5

MINIMIZING SYNCHRONIZATION COSTS IN SELF-TIMED SCHEDULES

The previous three chapters dealt with the Ordered Transactions strategy, which is a hardware approach to reducing IPC and synchronization costs in self-timed schedules. In this chapter we present algorithms that minimize synchronization costs in the final implementation of a given self-timed schedule, and we do not assume the availability of any hardware support for employing the OT approach.

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

In each type of platform, each IPC that requires a synchronization check costs performance, and sometimes extra hardware complexity. Semaphore checks cost execution time on the processors, synchronization instructions that make use of special synchronization hardware such as barriers also cost execution time, and blocking interfaces between a programmable processor and custom hardware in a combined hardware/software implementations require more hardware than non-blocking interfaces [Huis93].

In this chapter we present algorithms and techniques that reduce the rate at which processors must access shared memory for the purpose of synchronization in multiprocessor implementations of SDF programs. One of the procedures we present, for example, detects when the objective of one synchronization operation is guaranteed as a side effect of other synchronizations in the system, thus enabling us to eliminate such superfluous synchronization operations. The optimization procedure that we propose can be used as a post-processing step in any static scheduling technique (any one of the techniques presented in Chapter 1, section 1.2) for reducing synchronization costs in the final implementation. As before we assume that "good" estimates are available for the execution times of actors and that these execution times rarely display large variations so that self-timed scheduling is viable for the applications under consideration. If additional timing information is available, such as guaranteed upper and lower bounds on the execution times of actors, it is possible to use this information to further optimize synchronizations in the schedule. However, use of such timing bounds will be left as future work; we mention this again in Chapter 7.

This chapter is a part of ongoing research in collaboration with Dr. Shuvra Bhattacharyya, who is a Research Scientist at Hitachi America Ltd.

5.1 Related work

Among the prior art that is most relevant to this chapter is the *barrier-MIMD* principle of Dietz, Zaafrani, and O'Keefe, which is a combined hardware and software solution to reducing run-time synchronization overhead [Dietz92]. In this approach, a shared-memory MIMD computer is augmented with hardware support that allows arbitrary subsets of processors to synchronize precisely with respect to one another by executing a synchronization operation called a *barrier*. If a subset of processors is involved in a barrier operation, then each processor in this subset will wait at the barrier until all other processors in the subset have reached the barrier. After all processors in the subset have reached the barrier, the corresponding processes resume execution in *exact synchrony*.

In [Dietz92], the barrier mechanism is applied to minimize synchronization overhead in a self-timed schedule with hard lower and upper bounds on the task execution times. The execution time ranges are used to detect situations where the earliest possible execution time of a task that requires data from another processor is guaranteed to be later than the latest possible time at which the required data is produced. When such an inference cannot be made, a barrier is instantiated between the sending and receiving processors. In addition to performing the required data synchronization, the barrier resets (to zero) the uncertainty between the relative execution times for the processors that are involved in the barrier, and thus enhances the potential for subsequent timing analysis to eliminate the need for explicit synchronizations.

The techniques of barrier MIMD do not apply to the problem that we address because they assume that a hardware barrier mechanism exists; they assume that tight bounds on task execution times are available; they do not address iterative, self-timed execution, in which the execution of successive iterations of the dataflow graph can overlap; and even for non-iterative execution, there is no obvious correspondence between an optimal solution that uses barrier synchronizations and an optimal solution that employs decoupled synchronization checks at the sender and receiver end (**directed synchronization**). This last point is illustrated in Fig. 5.1. Here, in the absence of execution time bounds, an optimal application of barrier synchronizations can be obtained by inserting two barriers — one barrier across A_1 and A_3, and the other barrier across A_4 and A_5. This is illustrated in Figure 5.1(c). However, the corresponding collection of directed synchronizations (A_1 to A_3, and A_5 toA_4) is not sufficient since it does not guarantee that the data required by A_6 from A_1 is available before A_6 begins execution.

In [Sha89], Shaffer presents an algorithm that minimizes the number of directed synchronizations in the self-timed execution of a dataflow graph. However, this work, like that of Dietz *et al.*, does not allow the execution of successive iterations of the dataflow graph to overlap. It also avoids having to consider dataflow edges that have delay. The technique that we present for removing redundant synchronizations can be viewed as a generalization of Shaffer's algorithm to handle delays and overlapped, iterative execution, and we will discuss this further in section 5.6. The other major techniques that we present for optimizing synchronization — handling the feedforward edges of the *synchronization graph* (to be defined in section 5.4.2), discussed in section 5.7, and "resynchronization", defined and addressed in sections 5.9 and the appendix — are fundamentally different from Shaffer's technique since they address issues that are specific to our

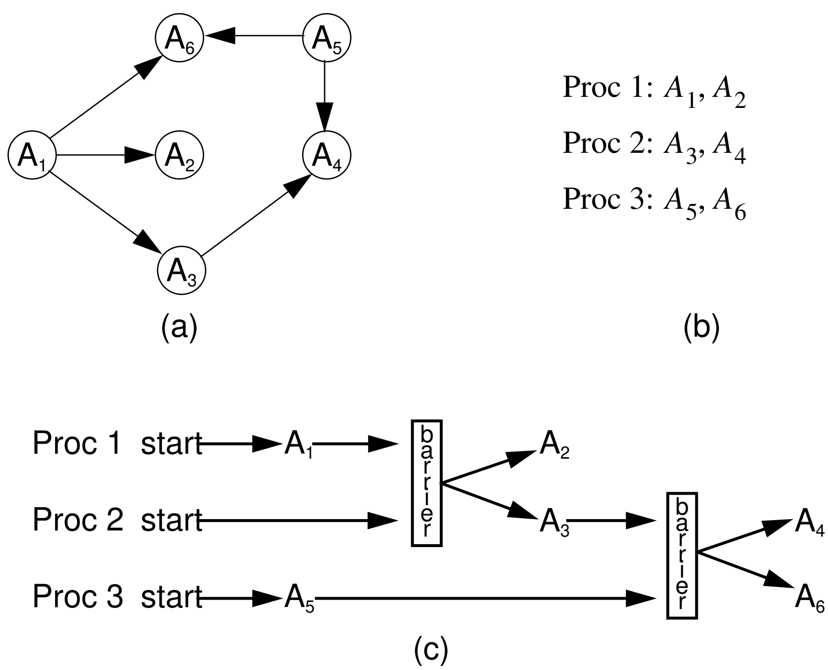


Figure 5.1. (a) An HSDFG (b) A three-pro(a) An HSDFG (b) A three-processor self-timed schedule for (a). (c) An illustration of execution under the placement of barriers.

more general context of overlapped, iterative execution.

As discussed in Chapter 1, section 1.2.2, a multiprocessor executing a self-timed schedule is one where each processor is assigned a sequential list of actors, some of which are *send* and *receive* actors, which it executes in an infinite loop. When a processor executes a communication actor, it synchronizes with the processor(s) it communicates with. Thus exactly when a processor executes each actor depends on when, at run time, all input data for that actor is available, unlike the fully-static case where no such run time check is needed. In this chapter we use "processor" in slightly general terms: a processor could be a programmable component, in which case the actors mapped to it execute as software entities, or it could be a hardware component, in which case actors assigned to it are implemented and execute in hardware. See [Kala93] for a discussion on combined hardware/software synthesis from a single dataflow specification. Examples of application-specific multiprocessors that use programmable processors and some form of static scheduling are described in [Bork88][Koh90], which were also discussed in Chapter 1, section 1.3.

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

Sender-receiver synchronization is also assumed to take place by setting flags in shared memory. Special hardware for synchronization (barriers, semaphores implemented in hardware, etc.) would be prohibitive for the embedded multiprocessor machines for applications such as DSP that we are considering. Interfaces between hardware and software are typically implemented using memory-mapped registers in the address space of the programmable processor (again a kind of shared memory), and synchronization is achieved using flags that can be tested and set by the programmable component, and the same can be done by an interface controller on the hardware side [Huis93].

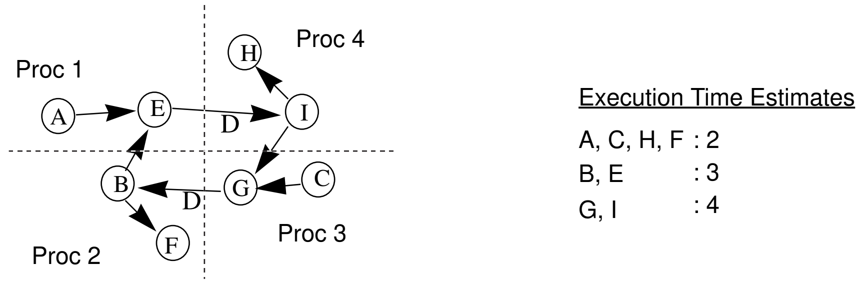
Under the model above, the benefits that our proposed synchronization optimization techniques offer become obvious. Each synchronization that we eliminate directly results in one less synchronization check, or, equivalently, one less shared memory access. For example, where a processor would have to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

Thus the metric for the optimizations we present in this chapter is the total number of accesses to shared memory that are needed for the purpose of synchronization in the final multiprocessor implementation of the self-timed schedule. This metric will be defined precisely in section 5.5.

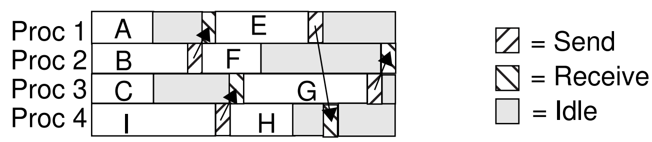
5.2 Analysis of self-timed execution

We model synchronization in a self-timed implementation using the IPC graph model introduced in the previous chapter. As before, an IPC graph $G_{ipc}(V, E_{ipc})$ is extracted from a given HSDFG G and multi-processor schedule; Fig. 5.2 shows one such example, which we use throughout this chapter.

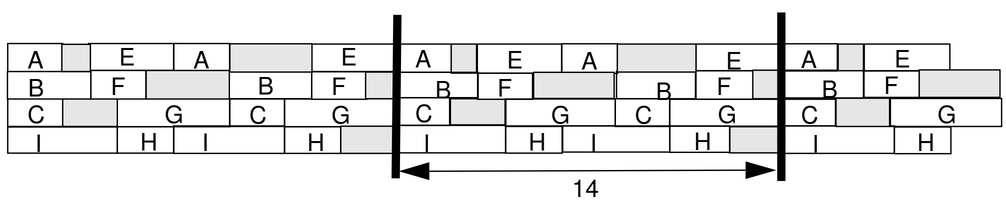
We will find it useful to partition the edges of the IPC graph in the following manner: $E_{ipc} \equiv E_{int} \cup E_{comm}$, where E_{comm} are the **communication edges** (shown dotted in Fig. 5.2(d)) that are directed from the send to the receive actors in G_{ipc}, and E_{int} are the "internal" edges that represent the fact that actors assigned to a particular processor (actors internal to that processor) are executed sequen-



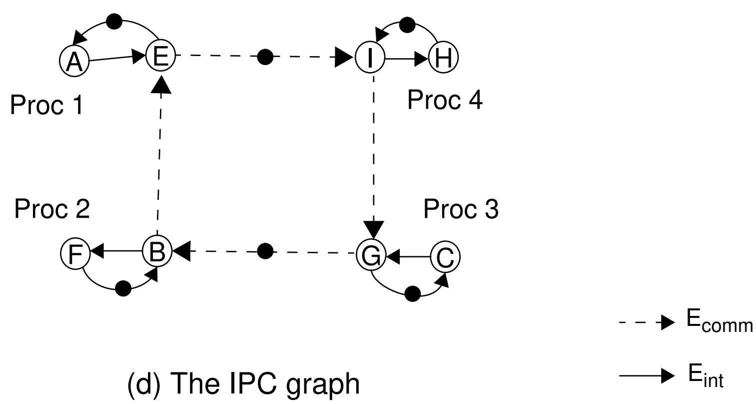
(a) HSDFG “G”



(b) Schedule on four processors



(c) Self-timed execution



(d) The IPC graph

Figure 5.2. Self-timed execution

tially according to the order predetermined by the self-timed schedule. A communication edge $e \in E_{comm}$ 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. As mentioned before, we assume the use of shared memory for the purpose of synchronization; the synchronization operation itself must be implemented using some kind of software *protocol* between the sender and the receiver. We discuss these synchronization protocols shortly.

5.2.1 Estimated throughput

Recall from Eqn. 4-3 that the average iteration period corresponding to a self-timed schedule with an IPC graph G_{ipc} is given by the maximum cycle mean of the graph $MCM(G_{ipc})$. If we only have execution time estimates available instead of exact values, and we set the execution times of actors $t(v)$ to be equal to these estimated values, then we obtain the *estimated* iteration period by computing $MCM(G_{ipc})$. Henceforth we will assume that we know the **estimated throughput** MCM^{-1} calculated by setting the $t(v)$ values to the available timing estimates.

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

5.3 Strongly connected components and buffer size bounds

In dataflow semantics, the edges between actors represent infinite buffers. Accordingly, the edges of the IPC graph are potentially buffers of infinite size. However, from Lemma 4.1, every **feedback edge** (an edge that belongs to a strongly connected component, and hence to some cycle) can only have a finite number of tokens at any time during the execution of the IPC graph. We will call this constant the **self-timed buffer bound** of that edge, and for a feedback edge e we will represent this bound by $B_{fb}(e)$. Lemma 4.1 yields the following self-timed buffer bound:

$$B_{fb}(e) = min(\{Delay(C) \mid C \text{ is a cycle that contains } e\}) \tag{5-1}$$

Feedforward edges (edges that do not belong to any SCC) have no such bound on buffer size; therefore for practical implementations we need to *impose* a bound on the sizes of these edges. For example, Figure 5.3(a) shows an IPC graph where the communication edge (s, r) could be unbounded when the execution time of A is less than that of B, for example. In practice, we need to bound the

Figure 5.3. An IPC graph with a feedforward edge: **(a)** original graph **(b)** imposing bounded buffers.

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 throughput can be modelled by adding a reverse edge that has m delays on it, where $m = B_{ff}(e) - delay(e)$, to G_{ipc} (grey edge in Fig. 5.3(b)). Since the addition of this edge introduces a new cycle in G_{ipc}, it has the potential to reduce the estimated throughput; to prevent such a reduction, $B_{ff}(e)$ must be chosen to be large enough so that the maximum cycle mean remains unchanged upon adding the reverse edge with m delays.

Sizing buffers optimally such that the maximum cycle mean remains unchanged has been studied by Kung, Lewis and Lo in [Kung87], 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 HSDFG (potentially an exponential number of constraints).

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

$$B_{ff}(e) \geq \left\lceil \left(\sum_{x \in V} t(x) \right) / (MCM(G_{ipc})) \right\rceil$$

holds for each feedforward edge e, then the maximum cycle mean of the resulting graph does not exceed MCM.

Then, a binary search on $B_{ff}(e)$ for each feedforward edge, while computing the maximum cycle mean at each search step and ascertaining that it is less than $MCM(G_{ipc})$, 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.

As we will see in section 5.7, however, imposing such a bound B_{ff} is a *naive* approach for bounding buffer sizes, because such a bound entails an added synchronization cost. In section 5.7 we show that there is a better technique for bounding buffer sizes; this technique achieves bounded buffer sizes by transforming the graph into a strongly connected graph by adding a minimal number of additional synchronization edges. Thus, in our final algorithm, we will not in fact find it necessary to use or compute these bounds B_{ff}.

5.4 Synchronization model

5.4.1 Synchronization protocols

We define two basic synchronization protocols for a communication 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 a communication 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 an empty buffer; *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 the previous section.

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 defined so far.

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

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

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

5.4.2 The synchronization graph G_s

As we discussed in the beginning of this chapter, some of the communication edges in G_{ipc} need not have explicit synchronization, whereas others require synchronization, which need to be implemented either using the UBS protocol or the BBS protocol. All communication edges also represent buffers in shared memory. Thus we divide the set of communication edges as follows: $E_{comm} \equiv E_s \cup E_r$, where the edges E_s need explicit synchronization operations to be implemented, and the edges E_r need no explicit synchronization. We call the edges E_s **synchronization edges**.

Recall that a communication edge (v_j, v_i) of G_{ipc} represents the **synchronization constraint**:

$$start(v_i, k) \geq end(v_j, k - delay((v_j, v_i))) \ \forall k > delay(v_j, v_i) \tag{5-2}$$

Thus, before we perform any optimization on synchronizations, $E_{comm} \equiv E_s$ and $E_r \equiv \phi$, because every communication edge represents a synchronization point. However, in the following sections we describe how we can move certain edges from E_s to E_r, thus reducing synchronization operations in the final implementation. At the end of our optimizations, the communication edges of the IPC graph fall into either E_s or E_r. At this point the edges $E_s \cup E_r$ in G_{ipc} represent buffer activity, and must be implemented as buffers in shared memory, whereas the edges E_s represent synchronization constraints, and are implemented using the UBS and BBS protocols introduced in the previous section. For the edges in E_s the synchronization protocol is executed before the buffers corresponding to the communication edge are accessed so as to ensure sender-receiver synchronization. For edges in E_r, however, no synchronization needs to be done before accessing the shared buffer. Sometimes we will also find it useful to introduce synchronization edges without actually communicating data between the sender and the receiver (for the purpose of ensuring finite buffers for example), so that no shared buffers need to be assigned to these edges, but the corresponding synchronization protocol is invoked for these edges.

All optimizations that move edges from E_s to E_r must respect the synchronization constraints implied by G_{ipc}. If we ensure this, then we only need to implement the synchronization protocols for the edges in E_s. We call the graph $G_s = (V, E_{int} \cup E_s)$ the **synchronization graph**. The graph G_s represents the synchronization constraints in G_{ipc} that need to be explicitly ensured, and the algorithms we present for minimizing synchronization costs operate on G_s. Before any synchronization related optimizations are performed $G_s \equiv G_{ipc}$, because $E_{comm} \equiv E_s$ at this stage, but as we move communication edges from E_s to E_r, G_s has fewer and fewer edges. Thus moving edges from E_s to E_r can be viewed as removal of edges from G_s. Whenever we remove edges from G_s we have to ensure, of course, that the synchronization graph G_s at that step respects all the synchronization constraints of G_{ipc}, because we only implement synchronizations represented by the edges E_s in G_s. The following theorem is useful to formalize the concept of when the synchronization constraints represented by one synchronization graph G_s^1 imply the synchronization constraints of another graph G_s^2. This theorem provides a useful constraint for synchronization optimization, and it underlies the validity of the main techniques that we will present in this chapter.

Theorem 5.1: The synchronization constraints in a synchronization graph $G_s^{\,1} = \left(V, E_{int} \cup E_s^{\,1}\right)$ imply the synchronization constraints of the synchronization graph $G_s^{\,2} = \left(V, E_{int} \cup E_s^{\,2}\right)$ if the following condition holds: $\forall \varepsilon$ s.t. $\varepsilon \in E_s^{\,2}, \varepsilon \notin E_s^{\,1}$, $\rho_{G_s^{\,1}}(src(\varepsilon), snk(\varepsilon)) \le delay(\varepsilon)$; that is, if for each edge ε that is present in $G_s^{\,2}$ but not in $G_s^{\,1}$ there is a minimum delay path from $src(\varepsilon)$ to $snk(\varepsilon)$ in $G_s^{\,1}$ that has total delay of at most $delay(\varepsilon)$.

(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 $G_s^{\,1}$ even though there are edges ε s.t. $\varepsilon \in E_s^{\,2}, \varepsilon \notin E_s^{\,1}$.)

First we prove the following lemma.

Lemma 5.1: If there is a path $p = (e_1, e_2, e_3, \ldots, e_n)$ in $G_s^{\,1}$, then

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

Proof of Lemma 5.1*:*

The following constraints hold along such a path p (as per Eqn. 4-1)

$$start(snk(e_1), k) \ge end(src(e_1), k - delay(e_1)) \ . \tag{5-3}$$

Similarly,

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

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

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

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

$$start(snk(e_2), k) \geq start(snk(e_1), k - delay(e_2)) \,. \tag{5-4}$$

Substituting Eqn. 5-3 in Eqn. 5-4,

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

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

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

that is,

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

Proof of Theorem 5.1: If $\varepsilon \in E_s^{\;2}, \varepsilon \in E_s^{\;1}$, then the synchronization constraint due to the edge ε holds in both graphs. But for each ε s.t. $\varepsilon \in E_s^{\;2}, \varepsilon \notin E_s^{\;1}$ we need to show that the constraint due to ε:

$$start(snk(\varepsilon), k) > end(src(\varepsilon), k - delay(\varepsilon)) \tag{5-5}$$

holds in $G_s^{\;1}$ provided $\rho_{G_s^{\;1}}(src(\varepsilon), snk(\varepsilon)) \leq delay(\varepsilon)$, which implies there is at least one path $p = (e_1, e_2, e_3, \ldots, e_n)$ from $src(\varepsilon)$ to $snk(\varepsilon)$ in $G_s^{\;1}$ ($src(e_1) = src(\varepsilon)$ and $snk(e_n) = snk(\varepsilon)$) such that $Delay(p) \leq delay(\varepsilon)$.

From Lemma 5.1, existence of such a path p implies

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

that is,

$$start((snk(\varepsilon), k) \geq end(src(\varepsilon), k - Delay(p))) \quad . \tag{5-6}$$

If $Delay(p) \leq delay(\varepsilon)$, then

$end(src(\varepsilon), k - Delay(p)) \geq end(src(\varepsilon), k - delay(\varepsilon))$. Substituting this in Eqn. 5-6 we get

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

The above relation is identical to Eqn. 5-5, and this proves the Theorem. *QED*.

The above theorem motivates the following definition.

Definition 5.1: If $G_s^1 = \left(V, E_{int} \cup E_s^1\right)$ and $G_s^2 = \left(V, E_{int} \cup E_s^2\right)$ are synchronization graphs with the same vertex-set, we say that G_s^1 **preserves** G_s^2 if $\forall \varepsilon$ s.t. $\varepsilon \in E_2, \varepsilon \notin E_1$, we have $\rho_{G_s^1}(src(\varepsilon), snk(\varepsilon)) \leq delay(\varepsilon)$.

Thus, Theorem 5.1 states that the synchronization constraints of $\left(V, E_{int} \cup E_s^1\right)$ imply the synchronization constraints of $\left(V, E_{int} \cup E_s^2\right)$ if $\left(V, E_{int} \cup E_s^1\right)$ preserves $\left(V, E_{int} \cup E_s^2\right)$.

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

5.5 Formal problem statement

We refer to each access of the shared memory "synchronization variable" $sv(e)$ by $src(e)$ and $snk(e)$ as a **synchronization access**[1] to shared memory. If synchronization for e is implemented using UBS, then we see that on average, 4 synchronization accesses are required for e in each iteration period, while BBS implies 2 synchronization accesses per iteration period. We define the **synchronization cost** of a synchronization graph G_s to be the average number of synchronization accesses required per iteration period. Thus, if n_{ff} denotes the number of synchronization edges in G_s that are feedforward edges, and n_{fb} denotes the number of synchronization edges that are feedback edges, then the synchronization cost of G_s can be expressed as $(4n_{ff} + 2n_{fb})$. In the remainder of this paper we develop techniques that apply the results and the analysis framework developed in sections 4.1 and sections 5.2-5.4 to minimize the synchronization cost of a self-timed implementation of an HSDFG without sacrificing the integrity of any interprocessor data transfer or reducing the estimated throughput.

We will explore three mechanisms for reducing synchronization accesses. The first (presented in section 5.6) is the detection and removal of *redundant* synchronization edges, which are synchronization edges whose respective synchronization functions are subsumed by other synchronization edges, and thus need not be implemented explicitly. This technique essentially detects the set of edges that can be moved from the E_s to the set E_r. In section 5.7, we examine the utility of adding additional synchronization edges to convert a synchronization graph that is not strongly connected into a strongly connected graph. Such a conversion allows us to implement all synchronization edges with BBS. We address optimization criteria in performing such a conversion, and we will show that the extra synchronization accesses required for such a conversion are always (at least) compensated by the number of synchronization accesses that are saved by the more expensive UBSs that are converted to BBSs. Finally, in section 5.9 we outline a mechanism, which we call *resynchronization*, for inserting synchronization edges in a way that

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.

the number of original synchronization edges that become redundant exceeds the number of new edges added.

5.6 Removing redundant synchronizations

The first technique that we explore for reducing synchronization overhead is removal of *redundant synchronization edges* from the synchronization graph, i.e. finding a minimal set of edges E_s that need explicit synchronization. Formally, a synchronization edge is **redundant** in a synchronization graph G if its removal yields a synchronization graph that preserves G. Equivalently, from definition 5.1, 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. 5.4 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 complete before each invocation of D is begun. Thus, x_2 is redundant in Fig. 5.4 and can be removed from E_s into the set E_r. It is easily verified that the path

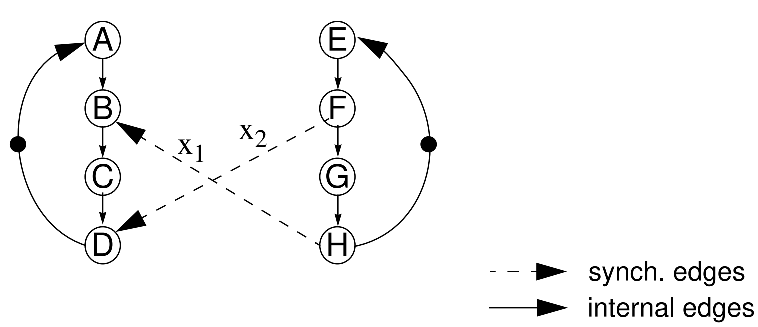


Figure 5.4. x_2 is an example of a redundant synchronization edge.

$p = ((F, G), (G, H), x_1, (B, C), (C, D))$ is directed from $src(x_2)$ to $snk(x_2)$, and has a path delay (zero) that is equal to the delay on x_2.

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

5.6.1 The independence of redundant synchronizations

The following theorem establishes that the order in which we remove redundant synchronization edges is not important; therefore all the redundant synchronization edges can be removed together.

Theorem 5.2: Suppose that $G_s = (V, E_{int} \cup E_s)$ is a synchronization graph, e_1 and e_2 are distinct redundant synchronization edges in G_s (i.e. these are edges that could be individually moved to E_r), and $\tilde{G}_s = \left(V, E_{int} \cup \left(E - \{e_1\}\right)\right)$. Then e_2 is redundant in $\tilde{G}_s$. Thus both e_1 and e_2 can be moved into E_r *together.*

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

$$Delay(p) \le delay(e_2). \tag{5-7}$$

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

$$Delay(p') \le delay(e_1). \tag{5-8}$$

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,

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

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

$$\le Delay(p) \qquad \text{(from Eqn. 5-8)}$$

$$\le delay(e_2) \qquad \text{(from Eqn. 5-7).}$$

QED.

Theorem 5.2 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, and since computing the weight of the shortest path in a weighted directed graph is a tractable problem, we can expect such a solution to be practical.

5.6.2 Removing redundant synchronizations

Fig. 5.5 presents an efficient algorithm, based on the ideas presented in the previous subsection, for optimal removal of redundant synchronization edges. In this algorithm, we first compute the path delay of a minimum-delay path from x to y for each ordered pair of vertices (x, y) ; here, we assign a path delay of ∞ whenever there is no path from x to y. This computation is equivalent to solving an instance of the well known *all points shortest paths problem* [Corm92]. Then, we examine each synchronization edge e — in some arbitrary sequence — and determine whether or not there is a path from $src(e)$ to $snk(e)$ that does not contain e, and that has a path delay that does not exceed $delay(e)$. This check for redundancy is equivalent to the check that is performed by the *if* statement in

RemoveRedundantSynchs because if p is a path from $src(e)$ to $snk(e)$ that contains more than one edge and that contains e, then p must contain a cycle c such that c does not contain e; and since all cycles must have positive path delay (from Lemma 4.1), the path delay of such a path p must exceed $delay(e)$. Thus, if e_0 satisfies the inequality in the *if* statement of *RemoveRedundantSynchs,* and $p*$ is a path from $snk(e_0)$ to $snk(e)$ such that $Delay(p*) = \rho(snk(e_0), snk(e))$, then $p*$ cannot contain e. This observation allows us to avoid having to recompute the shortest paths after removing a candidate redundant edge from G_s.

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

Function RemoveRedundantSynchs
Input: A synchronization graph $G_s = E_{int} \cup E_s$
Output: The synchronization graph $G_s* = (V, E_{int} \cup (E_s - E_r))$

1. Compute $\rho_{G_s}(x, y)$ for each ordered pair of vertices in G_s.
2. $E_r \leftarrow \varnothing$
3. **For** each $e \in E_s$
    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** $(V, E_{int} \cup (E_s - E_r))$.

Figure 5.5. An algorithm that optimally removes redundant synchronization edges.

minimum-delay path values computed in Step 1 need to be recalculated after removing a redundant synchronization edge in Step 3.

Observe that the complexity of the function *RemoveRedundantSynchs* is dominated by Step 1 and Step 3. Since all edge delays are non-negative, we can repeatedly apply Dijkstra's single-source shortest path algorithm (once for each vertex) to carry out Step 1 in $O\left(|V|^3\right)$ time; a modification of Dijkstra's algorithm can be used to reduce the complexity of Step 1 to $O\left(|V|^2\log_2(|V|) + |V||E|\right)$ [Corm92]. In Step 3, $|E|$ is an upper bound for the number of synchronization edges, and in the worst case, each vertex has an edge connecting it to every other member of V. Thus, the time complexity of Step 3 is $O(|V||E|)$, and if we use the modification to Dijkstra's algorithm mentioned above for Step 1, then the time complexity of *RemoveRedundantSynchs* is

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

5.6.3 Comparison with Shaffer's approach

In [Sha89], Shaffer presents an algorithm that minimizes the number of directed synchronizations in the self-timed execution of an HSDFG under the (implicit) assumption that the execution of successive iterations of the HSDFG are not allowed to overlap. In Shaffer's technique, a construction identical to our synchronization graph is used except 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 an HSDFG; 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.

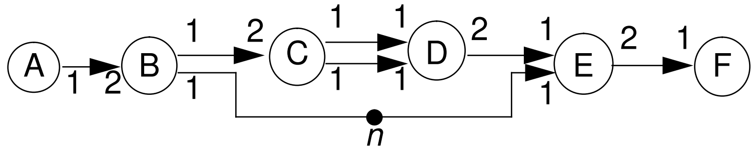
5.6.4 An example

In this subsection, we illustrate the benefits of removing redundant synchronizations through a practical example. Fig. 5.6(a) shows an abstraction of a three channel, multi-resolution quadrature mirror (QMF) filter bank, which has applications in signal compression [Vai93]. This representation is based on the general (not homogeneous) SDF model, and accordingly, each edge is annotated with the number of tokens produced and consumed by its source and sink actors. Actors A and F represent the subsystems that, respectively, supply and consume data to/from the filter bank system; B and C each represents a parallel combination of decimating high and low pass FIR analysis filters; D and E represent the corresponding pairs of interpolating synthesis filters. The amount of delay on the edge directed from B to E is equal to the sum of the filter orders of C and D. For more details on the application represented by Fig. 5.6(a), we refer the reader to [Vai93].

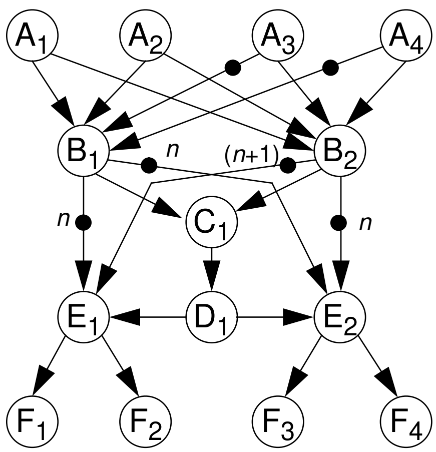
To construct a periodic, parallel schedule we must first determine the number of times $\mathbf{q}(N)$ that each actor N must be invoked in the periodic schedule. Systematic techniques to compute these values are presented in [Lee87]. Next, we must determine the precedence relationships between the actor invocations. In determining the exact precedence relationships, we must take into account the dependence of a given filter invocation on not only the invocation that produces the token that is "consumed" by the filter, but also on the invocations that produce the n preceding tokens, where n is the order of the filter. Such dependence can easily be evaluated with an additional dataflow parameter on each actor input that specifies the number of *past tokens* that are accessed [Prin91][1]. Using this infor-

1. It should be noted that some SDF-based design environments choose to forego parallelization across multiple invocations of an actor in favor of simplified code generation and scheduling. For example, in the GRAPE system, this restriction has been justified on the grounds that it simplifies inter-processor data management, reduces code duplication, and allows the derivation of efficient scheduling algorithms that operate directly on general SDF graphs without requiring the use of the acyclic precedence graph (APG) [Bil94].

(a)



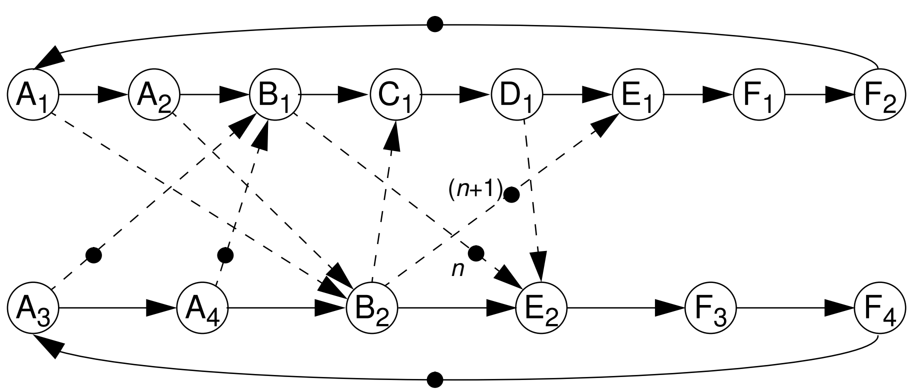
(b)



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

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

(c)



(d)

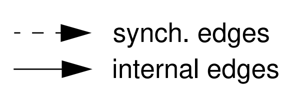


Figure 5.6. (a) A multi-resolution QMF filter bank used to illustrate the benefits of removing redundant synchronizations. (b) The precedence graph for (a). (c) A self-timed, two-processor, parallel schedule for (a). (d) The initial synchronization graph for (c).

mation, together with the invocation counts specified by $\mathbf{q}$, we obtain the precedence relationships specified by the graph of Fig. 5.6(b), in which the ith invocation of actor N is labeled N_i, and each edge e specifies that invocation $snk(e)$ requires data produced by invocation $src(e)$ $delay(e)$ iteration periods after the iteration period in which the data is produced.

A self-timed schedule for Fig. 5.6(b) that can be obtained from Hu's list scheduling method [Hu61] (described in is specified in Chapter 1 section 1.2) is specified in Fig. 5.6(c), and the synchronization graph that corresponds to the IPC graph of Fig. 5.6(b) and Fig. 5.6(c) is shown in Fig. 5.6(d). All of the dashed edges in Fig. 5.6(d) are synchronization edges. If we apply Shaffer's method, which considers only those synchronization edges that do not have delay, we can eliminate the need for explicit synchronization along only one of the 8 synchronization edges — edge (A_1, B_2). In contrast, if we apply *RemoveRedundantSynchs*, we can detect the redundancy of (A_1, B_2) as well as four additional redundant synchronization edges — (A_3, B_1), (A_4, B_1), (B_2, E_1), and (B_1, E_2). Thus, *RemoveRedundantSynchs* reduces the number of synchronizations from 8 down to 3 — a reduction of 62%. Fig. 5.7 shows the synchronization graph of Fig. 5.6(d) after all redundant synchronization edges are removed. It is easily verified that the synchronization edges that remain in this graph are not redundant; explicit synchronizations need only be implemented for these edges.

5.7 Making the synchronization graph strongly connected

In section 5.4.1, we defined two different synchronization protocols — bounded buffer synchronization (BBS), which has a cost of 2 synchronization accesses per iteration period, and can be used whenever the associated edge is contained in a strongly connected component of the synchronization graph; and unbounded buffer synchronization (UBS), which has a cost of 4 synchronization accesses per iteration period. We pay the additional overhead of UBS whenever the associated edge is a feedforward edge of the synchronization graph.

One alternative to implementing UBS for a feedforward edge e is to add synchronization edges to the synchronization graph so that e becomes encapsulated in a strongly connected component; such a transformation would allow e to be implemented with BBS. However, extra synchronization accesses will be required to implement the new synchronization edges that are inserted. In this section, we show that by adding synchronization edges through a certain simple procedure, the synchronization graph can be transformed into a strongly connected graph in a way that the overhead of implementing the extra synchronization edges is always compensated by the savings attained by being able to avoid the use of UBS. That is, our transformations ensure that the total number of synchronization accesses required (per iteration period) for the transformed graph is less than or equal to the number of synchronization accesses required for the original synchronization graph. Through a practical example, we show that this transformation can significantly reduce the number of required synchronization accesses. Also, we discuss a technique to compute the delay that should be added to each of the new edges added in the conversion to a strongly connected graph. This technique com-

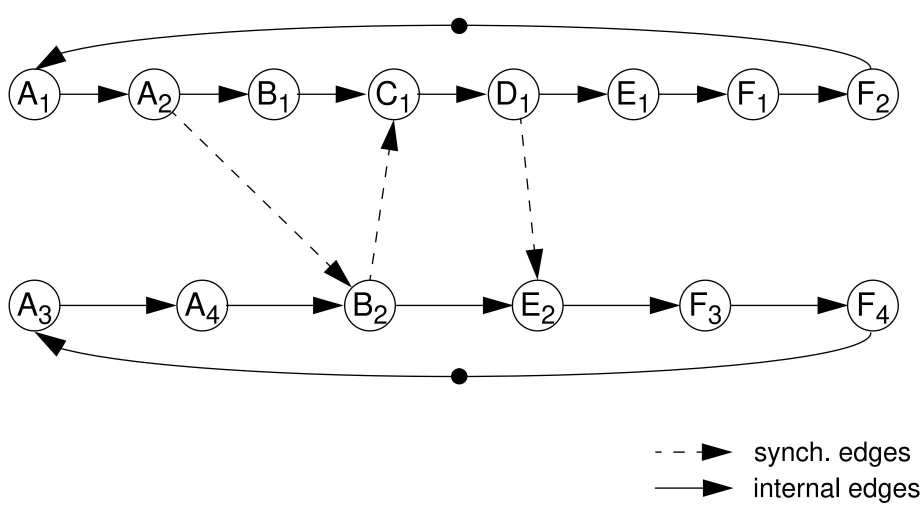


Figure 5.7. The synchronization graph of Fig. 5.6(d) after all redundant synchronization edges are removed.

putes the delays in a way that the estimated throughput of the IPC graph is preserved with minimal increase in the shared memory storage cost required to implement the communication edges.

5.7.1 Adding edges to the synchronization graph

Fig. 5.8 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 mini-

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

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

Figure 5.8. An algorithm for converting a synchronization graph that is not strongly connected into a strongly connected graph.

mum execution time to be the chain "link" corresponding to that SCC. Minimum execution time vertices are chosen in an attempt to minimize the amount of delay that must be inserted on the new edges to preserve the estimated throughput of the original graph. In section 5.7.2, we discuss the selection of delays for the edges introduced by *Convert-to-SC-graph*.

It is easily verified that algorithm *Convert-to-SC-graph* always produces a strongly connected graph, and that a conversion to a strongly connected graph cannot be attained by adding fewer edges than the number of edges added by *Convert-to-SC-graph*. Fig. 5.9 illustrates a possible solution obtained by algorithm *Convert-to-SC-graph*. Here, the black dashed edges are the synchronization edges contained in the original synchronization graph, and the grey dashed edges are the edges that are added by *Convert-to-SC-graph*. The dashed edge labeled e_s is the sink-source edge.

Assuming the synchronization graph is connected, the number of feedforward edges n_f must satisfy $(n_f \geq (n_c - 1))$, where n_c is the number of SCCs. This follows from the fundamental graph theoretic fact that in a connected graph (V^*, E^*), $|E^*|$ must be at least $(|V^*| - 1)$. Now, it is easily verified that the

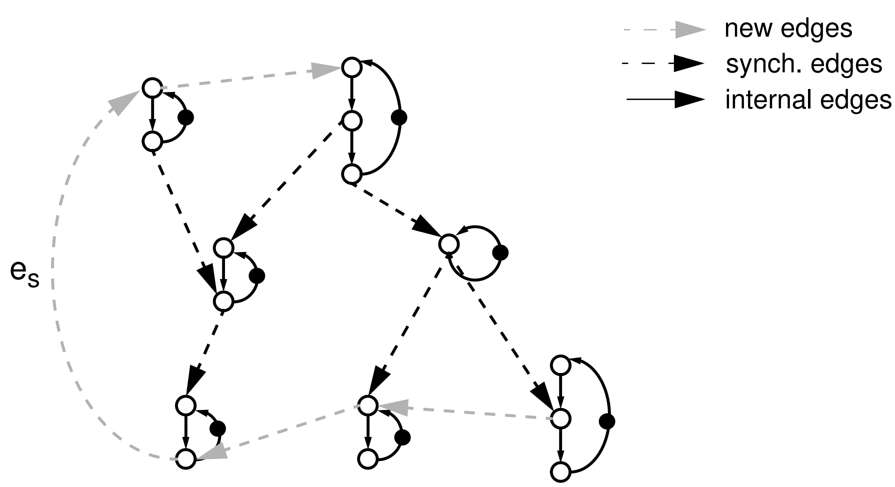


Figure 5.9. An illustration of a possible solution obtained by algorithm *Convert-to-SC-graph.*

number of new edges introduced by *Convert-to-SC-graph* is equal to $(n_{src} + n_{snk} - 1)$, where n_{src} is the number of source SCCs, and n_{snk} is the number of sink SCCs. Thus, the number of synchronization accesses per iteration period, S_+, that is required to implement the edges introduced by *Convert-to-SC-graph* is $(2 \times (n_{src} + n_{snk} - 1))$, while the number of synchronization accesses, S_-, eliminated by *Convert-to-SC-graph* (by allowing the feedforward edges of the original synchronization graph to be implemented with BBS rather than UBS) equals $2n_f$. It follows that the net change $(S_+ - S_-)$ in the number of synchronization accesses satisfies

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

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

Theorem 5.3: Suppose that G is a synchronization graph, and $\hat{G}$ is the graph that results from applying algorithm *Convert-to-SC-graph* to G. Then the synchronization cost of $\hat{G}$ is less than or equal to the synchronization cost of G.

For example, without the edges added by *Convert-to-SC-graph* (the dashed grey edges) in Fig. 5.9, there are 6 feedforward edges, which require 24 synchronization accesses per iteration period to implement. The addition of the 4 dashed edges requires 8 synchronization accesses to implement these new edges, but allows us to use UBS for the original feedforward edges, which leads to a savings of 12 synchronization accesses for the original feedforward edges. Thus, the net effect achieved by *Convert-to-SC-graph* in this example is a reduction of the total number of synchronization accesses by $(12 - 8) = 4$. As another example, consider Fig. 5.10, which shows the synchronization graph topology (after redundant synchronization edges are removed) 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 algorithm was also discussed in Chapter 3, as an example application that was implemented on the ordered memory access architecture prototype. This graph contains $n_i = 6$ synchronization edges (the dashed edges), all of which are feedforward edges, so the synchronization cost is $4n_i = 24$ synchronization access per iteration period. Since the graph has one source SCC and one sink SCC, only one edge is added by *Convert-to-SC-graph*, and adding this edge reduces the synchronization cost to $2n_i + 2 = 14$ — a 42% savings. Fig. 5.11 shows the topology of a possible solution computed by *Convert-to-SC-graph* on this example. Here, the dashed edges represent the synchronization edges in the synchronization graph returned by *Convert-to-SC-graph*.

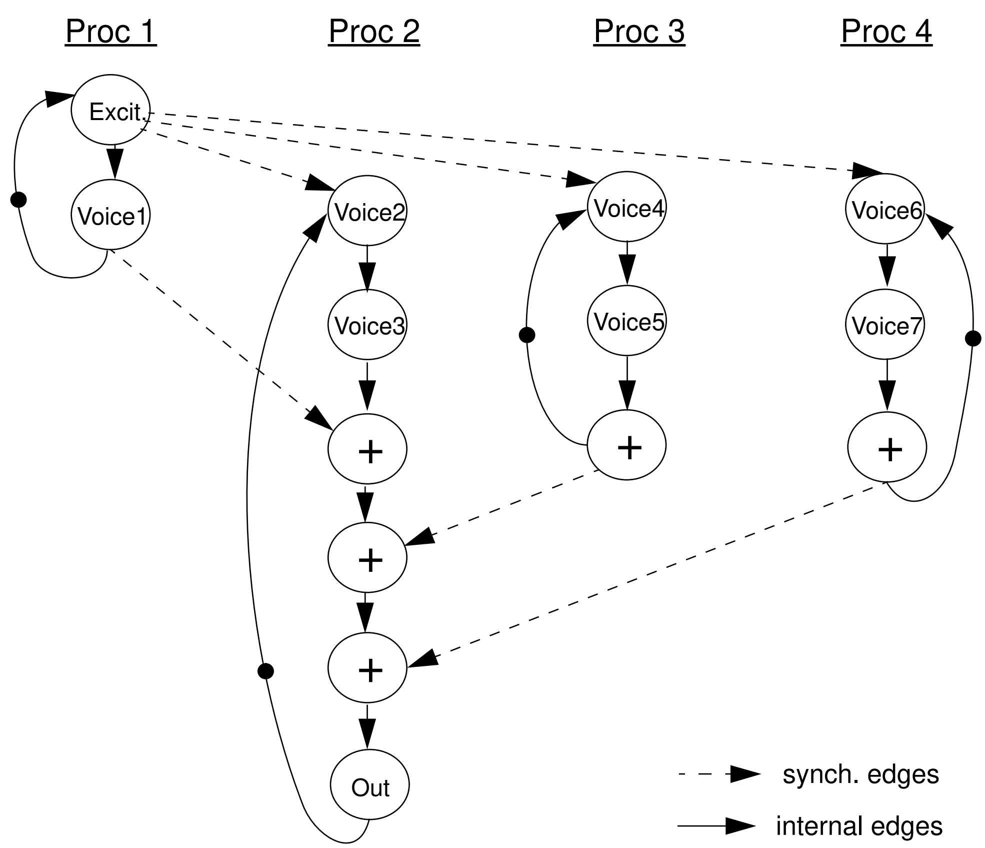


Figure 5.10. The synchronization graph, after redundant synchronization edges are removed, induced by a four-processor schedule of a music synthesizer based on the Karplus-Strong algorithm.

5.7.2 Insertion of delays

One issue remains to be addressed in the conversion of a synchronization graph G_s into a strongly connected graph $\hat{G}_s$ — the proper insertion of delays so that $\hat{G}_s$ is not deadlocked, and does not have lower estimated throughput than G_s. The potential for deadlock and reduced estimated throughput arise because the conversion to a strongly connected graph must necessarily introduce one or more new fundamental cycles. In general, a new cycle may be delay-free, or its cycle mean may exceed that of the critical cycle in G_s. Thus, we may have to insert delays on the edges added by *Convert-to-SC-graph*. The location (edge) and magnitude of the delays that we add are significant since they effect the self-timed buffer bounds of the communication edges, as shown subsequently in Theorem 5.4. 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 a way that the sum of the self-timed buffer bounds over all communication edges is minimized. In this section, we outline a simple and efficient algorithm for addressing this problem. Our algorithm pro-

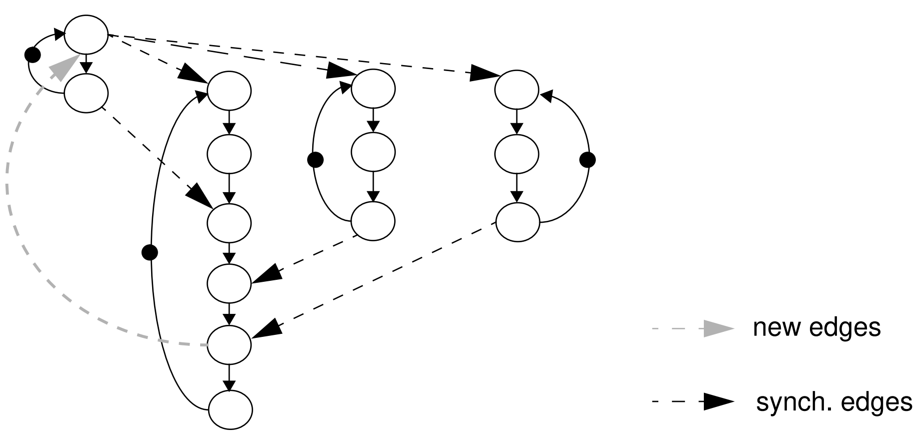


Figure 5.11. A possible solution obtained by applying *Convert-to-SC-graph* to the example of Figure 5.10.

duces 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.

Fig. 5.12 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 repeatedly applies the Bellman-Ford algorithm discussed in pp. 94-97 of [Law76] to return the cycle mean of the critical cycle in Z; if one or more cycles exist that have zero path delay, then *BellmanFord* returns ∞. Details of this procedure can be found in [Bhat95a].

Fig. 5.13 illustrates a solution obtained from *DetermineDelays*. Here we assume that $t(v) = 1$ for each vertex v, and we assume that the set of communication edges are e_a and e_b. The grey dashed edges are the edges added by *Convert-to-SC-graph*. We see that MCM is determined by the cycle in the sink SCC of the original graph, and inspection of this cycle yields $MCM = 4$. The solution determined by *DetermineDelays* for Fig. 5.13 is one delay on e_a and one delay on e_b ($\delta_0, \delta_1 = 1$); the resulting self-timed buffer bounds of e_a and e_b are, respectively, 1 and 2; the total buffer sizes for the communication edges is thus 3 (sum of the self-timed buffer bounds).

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

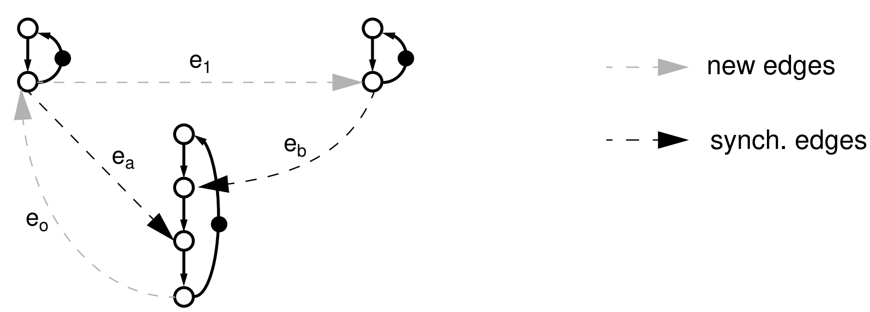


Figure 5.13. An example used to illustrate a solution obtained by algorithm *DetermineDelays*.

Function *DetermineDelays*

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

Output: Non-negative integers $d_o, d_1, \ldots, d_{m-1}$ such that the estimated throughput when $delay(e_i) = d_i$, $0 \le i \le m-1$, equals estimated throughput of G_s.

$X_0 = \hat{G}_s[e_0 \to \infty, \ldots, e_{m-1} \to \infty]$ /* set delays on each edge to be infinite */

λ_{max}= *BellmanFord*(X_0) /* compute the max. cycle mean of G_s */

$d_{ub} = \left\lceil \left(\sum_{x \in V} t(x) \right) / MCM \right\rceil$ /* an upper bound on the delay required for any e_i */

For $i = 0, 1, \ldots, m-1$

$\delta_i = MinDelay(X_i, e_i, MCM, d_{ub})$

$X_{i+1} = X_i[e_i \to \delta_i]$ /* fix the delay on e_i to be δ_i */

End For

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

Function *MinDelay*(X, e, λ, B)

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

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

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

Figure 5.12. An algorithm for determining the delays on the edges introduced by algorithm *Convert-to-SC-graph*.

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*. Further details related to these issues can be found in [Bhat95a].

DetermineDelays and its variations have complexity $O\left(|V|^4 (\log_2(|V|))^2 \right)$ [Bhat95a]. It is also easily verified that the time complexity of *DetermineDelays* dominates that of *Convert-to-SC-graph,* so the time complexity of applying *Convert-to-SC-graph* and *DetermineDelays* in succession is again $O\left(|V|^4 (\log_2(|V|))^2 \right)$.

Although the issue of deadlock does not explicitly arise in algorithm *DetermineDelays,* the algorithm does guarantee that the output graph is not deadlocked, assuming that the input graph is not deadlocked. This is because (from Lemma 4.1) deadlock is equivalent to the existence of a cycle that has zero path delay, and is thus equivalent to an infinite maximum cycle mean. Since *DetermineDelays* does not increase the maximum cycle mean, it follows that the algorithm cannot convert a graph that is not deadlocked into a deadlocked graph.

Converting a mixed grain HSDFG that contains feedforward edges into a strongly connected graph has been studied by Zivojnovic [Zivo94b] in the context of retiming when the assignment of actors to processors is fixed beforehand. In this case, the objective is to retime the input graph so that the number of communication edges that have nonzero delay is maximized, and the conversion is performed to constrain the set of possible retimings in such a way that an integer linear programming formulation can be developed. The technique generates two dummy vertices that are connected by an edge; the sink vertices of the original graph are connected to one of the dummy vertices, while the other dummy vertex is connected to each source. It is easily verified that in a self-timed execution, this scheme requires at least four more synchronization accesses per graph iteration than the method that we have proposed. We can obtain further relative savings if we succeed in detecting one or more beneficial resynchronization opportunities. The effect of Zivojnovic's retiming algorithm on synchronization overhead is unpredictable since one hand a communication edge becomes "easier to make redundant" when its delay increases, while on the other hand, the edge becomes less useful in making other communication edges redundant since the path delay of all paths that contain the edge increase.

5.8 Computing buffer bounds from G_s and G_{ipc}

After all the optimizations are complete we have a final synchronization graph $G_s = (V, E_{int} \cup E_s)$ that preserves G_{ipc}. Since the synchronization edges in G_s are the ones that are finally implemented, it is advantageous to calculate the self-timed buffer bound B_{fb} as a final step after all the transformations on G_s are complete, instead of using G_{ipc} itself to calculate these bounds. This is because addition of the edges in the *Convert-to-SC-graph* and *Resynchronize* steps may reduce these buffer bounds. It is easily verified that removal of edges cannot change the buffer bounds in Eqn. 5-1 as long as the synchronizations in G_{ipc} are preserved. Thus, in the interest of obtaining minimum possible shared buffer sizes, we compute the bounds using the optimized synchronization graph. The following theorem tells us how to compute the self-timed buffer bounds from G_s.

Theorem 5.4: If G_s preserves G_{ipc} and the synchronization edges in G_s are implemented, then for each feedback communication 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 be present on e — is given by:

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

Proof: By Lemma 5.1, if there is a path p from $snk(e)$ to $src(e)$ in G_s, then

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

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

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

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

The quantities $\rho_{G_s}(snk(e), src(e))$ can be computed using Dijkstra's algorithm [Corm92] to solve the all-pairs shortest path problem on the synchronization graph in time $O\left(|V|^3\right)$.

5.9 Resynchronization

It is sometimes possible to reduce the total number of synchronization edges E_s by adding new synchronization edges to a synchronization graph. We refer to the process of adding one or more new synchronization edges and removing the redundant edges that result as *resynchronization*; Fig. 5.14(a) illustrates this concept, where the dashed edges represent synchronization edges. Observe that if we insert the new synchronization edge $d_0(C, H)$, then two of the original synchronization edges — (B, G) and (E, J) — become redundant, and the net effect is that we require one less synchronization edge to be implemented. In Fig. 5.14(b), we show the synchronization graph that results from inserting the *resynchronization edge* $d_0(C, H)$ (grey edge) into Fig. 5.14(a), and then removing the redundant synchronization edges that result.

We refer to the problem of finding a resynchronization with the fewest number of final synchronization edges as the **resynchronization problem**. In

[Bhat95a] we formally establish that the resynchronization problem is NP-hard by deriving a polynomial time reduction from the classic *minimal set covering problem*, which is known to be NP-hard [Garey79], to the pair-wise resynchronization problem. The complexity remains the same whether we consider a general resynchronization problem that also attempts to insert edges within SCCs, or a restricted version that only adds feed-forward edges between SCCs (the *Resynchronize* procedure in [Bhat95a] restricts itself to the latter, because in this case it is simpler to ensure that the estimated throughput is unaffected by the added edges).

Although the correspondence that we establish between the resynchronization problem and set covering shows that the resynchronization problem probably cannot be attacked optimally with a polynomial-time algorithm, the correspondence allows any heuristic for set covering to be adapted easily into a heuristic for the pair-wise resynchronization problem, and applying such a heuristic to each pair of SCCs in a general synchronization graph yields a heuristic for the general (not just pair-wise) resynchronization problem [Bhat95a]. This is fortunate since the set covering problem has been studied in great depth, and efficient heuristic methods

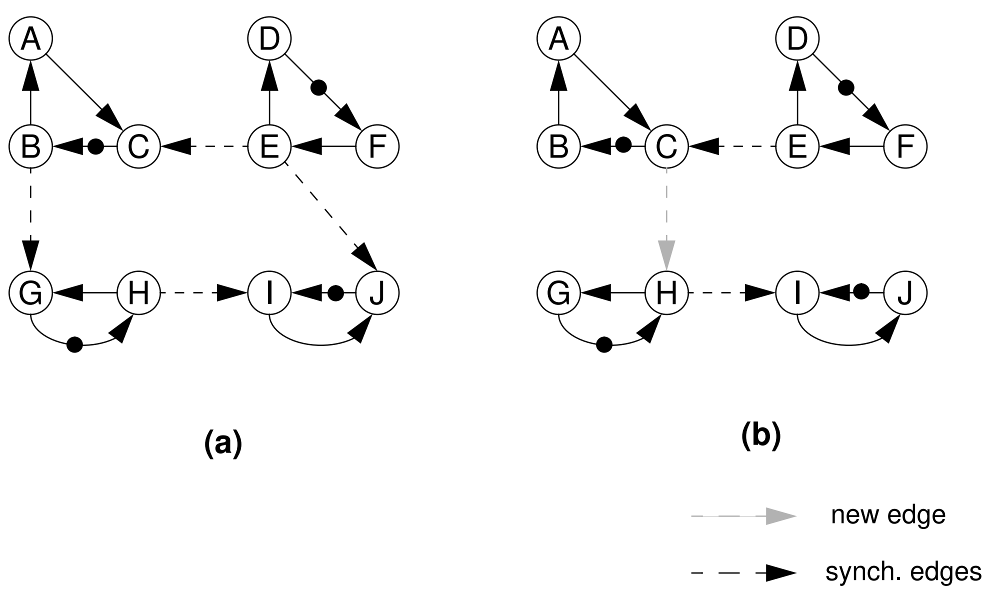


Figure 5.14. An example of resynchronization.

have been devised for it [Corm92].

For a certain class of IPC graphs (formally defined in [Bhat95b]) a provably optimum resynchronization can be obtained, using a procedure similar to pipelining. This procedure, however, leads to an implementation that in general has a larger latency than the implementation we start out with. The resynchronization procedure as outlined in [Bhat95a] in general can lead to implementations with increased latency. Latency is measured as the time delay between when an input data sample is available and when the corresponding output is generated. In [Bhat95b] we show how we can modify the resynchronization procedure to trade off synchronization cost with latency. An optimal latency constrained synchronization, however, is again shown to be NP-hard.

The work on resynchronization is very much ongoing research, a brief outline of which we have presented in this section.

5.10 Summary

We have addressed the problem of minimizing synchronization overhead in self-timed multiprocessor implementations. The metric we use to measure synchronization cost is the number of accesses made to shared memory for the purpose of synchronization, per schedule period. We used the IPC graph framework introduced in the previous chapter to extend an existing technique — detection of redundant synchronization edges — for noniterative programs to the iterative case. We presented a method for the conversion of the synchronization graph into a strongly connected graph, which again results in reduced synchronization overhead. Also, we briefly outlined the resynchronization procedure, which involves adding synchronization points in the schedule such that the overall synchronization costs are reduced. Details of resynchronization can be found in [Bhat95a] and [Bhat95b]. We demonstrated the relevance of our techniques through practical examples.

The input to our algorithm is an HSDFG and a parallel schedule for it. The output is an IPC graph $G_{ipc} = (V, E_{ipc})$, which represents buffers as communication edges; a strongly connected synchronization graph $G_s = (V, E_{int} \cup E_s)$, which represents synchronization constraints; and a set of shared-memory buffer sizes $\{B_{fb}(e) \mid e \text{ is an IPC edge in } G_{ipc}\}$. Fig. 5.15 specifies the complete algorithm.

A code generator can then accept G_{ipc} and G_s, allocate a buffer in shared memory for each communication edge e specified by G_{ipc} of size $B_{fb}(e)$, and generate synchronization code for the synchronization edges represented in G_s. These synchronizations may be implemented using the BBS protocol. The resulting synchronization cost is $2n_s$, where n_s is the number of synchronization edges in the synchronization graph G_s that is obtained after all optimizations are com-

Function *MinimizeSynchCost*
Input: An HSDFG G and a self-timed schedule S for this HSDFG.
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 communication edge is also a synchronization edge to begin with */

3. $G_s \leftarrow Resynchronize(G_s)$

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

5. $G_s \leftarrow DetermineDelays(G_s)$

6. $G_s \leftarrow RemoveRedundantSynchs(G_s)$

7. Calculate the buffer size $B_{fb}(e)$ for each communication 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 5.15. The complete synchronization optimization algorithm.

pleted.