graph that is not strongly connected into a strongly connected graph, called *Convert-to-SC-graph*, simply "chains together" the source SCCs, chains together the sink SCCs, and then connects the first SCC of the "source chain" to the last SCC of the sink chain with an edge. From each source or sink SCC, the algorithm selects a vertex that has minimum execution time to be the corresponding chain "link." Minimum execution time vertices are chosen in an attempt to minimize the delay that must be inserted on the new edges to preserve the estimated throughput. In [1], we prove that the solution computed by *Convert-to-SC-graph* always has a synchronization cost that is that no greater than that of the original synchronization graph.

Figure 5 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 $2n_i + 2 = 14$.

Since the conversion of a non-strongly-connected $G_s$ to a strongly connected graph necessarily must introduce one or more new cycles, and since in general, a new cycle may be




\pplication of *RemoveRedundantSynchs* to a multiresolution QMF filter bank.

delay-free, or its cycle mean may exceed that of the maximum cycle mean of $G_s$, 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 Lemma 2) they affect the self-timed buffer bounds of the IPC edges, which in turn determine the amount of memory that we allocate for the corresponding buffers. Thus, 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.

We have a developed an efficient algorithm, called *DetermineDelays*, that addresses this goal. In [1], we show that if $G_s$ has only one source SCC or only one sink SCC; $\hat{G}_s$ denotes the synchronization graph that results from applying *DetermineDelays* to compute the delays on the edges $(e_1, e_2, \ldots, e_n)$ introduced by *Convert-to-SC-graph*; and $G_s'$ is any synchronization graph that corresponds to an alternative assignment of delays to $(e_1, e_2, \ldots, e_n)$ and preserves the estimated throughput, then $\Phi\left(\hat{G}_s\right) \le \Phi(G_s')$, where $\Phi(X)$ denotes the sum of the self-timed buffer bounds over all IPC edges induced by the synchronization graph $X$. In practice, the assumptions under which this optimality result applies are frequently satisfied. For further details on algorithm *DetermineDelays*, see [1].

## 7. Computing Buffer Bounds from $G_s$ and $G_{ipc}$

After all the optimization steps in Section 5 and 6 are complete we have a strongly connected synchronization graph $G_s = (V, (E_{ipc} - F + F'))$ that preserves $G_{ipc}$. All the synchronization edges of this graph can now be implemented using the BBS protocol. To do this, we need to compute the self-timed buffer bounds for these edges; the following theorem tells us how to compute these bounds from $G_s$.

**Theorem 3:** [1] 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 tokens that can ever be present on $e$ — is given by: $B_{fb}(e) = \rho_{G_s}(snk(e), src(e)) + delay(e)$.

The quantities $\rho_{G_s}(snk(e), src(e))$ can be computed efficiently using Dijkstra's algorithm [2] to solve the all-pairs shortest path problem on the synchronization graph.

## 8. Summary

We have addressed the problem of minimizing synchronization overhead when imple-

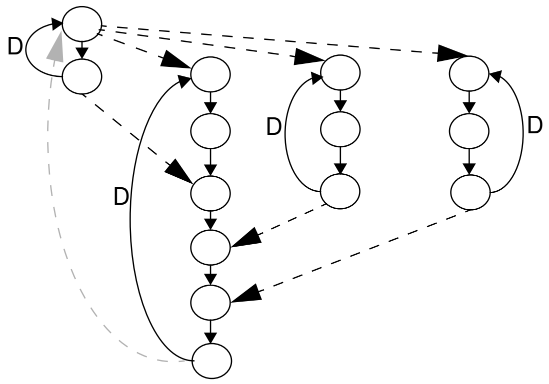



Figure 5. An illustration of the *Convert-to-SC-graph* algorithm.

menting 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 a new method for reducing synchronization overhead — the conversion of the synchronization graph into a strongly connected graph. We have demonstrated the relevance of our techniques through practical examples.

The input to our algorithm is a DFG and a parallel schedule for it. 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 $\{B_{fb}(e) \mid e \text{ is an IPC edge in } G_{ipc}\}$. Figure 6 specifies the complete algorithm.

A code generator can then accept $G_{ipc}$ and $G_s$, 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 resulting synchronization cost is $2n_s$, where $n_s$ is the number of synchronization edges in $G_s$.

## References


[1] S. S. Bhattacharyya, S. Sriram, and E. A. Lee, *Optimizing Synchronization in Shared Memory Implementations of Iterative Dataflow Programs*, Memorandum UCB/ERL M95/2, Electronics Research Laboratory, U. C. Berkeley, January 1995.

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

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

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

[5] 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*, April, 1990.


**Function** *MinimizeSynchCost*

**Input:** A DFG $G$ and a self-timed schedule $S$ 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 $S$
2. $G_s \leftarrow G_{ipc}$ /* Each IPC edge is also a synchronization edge to begin with */
3. $G_s \leftarrow \mathit{Convert\text{-}to\text{-}SC\text{-}graph}(G_s)$
4. $G_s \leftarrow \mathit{DetermineDelays}(G_s)$
5. $G_s \leftarrow \mathit{RemoveRedundantSynchs}(G_s)$
6. Calculate the buffer size $B_{fb}(e)$ for each IPC edge $e$ in $G_{ipc}$.
   a) Compute $\rho_{G_s}(src(e), snk(e))$
   b) $B_{fb}(e) \leftarrow \rho_{G_s}(src(e), snk(e)) + delay(e)$

Figure 6. The complete synchronization optimization algorithm.

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

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

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

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

[10] J. Pino, S. Ha, E. A. Lee, and J. T. Buck, “Software Synthesis for DSP Using Ptolemy,” *Journal of VLSI Signal Processing*, January, 1995, to appear.

[11] 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.

[12] R. Reiter, Scheduling Parallel Computations, *Journal of the Association for Computing Machinery*, October, 1968.

[13] S. Ritz, M. Pankert, and H. Meyr, “High Level Software Synthesis for Signal Processing Systems,” *Proc. of Intl. Conf. on Application Specific Array Processors*, Berkeley, August, 1992.

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

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

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

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