Memorandum UCB/ERL M95/70, Electronics Research Laboratory,
University of California, Berkeley, September, 1995.

RESYNCHRONIZATION FOR EMBEDDED MULTIPROCESSORS

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

ABSTRACT

This paper introduces a technique, called *resynchronization*, for reducing synchronization overhead in embedded multiprocessor implementations. The technique exploits the well-known observation [39] that in a given multiprocessor implementation, certain synchronization operations may be *redundant* in the sense that their associated sequencing requirements are ensured by other synchronizations in the system. The goal of resynchronization is to introduce new synchronizations in such a way that the number of additional synchronizations that become redundant exceeds the number of new synchronizations that are added, and thus the net synchronization cost is reduced.

First, we define the general form of our resynchronization problem; we show that it is NP hard by establishing a correspondence to the *set covering* problem; and based on this correspondence, we specify how an arbitrary heuristic for set covering can be applied to yield a heuristic for resynchronization. Next, we show that for a certain class of applications, optimal resynchronizations can be computed efficiently by means of *pipelining*. These pipelined solutions, however, can suffer from significantly increased latency, and this motivates the *latency-constrained* resynchronization problem, which we address for a restricted class of graphs that permit efficient computation of latency. Again using a reduction from set covering (although the construction is significantly different), we show that latency-constrained resynchronization is NP hard. However, we show that for the special case in which there are only two processors, latency-constrained resynchronization can be solved in polynomial time. We also present a heuristic for latency-constrained resynchronization, and through a practical example, we demonstrate that this heuristic gives an efficient means for systematically trading off between synchronization overhead and latency.

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), the Semiconductor Research Corporation (project 94-DC-008), the National Science Foundation (MIP-9201605), the State of California MICRO program, and the following companies: Bellcore, Bell Northern Research, Dolby Laboratories, Hitachi, LG Electronics, Mentor Graphics, Mitsubishi, Motorola, NEC, Pacific Bell, Philips, and Rockwell.

S. S. Bhattacharyya is with the Semiconductor Research Laboratory, Hitachi America, Ltd., 201 East Tasman Drive, San Jose, California 95134, USA.

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

1. Introduction

This paper develops a technique called *resynchronization* for reducing the rate at which synchronization operations must be performed in a shared memory, embedded multiprocessor system. Resynchronization is based on the concept that there can be redundancy in the synchronization functions of a given multiprocessor implementation [39]. Such redundancy arises whenever the objective of one synchronization operation is guaranteed as a side effect of other synchronizations in the system. In the context of noniterative execution, Shaffer showed that the amount of run-time overhead required for synchronization can be reduced significantly by detecting redundant synchronizations and implementing only those synchronizations that are found not to be redundant; an efficient, optimal algorithm was also proposed for this purpose [39], and this algorithm was subsequently extended to handle iterative computations in [4].

The objective of resynchronization is to introduce new synchronizations in such a way that the number of original synchronizations that consequently become redundant is significantly less that the number of new synchronizations. We formulate and study this problem in the context of self-timed execution of iterative *synchronous dataflow* programs. Over the past several years, synchronous dataflow programming of iterative computations has attained significant popularity in the application domain of digital signal processing (see for example [13, 23, 30, 35, 34, 38]), and a wide variety of techniques have been developed to schedule synchronous dataflow programs for efficient multiprocessor implementation, such as those described in [1, 7, 14, 27, 32, 36, 40, 44].

Resynchronization has been studied earlier in the context of hardware synthesis [12]. However in this work, the scheduling model and implementation model are significantly different from the structure of self-timed multiprocessor implementations, and as a consequence, the problem formulations, analysis techniques, and algorithmic solutions do not apply to our context, and vice-versa. We will explain the major differences between these two contexts in Subsection 3.5, once we have developed the analytical models on which our work is based.

The purpose of this paper is to introduce the utility of resynchronization for self-timed multiprocessor implementations of synchronous dataflow programs, examine the complexity of fundamental problems that emerge from this concept, and develop a number of optimal and heu-

ristic solutions to some of these problems. In synchronous dataflow (**SDF**), a program is represented as a directed graph in which the vertices (**actors**) represent computations, the edges specify data dependences, and 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. This form of "synchrony" should not be confused with the use of "synchronous" in synchronous languages [3].

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 graphs [25], our techniques can easily be adapted to general SDF graphs. In the remainder of this paper, when we refer to a **dataflow graph** (**DFG**) we imply a homogeneous SDF graph.

Delays on DFG edges represent initial tokens, and specify dependencies between iterations of the actors in iterative execution. For example, if tokens produced by the kth execution of actor A are consumed by the $(k+2)$th execution of actor B, then the edge (A, B) contains two delays. In drawings of DFGs, we place a "D" on top of an edge that has unit delay, and if an edge has $n > 1$ delays, then we place "nD" on top of the edge (see Figure 1).

Multiprocessor implementation of an algorithm specified as a DFG requires scheduling the actors, which involves 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 [26] the authors propose a scheduling taxonomy based on which of these tasks are performed at compile time (static strategy) and which at run time (dynamic strategy); in this paper we will use the same terminology that was introduced there.

In the **fully-static** scheduling strategy of [26], 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 [26]. Each processor executes the actors assigned to it in a fixed order that is 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.

By a *processor*, we mean either a programmable component, in which case the actors mapped to it execute as software entities, or a hardware component, in which case actors assigned to it are implemented and execute in hardware. See [19] 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 [6, 21, 42].

Interprocessor communication (**IPC**) between processors is assumed to take place through shared memory, which could be global memory between all processors, or it could be distributed between pairs of processors (for example, hardware first-in-first-out (FIFO) queues or dual ported memory). Sender-receiver synchronization is also assumed to take place by setting flags in shared memory. Typically, special hardware for synchronization, such as barriers [10] or semaphores implemented in hardware, is prohibitively expensive for embedded multiprocessor machines. For the same reason, we cannot assume the efficient support for polling shared synchronization variables that is available on certain cache-coherent multiprocessors [28]. 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 [17]. Thus, in our context, effective resynchronization results in a significantly reduced rate of accesses to shared memory for the purpose of synchronization.

2. Background

Much of the nonstandard terminology that is introduced in this and subsequent sections is summarized in a glossary at the end of the paper.

We frequently represent a DFG by an ordered pair (V, E), where V is the set of vertices and E is the set of edges. We refer to the source and sink vertices of a graph edge e by $src(e)$ and $snk(e)$, we denote the delay on e by $delay(e)$, and we frequently represent e by the ordered pair $(src(e), snk(e))$. We say that e is an **output edge** of $src(e)$, and that e is an **input edge** of $snk(e)$. Edge e is **delayless** if $delay(e) = 0$, and it is a **self loop** if $src(e) = snk(e)$. In the iterative execution of a DFG, the kth invocation of an actor x is denoted x_k, for $k = 1, 2, \ldots$.

Given $x, y \in V$, we say that x is a **predecessor** of y if there exists $e \in E$ such that $src(e) = x$ and $snk(e) = y$; we say that x is a **successor** of y if y is a predecessor of x. A **path** in (V, E) is a finite, nonempty sequence $(e_1, e_2, \ldots, e_n)$, where each e_i is a member of E, and $snk(e_1) = src(e_2)$, $snk(e_2) = src(e_3)$, ..., $snk(e_{n-1}) = src(e_n)$. We say that the path $p = (e_1, e_2, \ldots, e_n)$ **contains** each e_i and each subsequence of $(e_1, e_2, \ldots, e_n)$; p is **directed from** $src(e_1)$ **to** $snk(e_n)$; and each member of $\{src(e_1), src(e_2), \ldots, src(e_n), snk(e_n)\}$ is **traversed by** p. A path that is directed from some vertex to itself is called a **cycle**, and a **fundamental cycle** is a cycle of which no proper subsequence is a cycle.

If $(p_1, p_2, \ldots, p_k)$ is a finite sequence of paths such that $p_i = (e_{i,1}, e_{i,2}, \ldots, e_{i,n_i})$, for $1 \le i \le k$, and $snk(e_{i,n_i}) = src(e_{i+1,1})$, for $1 \le i \le (k-1)$, then we define the **concatenation** of $(p_1, p_2, \ldots, p_k)$, denoted $\langle (p_1, p_2, \ldots, p_k) \rangle$, by

$$\langle (p_1, p_2, \ldots, p_k) \rangle \equiv (e_{1,1}, \ldots, e_{1,n_1}, e_{2,1}, \ldots, e_{2,n_2}, \ldots, e_{k,1}, \ldots, e_{k,n_k}) \ .$$

Clearly, $\langle (p_1, p_2, \ldots, p_k) \rangle$ is a path from $src(e_{1,1})$ to $snk(e_{k,n_k})$.

If $p = (e_1, e_2, \ldots, e_n)$ is a path in a DFG, then we define the **path delay** of p, denoted $Delay(p)$, by

$$Delay(p) = \sum_{i=1}^{n} delay(e_i) .$$

Since the delays on all DFG edges are restricted to be non-negative, it is easily seen that between

any two vertices $x, y \in V$, either there is no path directed from x to y, or there exists a (not necessarily unique) **minimum-delay path** between x and y. Given a DFG G, and vertices x, y in G, we define $\rho_G(x, y)$ to be equal to ∞ if there is no path from x to y, and equal to the path delay of a minimum-delay path from x to y if there exist one or more paths from x to y. If G is understood, then we may drop the subscript and simply write "ρ" in place of "ρ_G".

By a **subgraph** of (V, E), we mean the directed graph formed by any $V' \subseteq V$ together with the set of edges $\{e \in E | src(e), snk(e) \in V'\}$. We denote the subgraph associated with the vertex-subset V' by $subgraph(V')$. We say that (V, E) is **strongly connected** if for each pair of distinct vertices x, y, there is a path directed from x to y and there is a path directed from y to x. We say that a subset $V' \subseteq V$ is strongly connected if $subgraph(V')$ is strongly connected. A **strongly connected component (SCC)** of (V, E) is a strongly connected subset $V' \subseteq V$ such that no strongly connected subset of V properly contains V'. If V' is an SCC, then when there is no ambiguity, we may also say that $subgraph(V')$ is an SCC. If C_1 and C_2 are distinct SCCs in (V, E), we say that C_1 is a **predecessor SCC** of C_2 if there is an edge directed from some vertex in C_1 to some vertex in C_2; C_1 is a **successor SCC** of C_2 if C_2 is a predecessor SCC of C_1. An SCC is a **source SCC** if it has no predecessor SCC; an SCC is a **sink SCC** if it has no successor SCC; and an SCC is an **internal SCC** if it is neither a source SCC nor a sink SCC. An edge is a **feedforward** edge of (V, E) if it is not contained in an SCC, or equivalently, if it is not contained in a cycle; an edge that is contained in at least one cycle is called a **feedback** edge.

We denote the number of elements in a finite set S by $|S|$. Also, if r is a real number, then we denote the smallest integer that is greater than or equal to r by $\lceil r \rceil$.

3. Synchronization model

In this section, we present the model that we use for analyzing synchronization in self-timed multiprocessor systems. The basic model was presented originally in [41] to study the execution and interprocessor communication patterns of actors under self-timed evolution, and in [5], the model was augmented for the analysis of synchronization overhead.

Consider the execution of the four-processor schedule in Figure 1. In the self-timed execu-

tion shown in Figure 1(c), it is assumed that zero time is required for interprocessor communication. If the timing estimates are accurate, the schedule execution settles into a repeating pattern spanning two iterations of G, and the average iteration period is 7 time units.

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 to refer to

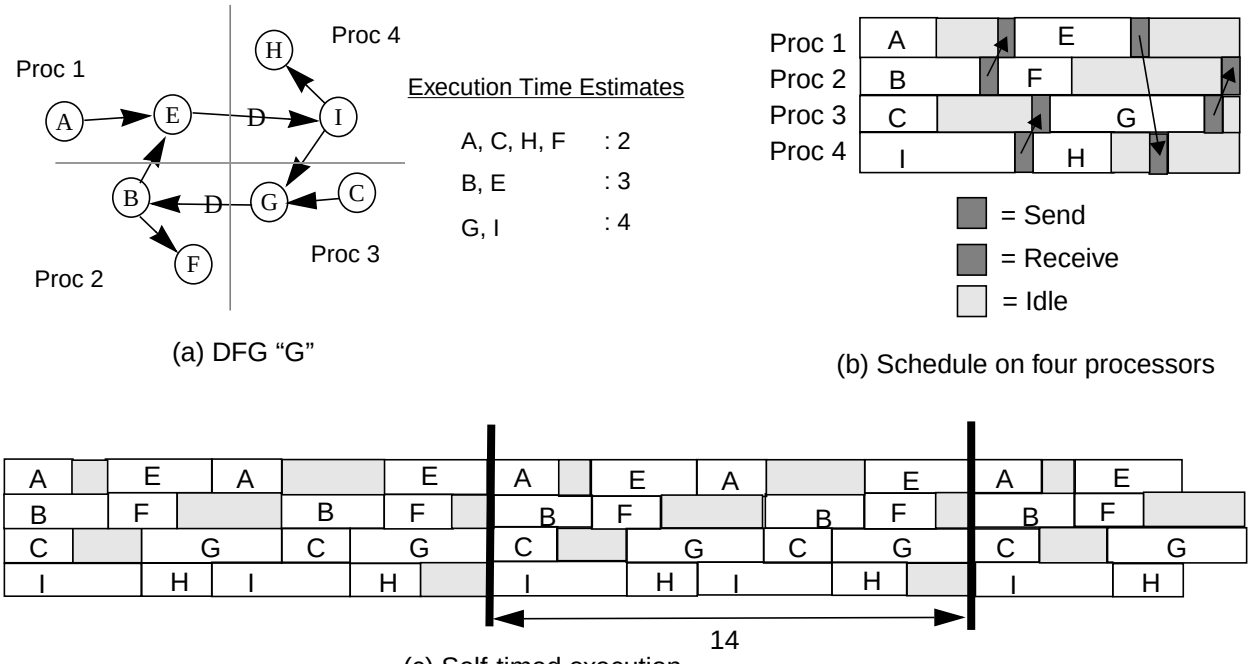


Figure 1. Self-timed execution.

as the **interprocessor communication modeling graph**, or **IPC graph** for short, models the sequential execution of actors that are assigned to the same processor, and it models constraints due to interprocessor communication. For example, the self-timed schedule in Figure 1(b) can be modeled by the IPC graph in Figure 1(d). The IPC edges are shown using dashed arrows. The rest of this subsection describes the construction of the IPC graph in detail.

The IPC graph has the same vertex set V as G, corresponding to the set of actors in G. The self-timed schedule specifies the actors assigned to each processor, and the order in which they execute. For example in 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 requirement that the kth execution of A must precede the kth execution of E, and the edge from E to A with a delay represents the constraint that the kth execution of A can occur only after the $(k-1)$th execution of E has completed. Thus if actors $v_1, v_2, \ldots, v_n$ are assigned to the same processor in that order, then G_{ipc} would have a cycle $((v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n), (v_n, v_1))$, with $delay((v_n, v_1)) = 1$. If there are P processors in the schedule, then we have P such cycles corresponding to each processor.

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

For each IPC edge in G we add an IPC edge e in G_{ipc} between the same actors. We also set the delay on this edge equal to the delay on the corresponding edge in G. Thus, we add an IPC edge from E to I in G_{ipc} with a single delay on it. The delay corresponds to the possibility that

execution of E can 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 this shared buffer. The number of initial tokens is equal to the delay on the IPC edge. In a straightforward self-timed implementation, each such IPC edge would also be a synchronization point between the two communicating processors.

The IPC graph has the same semantics as a DFG, and its execution models the execution of the corresponding self-timed schedule. As per the semantics of a DFG, each edge (v_j, v_i) of G_{ipc} represents the following data dependency constraint [41]:

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

where $start(v, k)$ and $end(v, k)$ respectively represent the time at which invocation k of actor v begins execution and completes execution. We set $start(v, k) = end(v, k) = 0$ for $k \leq 0$. Here time is modelled as an integer that can be considered a multiple of the base clock.

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 $t(v)$ with each vertex of the IPC graph; $t(v)$ assigns a positive integer execution time to each actor v.

The IPC graph can be viewed as a marked graph [33] or Reiter's computation graph [37]. Below we use some of the well-known properties of such graphs to measure the self-timed evolution of the system that corresponds to a given IPC graph.

Definition 1: The **cycle mean** of a cycle C, denoted $\lambda(C)$, in an IPC graph is defined by

$$\lambda(C) = \frac{\sum_{v \text{ is traversed by } C} t(v)}{Delay(C)}, \text{ and}$$

the maximum cycle mean of an IPC graph G_{ipc}, denoted $\lambda_{max}(G_{ipc})$, is defined by

$$\lambda_{max} = \max_{\text{cycle } C \text{ in } G} \{\lambda(C)\}.$$

A fundamental cycle in G_{ipc} whose cycle mean is equal to λ_{max} is called a **critical cycle** of G_{ipc}.

Note that for an IPC graph that is constructed from a schedule that is not deadlocked, the denominator in the expression for $\lambda(C)$ is necessarily positive [37]. Furthermore, in the self-timed schedule for G_{ipc}, an actor fires as soon as data is available at all of its input edges, and such an *as soon as possible (ASAP)* firing pattern implies that the throughput of the corresponding multiprocessor implementation is equal to $(\lambda_{max}(G_{ipc}))^{-1}$ [37].

For example, in Figure 1(d), G_{ipc} has one SCC, and its maximal cycle mean is 7 time units. This corresponds to the critical cycle $((B, E), (E, I), (I, G), (G, B))$: $t(B) = t(E) = 3$, $t(I) = t(G) = 4$ time units, so the total time along this cycle is 14, and there are two delays on this cycle. Thus the average iteration period for this schedule is 7 time units. We have not included IPC costs in this calculation, but these can be included in a straightforward manner by adding the *send* and *receive* costs to the corresponding actors performing these operations.

The maximum cycle mean can be calculated efficiently by repeated applications of the Bellman-Ford shortest path algorithm [24].

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

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

3.1 Synchronization protocols

In [4], we describe 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 **feedforward synchronization (FFS)**, 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, FFS 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 **feedback synchronization (FBS)**, that only explicitly ensures that underflow does not occur, and requires only two synchronization accesses per iteration period [4]. Such a simplified scheme is possible since the number of tokens that simultaneously reside on the buffer for a feedback edge can be shown to be bounded, and thus, it can be assumed that overflow will never occur if the buffer is sized appropriately [5].

3.2 The synchronization graph

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 FFS or FBS. To differentiate these two functions, we define another graph called the **synchronization graph**, in which edges between actors assigned to different processors, called **synchronization edges**, represent *synchronization constraints only.* An **execution source** of a synchronization graph is any actor that either has no input edges or has nonzero delay on all input edges.

Recall that an IPC edge (v_j, v_i) of G_{ipc} represents the **synchronization constraint**

$$start(v_i, k) \geq end(v_j, k - delay((v_j, v_i))), \ \forall k > delay(v_j, v_i). \qquad (2)$$

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, 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.

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 FFS and FBS. 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, which is developed in [5], underlies the validity of our synchronization optimizations.

Theorem 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 for all $\varepsilon \in E_2$ such that $\varepsilon \notin E_1$, we have $\rho_{G_1}(src(\varepsilon), snk(\varepsilon)) \leq delay(\varepsilon)$ — that is, if for each edge ε that is present in G_2 but not in G_1 there is a minimum delay path from $src(\varepsilon)$ to $snk(\varepsilon)$ in G_1 that has total delay of at most $delay(\varepsilon)$.

If $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$ are synchronization graphs with the same vertex-set and the same set of intraprocessor edges (edges that are not synchronization edges), then G_1 **preserves** G_2 if for all $\varepsilon \in E_2$ such that $\varepsilon \notin E_1$, we have $\rho_{G_1}(src(\varepsilon), snk(\varepsilon)) \leq delay(\varepsilon)$. Thus Theorem 1 guarantees that the synchronization edges of G_1 can be used to implement the synchronization constraints of G_2 if G_1 preserves G_2. It is easily verified that the *preserves* relation is transitive:

Fact 1: If G_1, G_2 and G_3 are synchronization graphs such that G_1 preserves G_2 and G_2 pre-

serves G_3, then G_1 preserves G_3.

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 which actors must wait on actions performed on remote processors, the iteration period of the resulting system is determined by the maximum cycle mean of G_s.

3.3 Redundant synchronization edges

Definition 2: A synchronization edge is **redundant** in a synchronization graph G if its removal yields a graph that preserves G. Equivalently from the definition of "preserves," a synchronization edge e is redundant if there is a path $p \neq (e)$ from $src(e)$ to $snk(e)$ such that $Delay(p) \leq delay(e)$.

Fig. 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 complete before each invocation of D is begun. Thus x_2 is redundant. From this example, we see that the synchronization function associated with a redun-

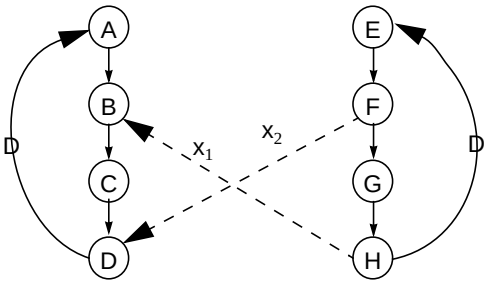


Figure 2. An example of a redundant synchronization edge: the edge x_2 is redundant.

dant synchronization edge “comes for free” as a by product of other synchronizations, and thus, it need not be implemented explicitly.

In [5], it is shown that if all redundant edges in a synchronization graph are removed, then the resulting graph preserves the original synchronization graph — that is, the redundancies are not interdependent. Thus, we need not implement the synchronization functions associated with any of the redundant synchronization edges. An efficient algorithm is given in [4] for determining the redundant synchronization edges of a synchronization graph. This algorithm is an extension to iterative dataflow programs of an earlier algorithm developed by Shaffer [39] that optimally removes redundant synchronization edges for the noniterative case.

3.4 Problem description

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 is $(4n_{ff} + 2n_{fb})$. The basic objective of the methods discussed in this paper is to minimize the synchronization cost.

In [5], a simple, efficient algorithm, called *Convert-to-SC-graph*, is described for introducing new synchronization edges so that the synchronization graph becomes strongly connected, which allows all synchronization edges to be implemented with FBS. A supplementary algorithm is also given for determining an optimal placement of delays on the new edges so that the estimated throughput is not degraded and the increase in shared memory buffer sizes is minimized. It is shown that the number of synchronization accesses required to implement the new edges that are added by *Convert-to-SC-graph* can be significantly less than the number of synchronization accesses that are eliminated by converting all uses of FFS to FBS. However, this technique may increase the latency.

In this paper, we propose another approach to reducing synchronization overhead called *resynchronization*. Resynchronization is also based on inserting new synchronization edges; the objective is to insert these edges so that the number of original synchronization edges that become

redundant is greater than the number of new synchronization edges. As with *Convert-to-SC-graph*, resynchronization can increase latency. We address the problem of optimal resynchronization both in the context where there is no restriction on latency, and in the context of a hard latency constraint that cannot be exceeded.

Generally, resynchronization can be viewed as complementary to the *Convert-to-SC-graph* optimization: resynchronization is performed first, followed by *Convert-to-SC-graph*. Under severe latency constraints, it may not be possible to accept the solution computed by *Convert-to-SC-graph*, in which case the feedforward edges that emerge from the resynchronized solution must be implemented with FFS. In such a situation, *Convert-to-SC-graph* can be attempted on the original (before resynchronization) graph to see if it achieves a better result than resynchronization without *Convert-to-SC-graph*. However, for synchronization graphs that have only one source SCC *and* only one sink SCC, the latency is not affected by *Convert-to-SC-graph*, and thus, for such systems resynchronization and *Convert-to-SC-graph* are fully complementary. This is fortunate since such systems arise frequently in practice.

3.5 Comparison with the resynchronization model of Filo, Ku, and De Micheli

As mentioned in Section 1, Filo, Ku and De Micheli have studied resynchronization in the context of minimizing the controller area for hardware synthesis of synchronization digital circuitry [11, 12], and significant differences in the underlying analytical models prevent these techniques from applying to our context. In the graphical hardware model of [12], called the *constraint graph* model, each vertex corresponds to a separate hardware device and edges have arbitrary weights that specify sequencing constraints. When the source vertex has bounded execution time, a positive weight $w(e)$ (*forward constraint*) imposes the constraint

$$start(snk(e)) \geq w(e) + start(src(e)),$$

while a negative weight (*backward constraint*) implies

$$start(snk(e)) \leq w(e) + start(src(e)).$$

If the source vertex has unbounded execution time, the forward and backward constraints are relative to the *completion* time of the source vertex. In contrast, in our synchronization graph model,

multiple actors can reside on the same processing element (implying zero synchronization cost between them), and the timing constraints always correspond to the case where $w(e)$ is positive and equal to the execution time of $src(e)$.

The implementation models, and associated implementation cost functions are also significantly different. A constraint graph is implemented using a scheduling technique called *relative scheduling* [22], which can roughly be viewed as intermediate between self-timed and fully-static scheduling. In relative scheduling, the constraint graph vertices that have unbounded execution time, called *anchors*, are used as reference points against which all other vertices are scheduled: for each vertex v, an offset f_i is specified for each anchor a_i that affects the activation of v, and v is scheduled to occur once f_i clock cycles have elapsed from the completion of a_i, for each i.

In the implementation of a relative schedule, each anchor has attached control circuitry that generates offset signals, and each vertex has a synchronization circuit that asserts an *activate* signal when all relevant offset signals are present. The resynchronization optimization is driven by a cost function that estimates the total area of the synchronization circuitry, where the offset circuitry area estimate for an anchor is a function of the maximum offset, and the synchronization circuitry estimate for a vertex is a function of the number of offset signals that must be monitored.

As a result of the significant differences in both the scheduling models and the implementation models, the techniques developed for resynchronizing constraint graphs do not extend in any straightforward manner to the resynchronization of synchronization graphs for self-timed multiprocessor implementation, and the solutions that we have developed for synchronization graphs are significantly different in structure from those reported in [12]. Most notably, the fundamental relationships that we develop between set covering and our use of resynchronization have not emerged in the context of constraint graphs.

4. Resynchronization

As discussed above, it is sometimes possible to reduce the total number of irredundant synchronization edges by adding new synchronization edges to a synchronization graph. We refer to the process of adding one or more new synchronization edges and removing the redundant

edges that result as *resynchronization* (defined more precisely below). Figure 3(a) illustrates this concept. Here, 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. Since redundant synchronization edges can be removed from the synchronization graph to yield an equivalent synchronization graph, we see that the net effect of adding the synchronization edge $d_0(C, H)$ is to reduce the number of synchronization edges that need to be implemented by 1. In Figure 3(b), we show the synchronization graph that results from inserting the *resynchronization edge* $d_0(C, H)$ into Figure 3(a), and then removing the redundant synchronization edges that result.

Definition 3 gives a formal definition of resynchronization that we will use throughout the remainder of this paper. This considers resynchronization only "across" feedforward edges. Resynchronization that includes inserting edges into the SCCs, is also possible; however, in general, such resynchronization may increase the estimated throughput (see Theorem 2 at the end of this section). Thus, for our objectives, it must be verified that each new synchronization edge introduced in an SCC does not decrease the estimated throughput. To avoid this complication, which requires a check of significant complexity ($O(|V||E|\log_2(|V|))$, where (V, E) is the modi-

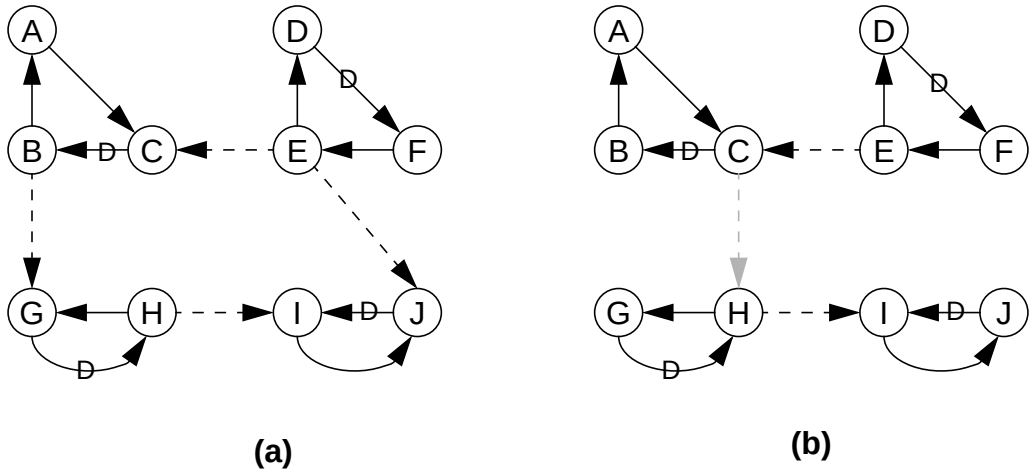


Figure 3. An example of resynchronization.

fied synchronization graph — this is using the Bellman Ford algorithm described in [24]) *for each* candidate resynchronization edge, we focus only on "feedforward" resynchronization in this paper. Future research may address combining the insights developed here for feedforward resynchronization with efficient techniques to estimate the impact that a given *feedback* resynchronization edge has on the estimated throughput.

Definition 3: Suppose that $G = (V, E)$ is a synchronization graph, and $F \equiv \{e_1, e_2, \ldots, e_n\}$ is the set of all feedforward edges in G. A **self-timed resynchronization** of G is a finite set $R \equiv \{e_1', e_2', \ldots, e_m'\}$ of edges that are not necessarily contained in E, but whose source and sink vertices are in V, such that (a). $e_1', e_2', \ldots, e_m'$ are feedforward edges in the DFG $G^* \equiv (V, ((E - F) + R))$; and (b). G^* preserves G — that is, $\rho_{G^*}(src(e_i), snk(e_i)) \leq delay(e_i)$ for all $i \in \{1, 2, \ldots, n\}$. Each member of R that is not in E is called a **resynchronization edge** of the resynchronization R, G^* is called the **resynchronized graph** associated with R, and this graph is denoted by $R(G)$.

Here we use the "self-timed" qualification to distinguish our context of resynchronization from the context of [12], which, as discussed in Section 3.5, is significantly different, and involves both the insertion of new synchronization edges (*serialization*), and the modification of selected edge weights (timing constraints). In the remainder of the paper, we drop the "self-timed" qualification, and when we use **resynchronization** in a technical sense, we mean the self-timed resynchronization concept defined in Definition 3.

If we let G denote the graph in Figure 3, then the set of feedforward edges is $F = \{(B, G), (E, J), (E, C), (H, I)\}$; $R = \{d_0(C, H), (E, C), (H, I)\}$ is a resynchronization of G; Figure 3(b) shows the DFG $G^* = (V, ((E - F) + R))$; and from Figure 3(b), it is easily verified that F, R, and G^* satisfy conditions (a) and (b) of Definition 3.

In the remainder of this section, we introduce a number of properties of resynchronization that we will use throughout the developments of this paper. The following definition is fundamental to these properties.

Definition 4: Suppose that G is a synchronization graph and R is a resynchronization of G. If s

is a synchronization edge in G that is not contained in R, we say that R **eliminates** s. If $s' \in R$, $s' \neq s$, and there is a path p from $src(s)$ to $snk(s)$ in $R(G)$ such that p contains s' and $Delay(p) \leq delay(s)$, then we say that s' **contributes to the elimination of** s.

A synchronization edge s can be eliminated if a resynchronization creates a path p from $src(s)$ to $snk(s)$ such that $Delay(p) \leq delay(s)$. In general, the path p may contain more than one resynchronization edge, and thus, it is possible that none of the resynchronization edges allows us to eliminate p "by itself". In such cases, it is the contribution of all of the resynchronization edges within the path p that enables the elimination of p. This motivates the our choice of terminology in Definition 4. An example is shown in Figure 4.

The following two facts follow immediately from Definition 4.

Fact 2: Suppose that G is a synchronization graph, R is a resynchronization of G, and r is a resynchronization edge in R. If r does not contribute to the elimination of any synchronization edges, then $(R - \{r\})$ is also a resynchronization of G. If r contributes to the elimination of one and only one synchronization edge s, then $(R - \{r\} + \{s\})$ is a resynchronization of G.

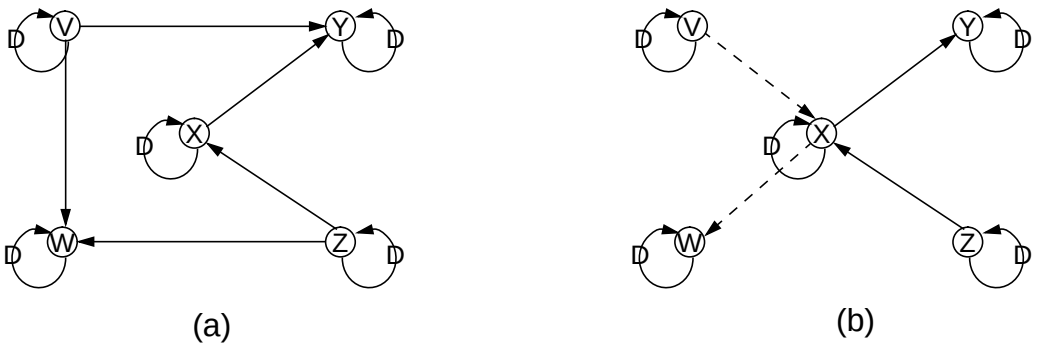


Figure 4. An illustration of Definition 4. Here each processor executes a single actor. A resynchronization of the synchronization graph in (a) is illustrated in (b). In this resynchronization, the resynchronization edges (V, X) and (X, W) both contribute to the elimination of (V, W).

Fact 3: Suppose that G is a resynchronization graph, R is a resynchronization of G, s is a synchronization edge in G, and s' is a resynchronization edge in R such that $delay(s') > delay(s)$. Then s' does not contribute to the elimination of s.

For example, let G denote the synchronization graph in Figure 5(a). Figure 5(b) shows a resynchronization R of G. In the resynchronized graph of Figure 5(b), the resynchronization edge (x_4, y_3) does not contribute to the elimination of any of the synchronization edges of G, and thus Fact 2 guarantees that $R' \equiv R - \{(x_4, y_3)\}$, illustrated in Figure 5(c), is also a resynchroniza-

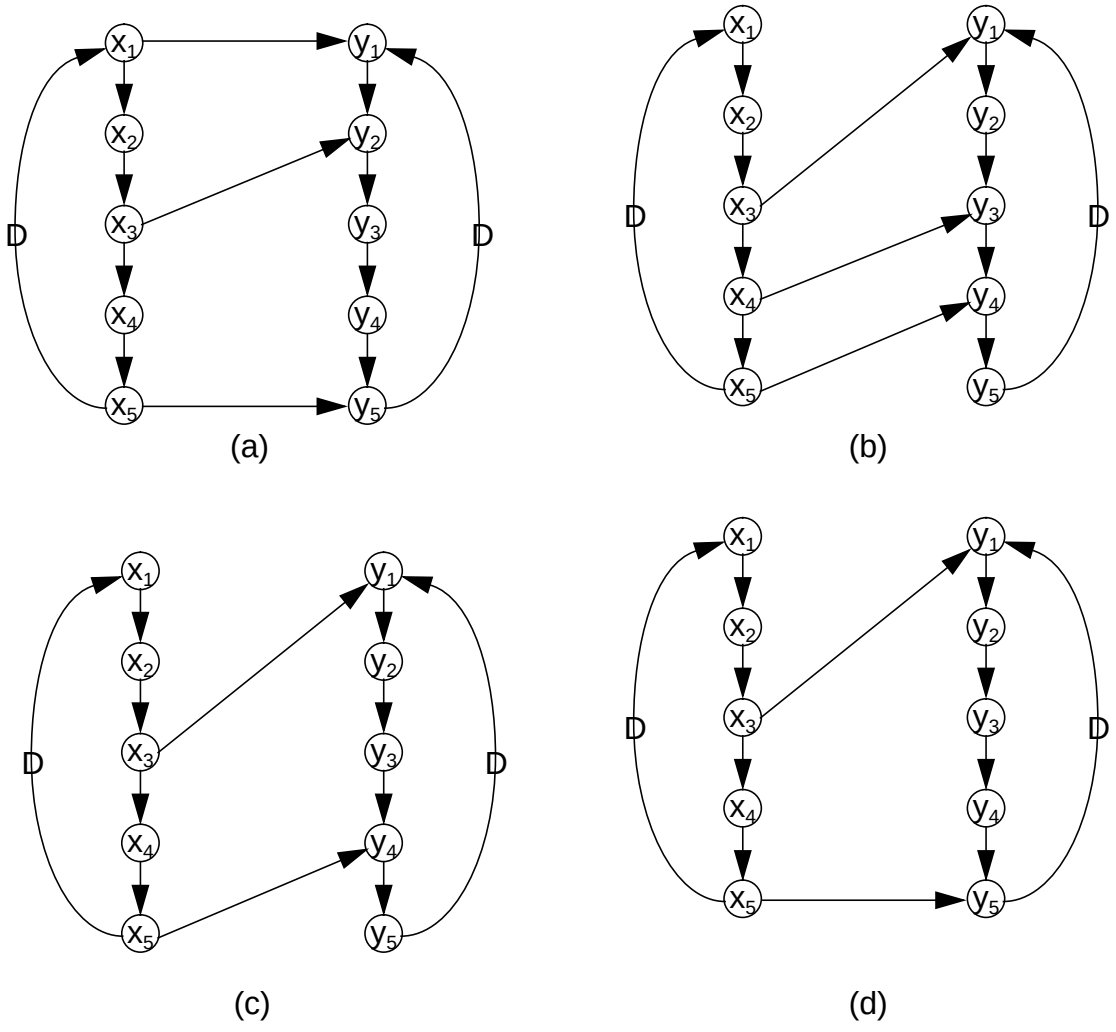


Figure 5. Properties of resynchronization.

tion of G. In Figure 5(c), it is easily verified that (x_5, y_4) contributes to the elimination of exactly one synchronization edge — the edge (x_5, y_5), and from Facts 1 and 2, we have that $R'' \equiv R' - \{(x_5, y_4)\} + \{(x_5, y_5)\}$, illustrated in Figure 5(d), is a also resynchronization of G.

Lemma 1: Suppose that G and G' are synchronization graphs such that G' preserves G, and p is a path in G from actor x to actor y. Then there is a path p' in G' from x to y such that $Delay(p') \leq Delay(p)$, and $tr(p) \subseteq tr(p')$, where $tr(\varphi)$ denotes the set of actors traversed by path φ.

In Figure 5(a), if we let $x = x_1$, $y = y_2$, and $p = ((x_1, y_1), (y_1, y_2))$, then the path $p' = ((x_1, x_2), (x_2, x_3), (x_3, y_1), (y_1, y_2))$ in Figure 5(b) confirms Lemma 1 for this example. Here $tr(p) = \{x_1, y_1, y_2\}$ and $tr(p') = \{x_1, x_2, x_3, y_1, y_2\}$.

Proof of Lemma 1: Let $p = (e_1, e_2, \ldots, e_n)$. By definition of the *preserves* relation, each e_i that is not a synchronization edge in G is contained in G'. For each e_i that is a synchronization edge in G, there must be a path p_i in G' from $src(e_i)$ to $snk(e_i)$ such that $Delay(p_i) \leq delay(e_i)$. Let $e_{i_1}, e_{i_2}, \ldots, e_{i_m}$, $i_1 < i_2 < \ldots < i_m$, denote the set of e_is that are synchronization edges in G, and define the path $\tilde{p}$ to be the concatenation

$$\langle (e_1, e_2, \ldots, e_{i_1 - 1}), p_1, (e_{i_1 + 1}, \ldots, e_{i_2 - 1}), p_2, \ldots, (e_{i_{m-1} + 1}, \ldots, e_{i_m - 1}), p_m, (e_{i_m + 1}, \ldots, e_n) \rangle .$$

Clearly, $\tilde{p}$ is a path in G' from x to y, and since $Delay(p_i) \leq delay(e_i)$ holds whenever e_i is a synchronization edge, it follows that $Delay(\tilde{p}) \leq Delay(p)$. Furthermore, from the construction of $\tilde{p}$, it is apparent that every actor that is traversed by p is also traversed by $\tilde{p}$. ■

Lemma 2: Suppose that G is a synchronization graph; R is a resynchronization of G; and (x, y) is a resynchronization edge such that $\rho_G(x, y) = 0$. Then (x, y) is redundant in $R(G)$. Thus, a minimal resynchronization (fewest number of elements) has the property that $\rho_G(x', y') > 0$ for each resynchronization edge (x', y').

Proof: Let p denote a minimum-delay path from x to y in G. Since (x, y) is a resynchronization edge, (x, y) is not contained in G, and thus, p traverses at least three actors. From Lemma 1,

it follows that there is a path p' in $R(G)$ from x to y such that $Delay(p') = 0$, and p' traverses at least three actors. Thus, $Delay(p') \leq delay((x, y))$ and $p' \neq (x, y)$, and we conclude that (x, y) is redundant in $R(G)$. ■

As a consequence of Lemma 1, the estimated throughput of a given synchronization graph is always less than or equal to that of every synchronization graph that it preserves.

Theorem 2: If R is a resynchronization of the synchronization graph G, then $\lambda_{max}(R(G)) \geq \lambda_{max}(G)$.

Proof: Suppose that C is a critical cycle in G. Lemma 1 guarantees that there is a cycle C' in $R(G)$ such that a) $Delay(C') \leq Delay(C)$, and b) the set of actors that are traversed by C is a subset of the set of actors traversed by C'. Now clearly, b) implies that

$$\sum_{v \text{ is traversed by } C'} t(v) \geq \sum_{v \text{ is traversed by } C} t(v), \tag{3}$$

and this observation together with a) implies that $\lambda(C') \geq \lambda(C)$. Since C is a critical cycle in G, it follows that $\lambda_{max}(R(G)) \geq \lambda_{max}(G)$. ■

Thus, in general, any saving in synchronization cost obtained by rearranging synchronization edges may come at the expense of a decrease in estimated throughput. As implied by Definition 3, we avoid this complication by restricting our attention to feedforward synchronization edges. Clearly, resynchronization that rearranges only feedforward synchronization edges cannot decrease the estimated throughput since no new cycles are introduced and no existing cycles are altered. Thus, with the form of resynchronization that we address in this paper, any decrease in synchronization cost that we obtain is not diminished by a degradation of the estimated throughput.

Before implementation, a synchronization graph must be made strongly connected to ensure that all IPC buffer sizes are bounded (in the absence of guarantees on actor execution times) [5]. As outlined in Section 3, this is done either by applying the FFS synchronization protocol, which effectively adds a feedback edge from $snk(e)$ to $src(e)$ for each feedforward synchro-

nization edge e, or by applying the *Convert-to-SC-graph* transformation, which permits use of the more efficient FBS synchronization protocol, and in general yields a lower overall synchronization cost than use of FFS [5]. By placing sufficient delay on certain edges, either of these techniques can be performed in such a way that the estimated throughput does not decrease. Thus, if we apply self-timed resynchronization to a synchronization graph that is not strongly connected, and then transform the resulting synchronization graph to a strongly connected graph using one of the two methods described above, there still will be no net degradation in estimated throughput.

5. Correspondence to set covering

We refer to the problem of finding a resynchronization with the fewest number of elements as the **resynchronization problem**. In [5], we formally show that the resynchronization problem is NP-hard; in this section, we explain the intuition behind this result. To establish the NP-hardness of the resynchronization problem, we examine a special case that occurs when there are exactly two SCCs, which we call the **pairwise resynchronization problem**, and we derive a polynomial-time reduction from the classic *set covering problem* [9], a well-known NP-hard problem, to the pairwise resynchronization problem. In the set covering problem, one is given a finite set X and a family T of subsets of X, and asked to find a minimal (fewest number of members) subfamily $T_s \subseteq T$ such that $\bigcup_{t \in T_s} t = X$. A subfamily of T is said to *cover* X if each member of X is contained in some member of the subfamily. Thus, the set covering problem is the problem of finding a minimal cover.

Although the correspondence that we establish between the resynchronization problem and set covering shows that the resynchronization problem probably cannot be attacked optimally with a polynomial-time algorithm, we will show that the correspondence allows any heuristic for set covering to be adapted easily into a heuristic for the pairwise resynchronization problem, and applying such a heuristic to each pair of SCCs in a general synchronization graph yields a heuristic for the general (not just pairwise) resynchronization problem. This is fortunate since the set covering problem has been studied in great depth, and efficient heuristic methods have been devised [9].

Definition 5: Given a synchronization graph G, let (x_1, x_2) be a synchronization edge in G, and let (y_1, y_2) be an ordered pair of actors in G. We say that (y_1, y_2) **subsumes** (x_1, x_2) in G if $\rho(x_1, y_1) + \rho(y_2, x_2) \leq delay((x_1, x_2))$. Thus, if $delay((x_1, x_2)) = 0$, then (y_1, y_2) subsumes (x_1, x_2) if $\rho(x_1, y_1) = \rho(y_2, x_2) = 0$. We may omit the qualification "in G" if the graph in question is understood from context.

Thus, in a synchronization graph $G = (V, E)$, (y_1, y_2) subsumes (x_1, x_2) if and only if a zero-delay synchronization edge directed from y_1 to y_2 makes (x_1, x_2) redundant.

The following fact is easily verified from Definitions 3 and 5.

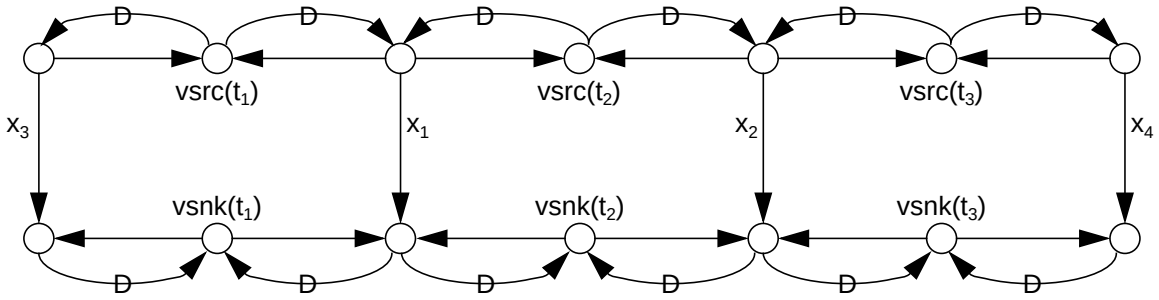
Fact 4: Suppose that G is a synchronization graph that contains exactly two SCCs, F is the set of feedforward edges in G, and F' is a resynchronization of G. Then for each $e \in F$, there exists $e' \in F'$ such that $(src(e'), snk(e'))$ subsumes e in G.

5.1 NP-hardness

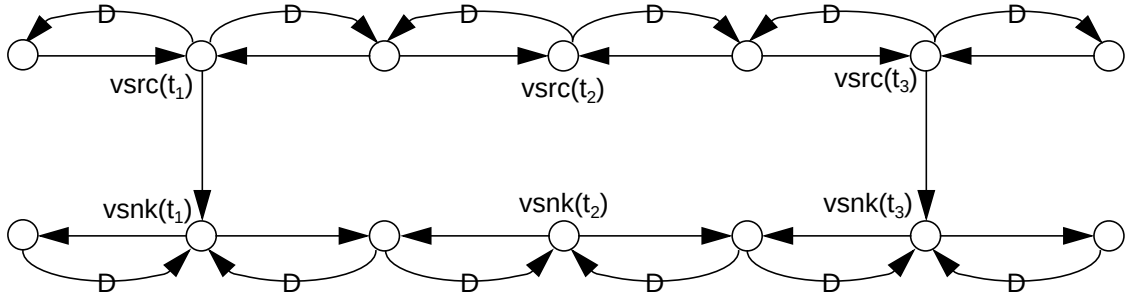
An intuitive correspondence between the pairwise resynchronization problem and the set covering problem can be derived from Fact 4. Suppose that G is a synchronization graph with exactly two SCCs C_1 and C_2 such that each feedforward edge is directed from a member of C_1 to a member of C_2. We start by viewing the set F of feedforward edges in G as the finite set that we wish to cover, and with each member p of $\{(x, y) | (x \in C_1, y \in C_2)\}$, we associate the subset of F defined by $\chi(p) \equiv \{e \in F | (p \text{ subsumes } e)\}$. Thus, $\chi(p)$ is the set of feedforward edges of G whose corresponding synchronizations can be eliminated if we implement a zero-delay synchronization edge directed from the first vertex of the ordered pair p to the second vertex of p. Clearly then, $\{e_1', e_2', \ldots, e_n'\}$ is a resynchronization if and only if each $e \in F$ is contained in at least one $\chi((src(e_i'), snk(e_i')))$ — that is, if and only if $\{\chi((src(e_i'), snk(e_i'))) \mid 1 \leq i \leq n\}$ covers F. Thus, solving the pairwise resynchronization problem for G is equivalent to finding a minimal cover for F given the family of subsets $\{\chi(x, y) | (x \in C_1, y \in C_2)\}$.

Figure 6 helps to illustrate this intuition and our method for converting an instance of the set covering problem to an instance of pairwise resynchronization. Suppose that we are given the set $X = \{x_1, x_2, x_3, x_4\}$, and the family of subsets $T = \{t_1, t_2, t_3\}$, where $t_1 = \{x_1, x_3\}$,

$t_2 = \{x_1, x_2\}$, and $t_3 = \{x_2, x_4\}$. To construct an instance of the pairwise resynchronization problem, we first create two vertices and an edge directed between these vertices *for each* member of X; we label each of the edges created in this step with the corresponding member of X. Then for each $t \in T$, we create two vertices $vsrc(t)$ and $vsnk(t)$. Next, for each relation $x_i \in t_j$ (there are six such relations in this example), we create two zero-delay edges — one directed from the



(a)



(b)

Figure 6. (a). An instance of the pairwise resynchronization problem that is derived from an instance of the set covering problem.

(b). The DFG that results from a solution to this instance of pairwise resynchronization.

source of the edge corresponding to x_i to $vsrc(t_j)$, and another directed from $vsnk(t_j)$ to the sink of the edge corresponding to x_i. This last step has the effect of making each pair $(vsrc(t_i), vsnk(t_i))$ preserve exactly those edges that correspond to members of t_i; in other words, after this construction, $\chi((vsrc(t_i), vsnk(t_i))) = t_i$, for each i. Finally, for each edge created in the previous step, we create a corresponding feedback edge oriented in the opposite direction, and having a unit delay.

Figure 6 shows the synchronization graph that results from this construction process. Here, it is assumed that each vertex corresponds to a separate processor; the associated unit delay, self loop edges are not shown to avoid excessive clutter. Observe that the graph contains two SCCs — $(\{src(x_i)\} \cup \{vsrc(t_i)\})$ and $(\{snk(x_i)\} \cup \{vsnk(t_i)\})$ — and that the set of feedforward edges is the set of edges that correspond to members of X. Now, recall that a major correspondence between the given instance of set covering and the instance of pairwise resynchronization defined by Figure 6(a) is that $\chi((vsrc(t_i), vsnk(t_i))) = t_i$, for each i. Thus, if we can find a minimal resynchronization of Figure 6(a) such that each edge in this resynchronization is directed from some $vsrc(t_k)$ to the corresponding $vsnk(t_k)$, then the associated t_k's form a minimum cover of X. For example, it is easy, albeit tedious, to verify that the resynchronization illustrated in Figure 6(b), $\{d_0(vsrc(t_1), vsnk(t_1)), d_0(vsrc(t_3), vsnk(t_3))\}$, is a minimal resynchronization of Figure 6(a), and from this, we can conclude that $\{t_1, t_3\}$ is a minimal cover for X. From inspection of the given sets X and T, it is easily verified that this conclusion is correct.

This example illustrates how an instance of pairwise resynchronization can be constructed (in polynomial time) from an instance of set covering, and how a solution to this instance of pairwise resynchronization can easily be converted into a solution of the set covering instance. Our proof of the NP-hardness of pairwise resynchronization, presented in [5], is a generalization of the example in Figure 6.

We summarize with the following theorem.

Theorem 3: The pairwise resynchronization problem is NP-hard, and thus, the resynchronization problem is NP-hard.

Proof: A formal proof is given in [5].

5.2 A family of heuristics

We have pointed out that the correspondence we have established between set covering and pairwise resynchronization allows us to adapt any heuristic for set covering into a heuristic for pairwise resynchronization. Furthermore applying such a heuristic for pairwise resynchronization to each pair of SCCs in a general synchronization graph gives a heuristic for the general resynchronization problem. Figure 7 below shows how any algorithm *Cover* that solves the set covering problem can be applied to derive a heuristic algorithm for resynchronization.

6. Chaining synchronization graphs

In this section, we define a broad class of synchronization graphs for which optimal resynchronizations can be computed efficiently using a simple *chaining* procedure. Such a solution, when applicable, can be viewed as a pipelined implementation in which each SCC of the synchronization graph corresponds to a pipeline stage.

Definition 6: Suppose that C is an SCC in a synchronization graph G, and x is an actor in C. Then x is an **input hub** of C if for each synchronization edge e in G whose sink actor is in C, we have $\rho_C(x, snk(e)) = 0$. Similarly, x is an **output hub** of C if for each synchronization edge e in G whose source actor is in C, we have $\rho_C(src(e), x) = 0$. We say that C is **linkable** if there exist actors x, y in C such that x is an input hub, y is an output hub, and $\rho_C(x, y) = 0$. A synchronization graph is **chainable** if each SCC is linkable.

For example, consider the SCC in Figure 8(a), and assume that the dashed edges represent the synchronization edges that connect this SCC with other SCCs. This SCC has exactly one input hub, actor A, and exactly one output hub, actor F, and since $\rho(A, F) = 0$, it follows that the SCC is linkable. However, if we remove the edge (C, F), then the resulting graph (shown in Figure 8(b)) is not linkable since it does not have an output hub. A class of linkable SCCs that occur commonly in practical synchronization graphs are those SCCs that corresponds to only one processor, such as the SCC shown in Figure 8(c). In such cases, the first actor executed on the proces-

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

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

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

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

Figure 7. An algorithm for resynchronization that is derived from an arbitrary algorithm *Cover* for the set covering problem

sor is always an input hub and the last actor executed is always an output hub.

In the remainder of this section, we assume that for each linkable SCC, an input hub x and output hub y are selected such that $\rho(x, y) = 0$, and these actors are referred to as the **selected input hub** and the **selected output hub** of the associated SCC. Which input hub and output hub are chosen as the "selected" ones makes no difference to our discussion of the techniques in this section as long they are selected so that $\rho(x, y) = 0$.

An important property of linkable synchronization graphs is that if C_1 and C_2 are distinct linkable SCCs, then all synchronization edges directed from C_1 to C_2 are preserved by the single ordered pair (l_1, l_2), where l_1 denotes the selected output hub of C_1 and l_2 denotes the selected

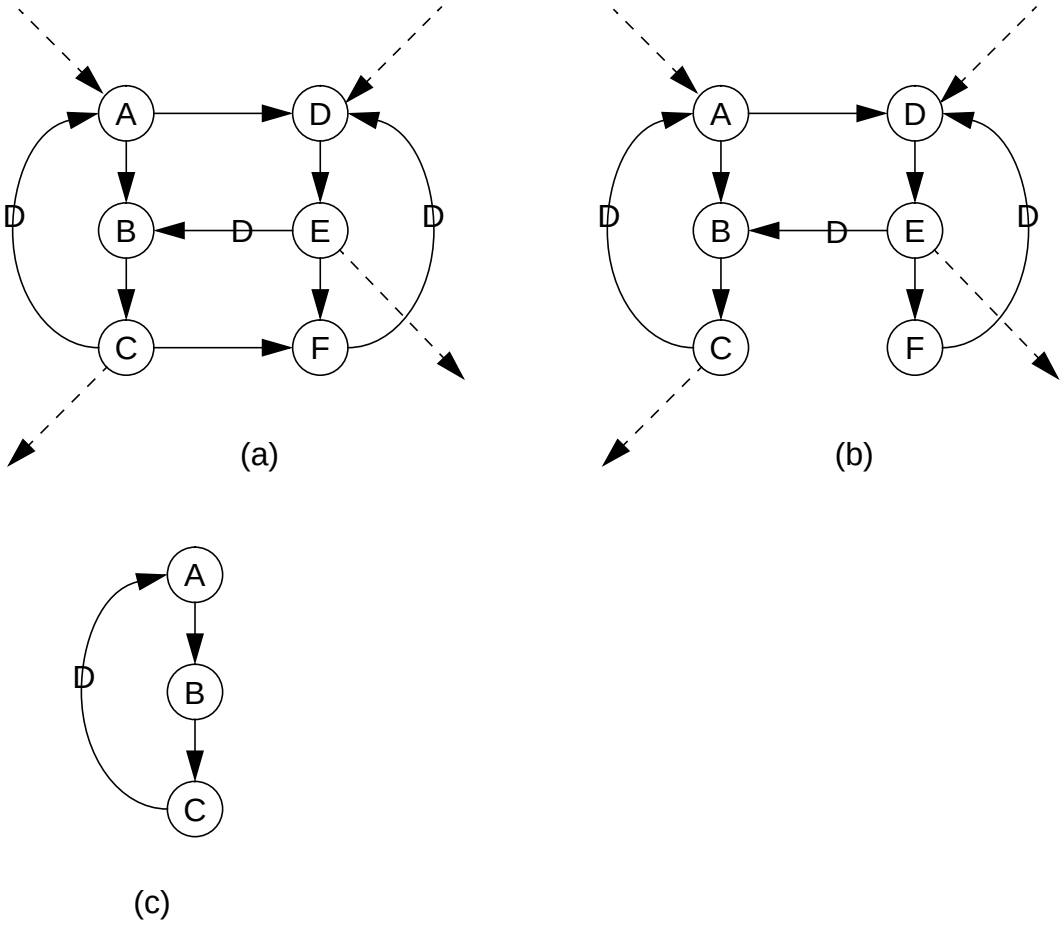
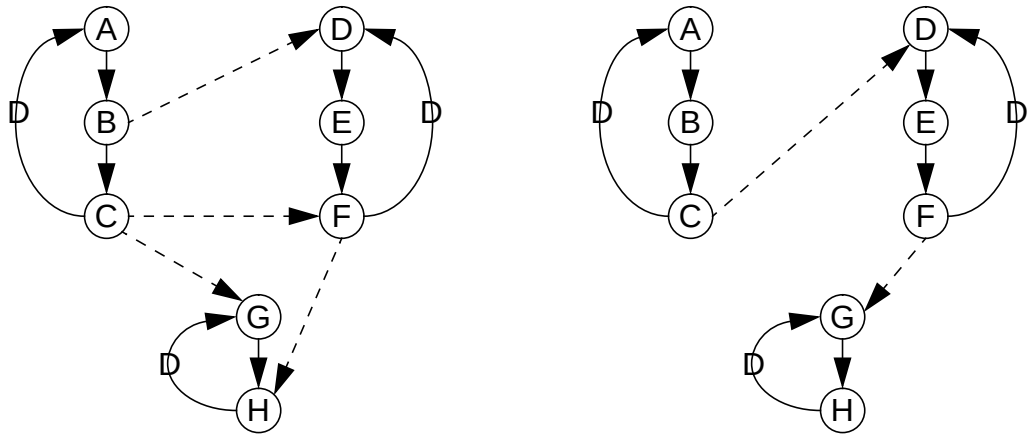


Figure 8. An illustration of input and output hubs for synchronization graph SCCs.

input hub of C_1. Furthermore, if there exists a path between two SCCs C_1', C_2' of the form $((o_1, i_2), (o_2, i_3), \ldots, (o_{n-1}, i_n))$, where o_1 is the selected output hub of C_1', i_n is the selected input hub of C_2', and there exist distinct SCCs $Z_1, Z_2, \ldots, Z_{n-2} \notin \{C_1', C_2'\}$ such that for $k = 2, 3, \ldots, (n-1)$, i_k, o_k are respectively the selected input hub and the selected output hub of Z_{k-1}, then all synchronization edges between C_1' and C_2' are redundant.

From these properties, an optimal resynchronization for a chainable synchronization graph can be constructed efficiently by computing a topological sort of the SCCs, instantiating a zero delay synchronization edge from the selected output hub of the ith SCC in the topological sort to the selected input hub of the $(i+1)$th SCC, for $i = 1, 2, \ldots, (n-1)$, where n is the total number of SCCs, and then removing all of the redundant synchronization edges that result. For example, if this algorithm is applied to the chainable synchronization graph of Figure 9(a), then the synchronization graph of Figure 9(b) is obtained, and the number of synchronization edges is reduced from 4 to 2.



(a) (b)

Figure 9. An illustration of a simple algorithm for optimal resynchronization of chainable synchronization graphs. The dashed edges correspond to synchronization edges.

This simple chaining technique corresponds to pipelining, where each SCC in the output synchronization graph corresponds to a pipeline stage. Pipelining has been used extensively to increase throughput via improved parallelism ("temporal parallelism") in multiprocessor DSP implementations (see for example, [2, 15, 31]). However, in our application of pipelining, the load of each processor is unchanged, and the estimated throughput is not affected (since no new cyclic paths are introduced), and thus, the benefit to the *overall* throughput of our chaining technique arises chiefly from the optimal reduction of synchronization overhead.

This technique can be generalized to optimally resynchronize a somewhat broader class of synchronization graphs. This class consists of all synchronization graphs for which each source SCC has an output hub (but not necessarily an input hub), each sink SCC has an input hub (but not necessarily an output hub), and each internal SCC is linkable. In this case, the internal SCCs are pipelined as in the previous algorithm, and then each for each source SCC, a synchronization edge is inserted from one of its output hubs to the selected input hub of the first SCC in the pipeline of internal SCCs, and for each sink SCC, a synchronization edge is inserted to one of its input hubs from the selected output hub of the last SCC in the pipeline of internal SCCs. If there are no internal SCCs, then the sink SCCs are pipelined by selecting one input hub from each SCC, and joining these input hubs with a chain of synchronization edges, and then inserting a synchronization edge from an output hub of each source SCC to an input hub of the first SCC in the chain of sink SCCs.

As implied earlier, although the techniques described in this section preserve the estimated throughput, they may result in significantly increased latency. For example in Figure 9, the latency of the resynchronized solution is the sum of the execution times of all actors in the graph. In the following section, we address a problem that becomes relevant when such a latency increase cannot be tolerated, but there is some margin within which the latency can be increased. This is the problem of optimally reducing the number of synchronization edges without increasing the latency beyond some pre-specified threshold L_{max}.

A somewhat analogous conflict between latency and synchronization overhead is discussed in a significantly different context in [43]. Here the objective is to reduce the probability of metastable states in a digital circuit. This probability can be reduced arbitrarily by passing a signal

through a sufficiently long chain of flip flops. However, alternative techniques that have minimal impact on latency have been devised for a restricted class of problems [43].

7. Latency-constrained resynchronization

As discussed in section 4, resynchronization cannot decrease the estimated throughput since it manipulates only the feedforward edges of a synchronization graph. Frequently in real-time DSP systems, latency is also an important issue, and although resynchronization does not degrade the estimated throughput, it generally does increase the latency.

In this section we define the *latency-constrained resynchronization problem* for self-timed multiprocessor systems, and we show that although the resynchronization problem has a simple polynomial time solution for the class of chainable synchronization graphs, the latency-constrained resynchronization problem is NP-hard even if the input graph is assumed to be chainable.

7.1 Problem statement

Definition 7: Suppose G_0 is a DFG, G is a synchronization graph that results from a multiprocessor schedule for G_0, x is an execution source in G, and y is an actor in G other than x, then we define the **latency** from x to y by $L_G(x, y) \equiv end(y, 1 + \rho_{G_0}(x, y))$ [1]. We refer to x as the **latency input** associated with this measure of latency, and we refer to y as the **latency output**.

Intuitively, the latency is the time required for the first invocation of the latency input to influence the associated latency output. More precisely, it is the earliest time at which an invocation y_k of y executes such that there is a sequence of invocations $i_1, i_2, \ldots, i_n$, where $i_1 = x_1$, $i_n = y_k$ and for $1 \leq k < n$, the semantics of the DFG G_0 imply that at least one token produced by i_k is consumed by i_{k+1}.

Note that our measure of latency is explicitly concerned only with the time that it takes for the *first* input to propagate to the output, and does not in general give an upper bound on the time

1. Recall that $start(v, k)$ and $end(v, k)$ denote the time at which invocation k of actor v commences and completes execution. Also, note that $start(x, 1) = 0$ since x is an execution source.

for subsequent inputs to influence subsequent outputs. Extending our latency measure to maximize over all pairs of "related" input and output invocations would yield the alternative measure L_G' defined by

$$L_G'(x, y) = max(\{end(y, k + \rho_{G_0}(x, y)) - start(x, k) \mid (k = 1, 2, \ldots)\}). \quad (4)$$

Currently, there are no known tight upper bounds on L_G' that can be computed efficiently from the synchronization graph for any useful subclass of graphs, and thus, we use the lower bound approximation L_G, which corresponds to the critical path, when attempting to analyze and optimize the input-output propagation delay of a self-timed system. The heuristic that we present in Section 9 for latency-constrained resynchronization can easily be adapted to handle arbitrary latency measures; however, the efficiency of the heuristic depends on the existence of an algorithm to efficiently compute the change in latency that arises from inserting a single new synchronization edge. The exploration of incorporating alternative measures — or estimates — of latency in this heuristic framework, possibly with adaptations to the basic framework, would be a useful area for further study.

Before defining the complexity of the latency-constrained resynchronization problem, we define a class of synchronization graphs for which our latency measure can be computed efficiently. In words, this is simply the class of graphs in which the first invocation of the latency output is influenced by the first invocation of the latency input. Equivalently, it is the class of graphs that have at least one delayless path in the DFG directed from the latency input to the latency output.

Definition 8: Suppose that G_0 is a DFG, x is a source actor in G, and y is actor in G that is not identical to x. If $\rho_{G_0}(x, y) = 0$, then we say that G_0 is **transparent** with respect to latency input x and latency output y. If G is a synchronization graph that corresponds to a multiprocessor schedule for G_0, we also say that G is **transparent**.

If a synchronization graph is transparent with respect to a latency input/output pair, then the latency can be computed efficiently using longest path calculations on an *acyclic* graph that is derived from the input synchronization graph G. This acyclic graph, which we call the **first-iter-**

ation graph of G, denoted $\mathfrak{fi}(G)$ is constructed by removing all edges from G that have non-zero-delay, adding a vertex υ, setting $t(\upsilon) = 0$, and adding delayless edges from υ to each source actor (other than υ) of the partial construction until the only source actor that remains is υ. Figure 10 illustrates the derivation of $\mathfrak{fi}(G)$.

Given two vertices x and y in $\mathfrak{fi}(G)$ such that there is a path in $\mathfrak{fi}(G)$ from x to y, we denote the sum of the execution times along a path from x to y that has maximum cumulative execution time by $T_{\mathfrak{fi}(G)}(x, y)$. That is,

$$T_{\mathfrak{fi}(G)}(x, y) = max\left(\sum_{p \text{ traverses z}} t(z) \middle| (p \text{ is a path from } x \text{ to } y \text{ in } \mathfrak{fi}(G)) \right). \quad (5)$$

If there is no path from x to y, then we define $T_{\mathfrak{fi}(G)}(x, y)$ to be $-\infty$. Note that for all x, y $T_{\mathfrak{fi}(G)}(x, y) < +\infty$, since $\mathfrak{fi}(G)$ is acyclic. The values $T_{\mathfrak{fi}(G)}(x, y)$ for all pairs x, y can be computed in $O(n^3)$ time, where n is the number of actors in G, by using a simple adaptation of the Floyd-Warshall algorithm specified in [9].

Fact 5: Suppose that G_0 is a DFG that is transparent with respect to latency input x and latency output y, G_s is the synchronization graph that results from a multiprocessor schedule for G_0, and

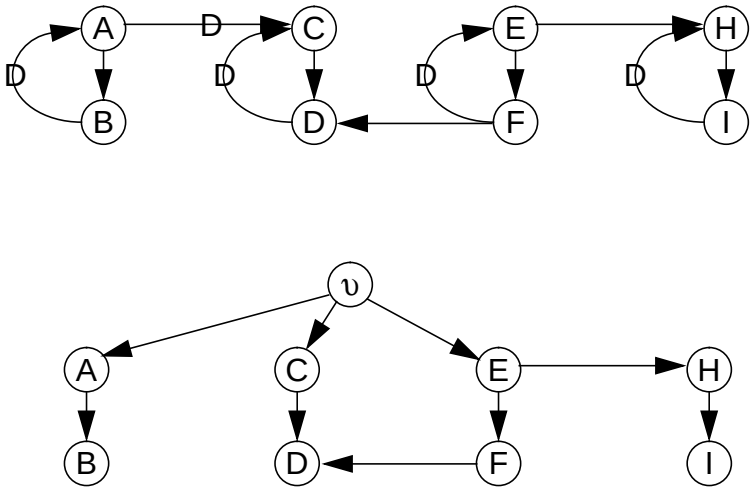


Figure 10. An example used to illustrate the construction of $\mathfrak{fi}(G)$. The lower graph is $\mathfrak{fi}(G)$ when G is the upper graph.

G is a resynchronization G_s. Then $\rho_G(x, y) = 0$, and thus $T_{\mathit{fi}(G)}(x, y) \geq 0$.

Proof: Since G_0 is transparent, there is a delayless path p in G_0 from x to y. Let $(u_1, u_2, \ldots, u_n)$, where $x = u_1$ and $y = u_n$, denote the sequence of actors traversed by p. From the semantics of the DFG G_0, it follows that for $1 \leq i < n$, either u_i and u_{i+1} execute on the same processor, with u_i scheduled earlier than u_{i+1}, or there is a zero-delay synchronization edge in G_s directed from u_i to u_{i+1}. Thus, for $1 \leq i < n$, we have $\rho_{G_s}(u_i, u_{i+1}) = 0$, and thus, that $\rho_{G_s}(x, y) = 0$. Since G is a resynchronization of G_s, it follows from Lemma 1 that $\rho_G(x, y) = 0$. ■

The following theorem gives an efficient means for computing the latency L_G for transparent synchronization graphs.

Theorem 4: Suppose that G is a synchronization graph that is transparent with respect to latency input x and latency output y. Then $L_G(x, y) = T_{\mathit{fi}(G)}(\upsilon, y)$.

Proof: By induction, we show that

$$\mathit{end}(w, 1) = T_{\mathit{fi}(G)}(\upsilon, w) \text{ for every actor } w \text{ in } \mathit{fi}(G), \tag{6}$$

which clearly implies the desired result.

First, let $mt(w)$ denote the maximum number of actors that are traversed by a path in $\mathit{fi}(G)$ (over all paths in $\mathit{fi}(G)$) that starts at υ and terminates at w. If $mt(w) = 1$, then clearly $w = \upsilon$. Since both the LHS and RHS of (6) are identically equal to $t(\upsilon) = 0$ when $w = \upsilon$, we have that (6) holds whenever $mt(w) = 1$.

Now suppose that the result holds whenever $mt(w) \leq k$, for some $k \geq 1$, and consider the scenario $mt(w) = k + 1$. Clearly, in the self-timed (ASAP) execution of G, invocation w_1 commences as soon as all invocations in the set

$$Z = \{z_1 \mid z \text{ is a predecessor of } w \text{ in } \mathit{fi}(G)\}$$

have completed execution. All members $z \in Z$ satisfy $mt(z) \leq k$, since otherwise $mt(w)$ would exceed $(k + 1)$. Thus, from the induction hypothesis, we have

$$start(w, 1) = max(end(z, 1) | z_1 \in Z) = max(T_{\mathcal{fl}(G)}(\upsilon, z) \big| (z \in Z)),$$

which implies that

$$end(w, 1) = max(T_{\mathcal{fl}(G)}(\upsilon, z) \big| (z \in Z)) + t(w). \quad (7)$$

But, by definition of $T_{\mathcal{fl}(G)}$, the RHS of (7) is clearly equal to $T_{\mathcal{fl}(G)}(\upsilon, w)$, and thus we have that $end(w, 1) = T_{\mathcal{fl}(G)}(\upsilon, w)$.

We have shown that (6) holds for $mt(w) = 1$, and that whenever it holds for $mt(w) = k \geq 1$, it must hold for $mt(w) = (k + 1)$. Thus, (6) holds for all values of $mt(w)$. ■

Theorem 4 shows that latency can be computed efficiently for transparent synchronization graphs. A further benefit of transparent synchronization graphs is that the change in latency induced by adding a new synchronization edge (a "resynchronization operation") can be computed in $O(1)$ time, given $T_{\mathcal{fl}(G)}(a, b)$ for all actor pairs (a, b). We will discuss this further, as well as its application to developing an efficient resynchronization heuristic, in Section 9.

Definition 9: An instance of the **latency-constrained synchronization problem** consists of a transparent synchronization graph G with latency input x and latency output y, and a *latency constraint* $L_{max} \geq L_G(x, y)$. A solution to such an instance is a resynchronization R (if one exists) such that (1) $L_{R(G)}(x, y) \leq L_{max}$, and (2) no resynchronization of G that results in a latency less than or equal to L_{max} has smaller cardinality than R.

Given a transparent synchronization graph G with latency input x and latency output y, and a latency constraint L_{max}, we say that a resynchronization R of G is a **latency-constrained resynchronization (LCR)** if $L_{R(G)}(x, y) \leq L_{max}$. Thus, the latency-constrained resynchronization problem is the problem of determining a minimal LCR.

7.2 NP-hardness

Recall that optimal solutions for the resynchronization problem can be computed in polynomial time for chainable synchronization graphs. In this subsection we show that latency-constrained resynchronization is NP-hard even for the very restricted subclass of chainable synchronization graphs in which each SCC corresponds to a single actor, and all synchronization edges have zero delay. Thus, the algorithms in Section 6 cannot be extended to yield polynomial time algorithms for optimal latency-constrained resynchronization on the same classes of graphs.

As with the resynchronization problem, the intractability of latency-constrained resynchronization can be established by a reduction from set covering. To illustrate this reduction, we suppose, as in the illustration of our reduction for the resynchronization problem, that we are given the set $X = \{x_1, x_2, x_3, x_4\}$, and the family of subsets $T = \{t_1, t_2, t_3\}$, where $t_1 = \{x_1, x_3\}$, $t_2 = \{x_1, x_2\}$, and $t_3 = \{x_2, x_4\}$. Figure 11 illustrates the instance of latency-constrained resynchronization that we derive from the instance of set covering specified by (X, T). As in Figure 6, each actor corresponds to a single processor and the self loop edges for each actor are not shown. The numbers beside the actors specify the actor execution times, and the latency constraint is $L_{max} = 103$. In the graph of Figure 11, which we denote by G, the edges labeled ex_1, ex_2, ex_3, ex_4 correspond respectively to the members x_1, x_2, x_3, x_4 of the set X in the set covering instance, and the vertex pairs (resynchronization candidates) $(v, st_1), (v, st_2), (v, st_3)$ correspond to the members of T. For each relation $x_i \in t_j$, an edge exists that is directed from st_j to sx_i. The latency input and latency output are defined to be in and out respectively, and it is assumed that G is transparent.

The synchronization graph that results from an optimal resynchronization of G is shown in Figure 11, with redundant resynchronization edges removed. Since the resynchronization candidates $(v, st_1), (v, st_3)$ were chosen to obtain the solution shown in Figure 11, this solution corresponds to the solution of (X, T) that consists of the subfamily $\{t_1, t_3\}$.

A correspondence between the set covering instance (X, T) and the instance of latency-constrained resynchronization defined by Figure 11 arises from two properties of our construction:

Observation 1: $(x_i \in t_j$ in the set covering instance $) \Leftrightarrow ((v, st_j)$ subsumes ex_i in $G)$.

Observation 2: If R is an optimal LCR of G, then

each resynchronization edge in R is of the form

$(v, st_i), i \in \{1, 2, 3\}$, or of the form $(st_j, sx_i), x_i \notin t_j$. (8)

The first observation is immediately apparent from inspection of Figure 11. The second observation is justified in the following proof.

Proof of Observation 2: We must show that no other resynchronization edges can be contained in an optimal LCR of G. Figure 13 specifies arguments with which we can discard all possibilities

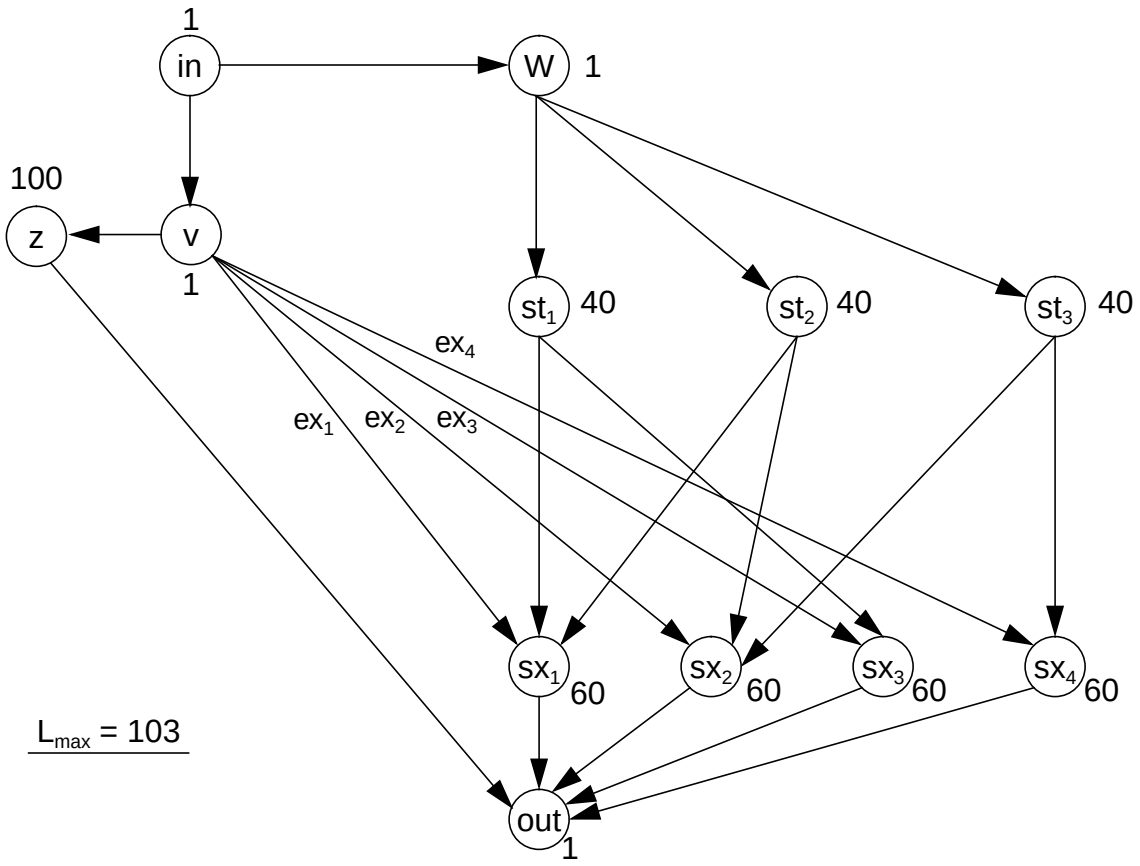


Figure 11. An instance of latency-constrained resynchronization that is derived from an instance of the set covering problem.

other than those given in (8). In the matrix shown in Figure 13, the entry corresponding to row r and column c specifies an index into the list of arguments on the right side of the figure. For each of the six categories of arguments, except for #6, the reasoning is either obvious or easily understood from inspection of Figure 11. For example, edge (v, z) cannot be a resynchronization edge in R because the edge already exists in the original synchronization graph; an edge of the form (sx_j, w) cannot be in R because there is a path in G from w to each sx_i; $(z, w) \notin R$ since otherwise there would be a path from in to out that traverses v, z, w, st_1, sx_1, and thus, the latency

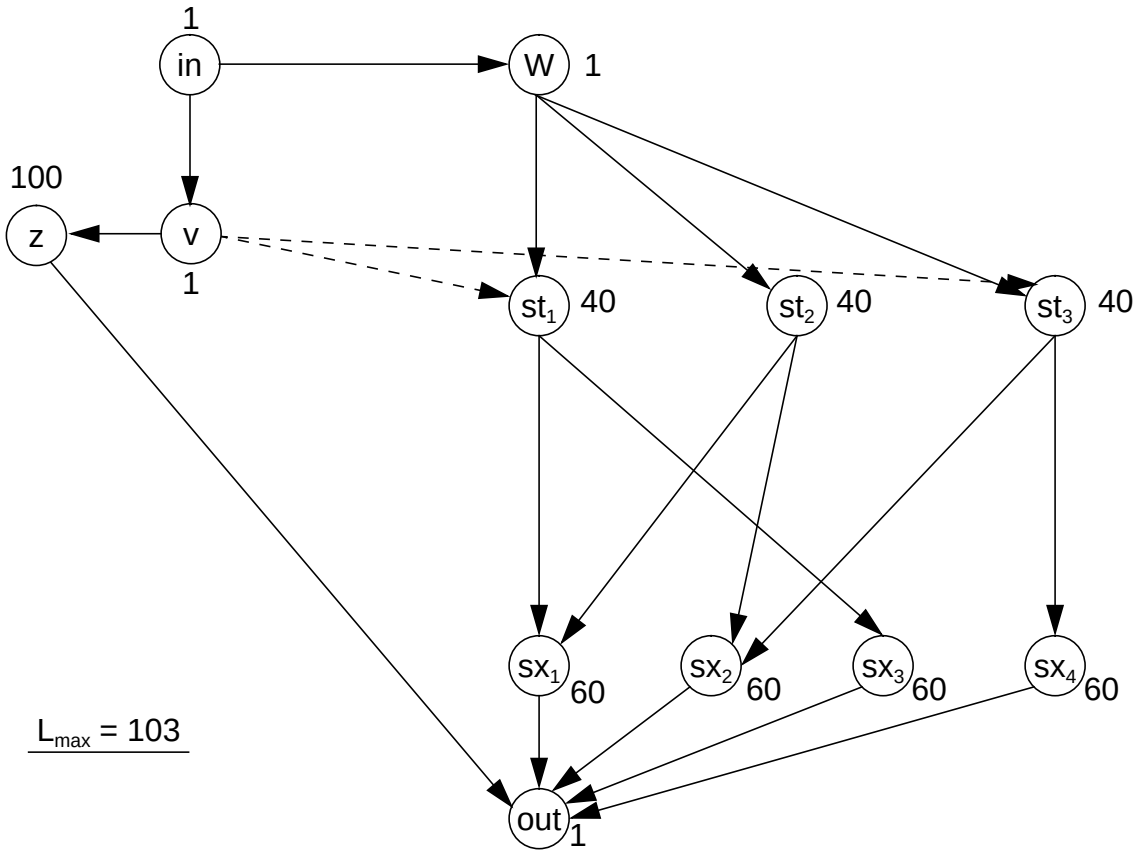


Figure 12. The synchronization graph that results from a solution to the instance of latency-constrained resynchronization shown in Figure 11.

would be increased to at least 204; $(in, z) \notin R$ from Lemma 2 since $\rho_G(in, z) = 0$; and $(v, v) \notin R$ since otherwise there would be a delayless self loop. Three of the entries in Figure 13 point to multiple argument categories. For example, if $x_j \in t_i$, then (sx_j, st_i) introduces a cycle, and if $x_j \notin t_i$ then (sx_j, st_i) cannot be contained in R because it would increase the latency beyond L_{max}.

The entries in Figure 13 marked *OK* are simply those that correspond to (8), and thus we have justified Observation 2. ■

Observation 3: Suppose that R is an optimal LCR of G and suppose that $e = (st_j, sx_i)$ is a resynchronization edge in R, for some $i \in \{1, 2, 3, 4\}$, $j \in \{1, 2, 3\}$ such that $x_i \notin t_j$. Then e contributes to the elimination of one and only one synchronization edge — ex_i.

Proof: See Appendix B.

Now, suppose that we are given an optimal LCR R of G. From Observation 3 and Fact 2,

	v	w	z	in	out	st_i	sx_i
v	5	3	1	2	4	OK	1
w	3	5	6	2	4	1	4
z	2	3	5	2	1	3	3
in	1	1	4	5	4	4	4
out	2	2	2	2	5	2	2
st_j	3	2	3	2	4	3/5	OK[a]
sx_j	2	2	3	2	1	2/3	3/5

a. Assuming that $x_j \notin t_i$; otherwise 1 applies.

1. Exists in G.
2. Introduces a cycle.
3. Increases the latency beyond L_{max}.
4. $\rho_G(a_1, a_2) = 0$ (Lemma 2).
5. Introduces a delayless self loop.
6. See Appendix A.

Figure 13. Arguments that support Observation 2.

we have that for each resynchronization edge (st_j, sx_i) in R, we can replace this resynchronization edge with ex_i and obtain another optimal LCR. Thus from Observation 2, we can efficiently obtain an optimal LCR R' such that the only resynchronization edges in R' are of the form (v, st_i).

For each $x_i \in X$ such that

$$\exists t_j \big| ((x_i \in t_j) \textbf{ and } ((v, st_j) \in R')), \tag{9}$$

we have that $ex_i \notin R'$. This is because R' is assumed to be optimal, and thus, $R(G)$ contains no redundant synchronization edges. For each $x_i \in X$ for which (9) does not hold, we can replace ex_i with any (v, st_j) that satisfies $x_i \in t_j$, and since such a replacement does not affect the latency, we know that the result will be another optimal LCR for G. In this manner, if we repeatedly replace each ex_i that does not satisfy (9) then we obtain an optimal LCR R'' such that

$$\text{each resynchronization edge in } R'' \text{ is of the form } (v, st_i), \text{ and} \tag{10}$$

$$\text{for each } x_i \in X, \text{ there exists a resynchronization edge } (v, t_j) \text{ in } R'' \text{ such that } x_i \in t_j. \tag{11}$$

It is easily verified that the set of synchronization edges eliminated by R'' is $\{ex_i \big| x_i \in X\}$. Thus, the set $T' \equiv \{t_j \big| (v, t_j)$ is a resynchronization edge in $R''\}$ is a cover for X, and the cost (number of synchronization edges) of the resynchronization R'' is $(N - |X| + |T'|)$, where N is the number of synchronization edges in the original synchronization graph. Now, it is also easily verified (from Figure 11) that given an arbitrary cover T_a for X, the resynchronization defined by

$$R_a \equiv (R'' - \{(v, t_j) \big| (t_j \in T')\}) + \{(v, t_j) \big| (t_j \in T_a)\} \tag{12}$$

is also a valid LCR of G, and that the associated cost is $(N - |X| + |T_a|)$. Thus, it follows from the optimality of R'' that T' must be a minimal cover for X, given the family of subsets T. The synchronization graph shown in Figure 11 is precisely R''.[1]

To summarize, we have shown how from the particular instance (X, T) of set covering, we can construct a synchronization graph G such that from a solution to the latency-constrained resynchronization problem instance defined by G, we can efficiently derive a solution to (X, T). Once this example of the reduction from set covering to latency-constrained resynchronization is understood, it is easily generalized to an arbitrary set covering instance (X', T'). The generalized construction of the initial synchronization graph G is specified by the steps listed in Figure 14.

The main task in establishing our general correspondence between latency-constrained resynchronization and set covering is generalizing Observation 2 to apply to all constructions that follow the steps in Figure 14. This generalization is not conceptually difficult (although it is rather

- Instantiate actors v, w, z, in, out, with execution times 1, 1, 100, 1, and 1, respectively, and instantiate all of the edges in Figure 11 that are contained in the subgraph associated with these five actors.
- For each $t \in T'$, instantiate an actor labeled st that has execution time 40.
- For each $x \in X'$
  - Instantiate an actor labeled sx that has execution time 60.
  - Instantiate the edge $ex \equiv d_0(v, sx)$.
  - Instantiate the edge $d_0(sx, out)$.
- For each $t \in T'$
  - Instantiate the edge $d_0(w, st)$.
  - For each $x \in t$, instantiate the edge $d_0(st, sx)$.
- Set $L_{max} = 103$.

Figure 14. A procedure for constructing an instance of latency-constrained resynchronization from an instance I_{sc} of set covering such that a solution to I_{lr} yields a solution to I_{sc}.

1. In general (for an arbitrary instance of set covering), R'' can be one of multiple possible sets; the number of possibilities for R'' is equal to the number of distinct solutions to the given set covering instance.

tedious) since it is easily verified that all of the arguments in Figure 14 hold for the general construction. Similarly, the reasoning that justifies converting an optimal LCR for the construction into an optimal LCR of the form implied by (10) and (11) extends in a straightforward fashion to the general construction. Furthermore, it is easily verified that this conversion can be executed in $O(|T'|)$ time ($O(|T'|)$ time to convert the initial solution to a form that corresponds to (9) and $O(|T'|)$ time to convert this intermediate solution to a form that corresponds to (10) and (11)). We thus arrive at the following theorem.

Theorem 5: The latency-constrained resynchronization problem is NP-hard for the following classes of synchronization graphs (in decreasing order of generality): general transparent synchronization graphs; transparent, chainable synchronization graphs; transparent synchronization graphs in which each SCC corresponds to a single processor; transparent synchronization graphs in which each SCC corresponds to a single processor, each processor executes only one actor, and all synchronization edges have zero delay.

8. Two-processor systems

In this section, we show that although latency-constrained resynchronization for transparent synchronization graphs is NP-hard, the problem becomes tractable for systems that consist of only two processors — that is, synchronization graphs in which there are two SCCs and each SCC is a fundamental cycle. This reveals a pattern of complexity that is analogous to the classic non-preemptive processor scheduling problem with deterministic execution times, in which the problem is also intractable for general systems, but an efficient greedy algorithm suffices to yield optimal solutions for two-processor systems in which the execution times of all tasks are identical [8, 16]. However, for latency-constrained resynchronization, the tractability for two-processor systems does not depend on any constraints on the task (actor) execution times. Two processor optimality results in multiprocessor scheduling have also been reported in the context of a stochastic model for parallel computation in which tasks have random execution times and communication patterns [29].

In an instance of the **two-processor latency-constrained resynchronization (2LCR)**

problem, we are given a set of *source processor actors* $x_1, x_2, \ldots, x_p$, with associated execution times $\{t(x_i)\}$, such that each x_i is the ith actor scheduled on the processor that corresponds to the source SCC of the synchronization graph; a set of *sink processor actors* $y_1, y_2, \ldots, y_q$, with associated execution times $\{t(y_i)\}$, such that each y_i is the ith actor scheduled on the processor that corresponds to the sink SCC of the synchronization graph; a set of irredundant synchronization edges $S = \{s_1, s_2, \ldots, s_n\}$ such that for each s_i, $src(s_i) \in \{x_1, x_2, \ldots, x_p\}$ and $snk(s_i) \in \{y_1, y_2, \ldots, y_q\}$; and a latency constraint L_{max}, which is a positive integer. A solution to such an instance is a minimal resynchronization R that satisfies $L_{R(G)}(x_1, y_q) \leq L_{max}$. In the remainder of this section, we denote the synchronization graph corresponding to our generic instance of 2LCR by $\tilde{G}$.

As in the previous section, we assume that $\tilde{G}$ is transparent. Also, we start by assuming that $delay(s_i) = 0$ for all s_i, and we refer to the subproblem that results from this restriction as **delayless 2LCR**. In the following two subsections, we show that delayless 2LCR can be solved efficiently in polynomial time, and in Subsection 8.3, we extend this analysis to show that the general 2LCR problem can also be solved efficiently.

8.1 Interval covering

An efficient polynomial time solution to delayless 2LCR can be derived by reducing the problem to a special case of set covering called **interval covering**, in which we are given an ordering $w_1, w_2, \ldots, w_N$ of the members of X (the set that must be covered), such that the collection of subsets T consists entirely of subsets of the form $\{w_a, w_{a+1}, \ldots, w_b\}$, $1 \leq a \leq b \leq N$. Thus, while general set covering involves covering a set from a collection of subsets, interval covering amounts to covering an interval from a collection of subintervals.

Interval covering can be solved in $O(|X||T|)$ time by a simple procedure that first selects the subset $\{w_1, w_2, \ldots, w_{b_1}\}$, where

$$b_1 = max(\{b | (w_1, w_b \in t) \text{ for some } t \in T\});$$

then selects any subset of the form $\{w_{a_2}, w_{a_2+1}, \ldots, w_{b_2}\}$, $a_2 \leq b_1 + 1$, where

$$b_2 = max(\{b | (w_{b_1+1}, w_b \in t) \text{ for some } t \in T\});$$

then selects any subset of the form $\{w_{a_3}, w_{a_3+1}, \ldots, w_{b_3}\}$, $a_3 \leq b_2 + 1$, where

$$b_3 = max(\{b | (w_{b_2+1}, w_b \in t) \text{ for some } t \in T\});$$

and so on until $b_n = N$. It is easily verified that the collection of selected subsets will be a minimal cover for X.

8.2 Two-processor latency-constrained resynchronization

To reduce delayless 2LCR to interval covering, we start with the following observations.

Observation 4: Suppose that R is a resynchronization of $\tilde{G}$, $r \in R$, and r contributes to the elimination of synchronization edge s. Then r subsumes s. Thus, the set of synchronization edges that r contributes to the elimination of is simply the set of synchronization edges that are subsumed by r.

Proof: This follows immediately from the restriction that there can be no resynchronization edges directed from a y_j to an x_i (feedforward resynchronization), and thus in $R(\tilde{G})$, there can be at most one synchronization edge in any path directed from $src(s)$ to $snk(s)$. ■

Observation 5: If R is a resynchronization of $\tilde{G}$, then

$$L_{R(\tilde{G})}(x_1, y_q) = max(\{t_{pred}(src(s')) + t_{succ}(snk(s')) \,|\, s' \in R\}), \text{ where}$$

$$t_{pred}(x_i) \equiv \sum_{j \leq i} t(x_j) \text{ for } i = 1, 2, \ldots, p, \text{ and } t_{succ}(y_i) \equiv \sum_{j \geq i} t(y_j) \text{ for } i = 1, 2, \ldots, q.$$

Proof: Given a synchronization edge $(x_a, y_b) \in R$, there is exactly one delayless path in $R(\tilde{G})$ from x_1 to y_q that contains (x_a, y_b) and the set of vertices traversed by this path is $\{x_1, x_2, \ldots, x_a, y_b, y_{b+1}, \ldots, y_q\}$. The desired result follows immediately. ■

Now, corresponding to each of the source processor actors x_i that satisfies $t_{pred}(x_i) + t(y_q) \leq L_{max}$ we define an ordered pair of actors (a "resynchronization candidate") by

$$v_i \equiv (x_i, y_j), \text{ where } j = min(\{k | (t_{pred}(x_i) + t_{succ}(y_k) \leq L_{max})\}). \quad (13)$$

Consider the example shown in Figure 15. Here, we assume that $t(z) = 1$ for each actor z, and $L_{max} = 10$. From (13), we have

$$v_1 = (x_1, y_1), v_2 = (x_2, y_1), v_3 = (x_3, y_2), v_4 = (x_4, y_3),$$
$$v_5 = (x_5, y_4), v_6 = (x_6, y_5), v_7 = (x_7, y_6), v_8 = (x_8, y_7). \quad (14)$$

If v_i exists for a given x_i, then $d_0(v_i)$ can be viewed as the "best" resynchronization edge that has x_i as the source actor, and thus, to construct an optimal LCR, we can select the set of

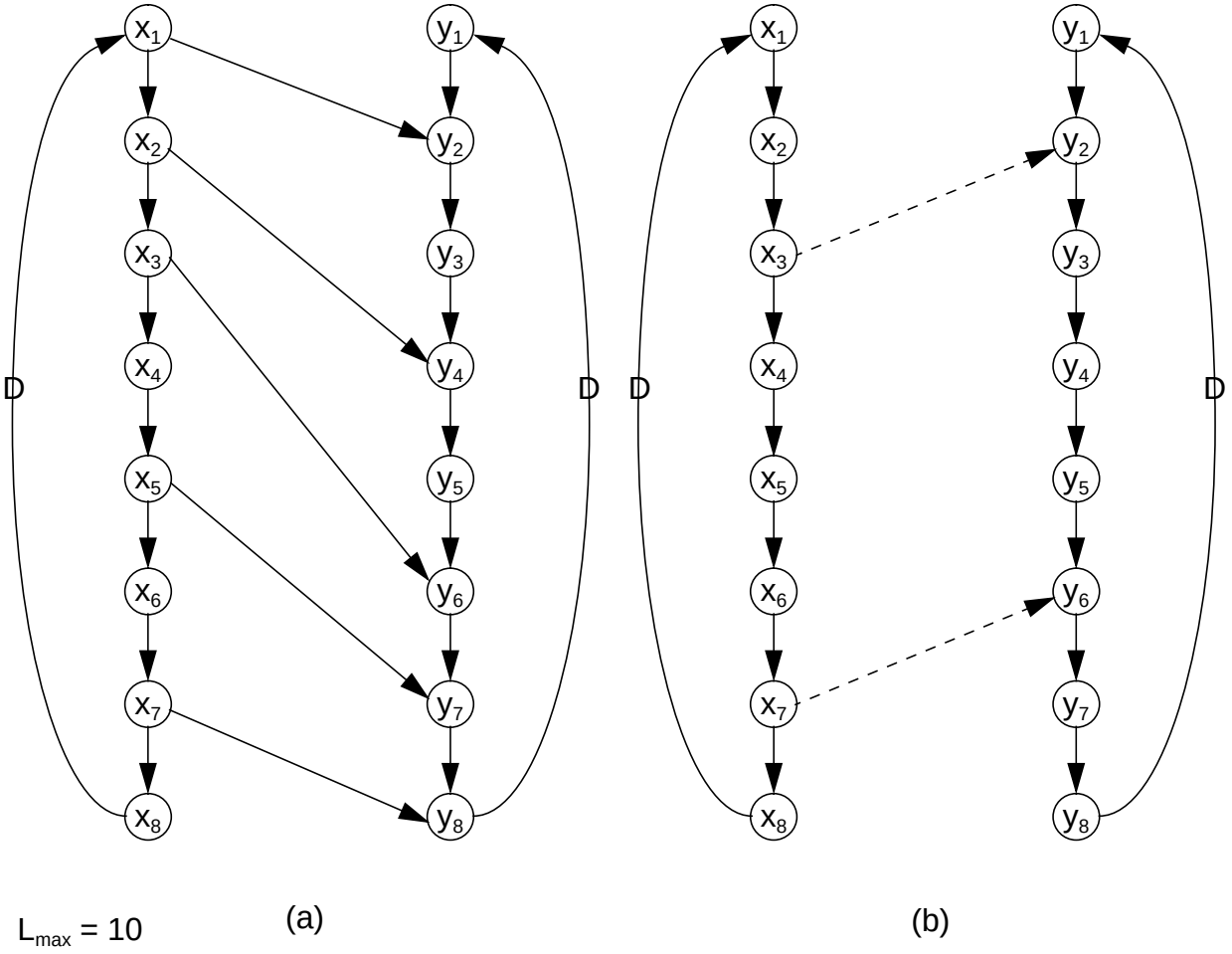


Figure 15. An instance of delayless, two-processor latency-constrained resynchronization. In this example, the execution times of all actors are identically equal to unity.

resynchronization edges entirely from among the v_is. This is established by the following two observations.

Observation 6: Suppose that R is an LCR of G, and suppose that (x_a, y_b) is a synchronization edge in R such that $(x_a, y_b) \neq v_a$. Then $(R - \{(x_a, y_b)\} + \{d_0(v_a)\})$ is an LCR of R.

Proof: Let $v_a = (x_a, y_c)$ and $R' = (R - \{(x_a, y_b)\} + \{d_0(v_a)\})$, and observe that v_a exists, since

$$((x_a, y_b) \in R) \Rightarrow (t_{pred}(x_a) + t_{succ}(y_b) \leq L_{max}) \Rightarrow (t_{pred}(x_a) + t(y_q) \leq L_{max}).$$

From Observation 4, the set of synchronization edges that (x_a, y_b) contributes to the elimination of is simply the set of synchronization edges that are subsumed by (x_a, y_b). Now, if s is a synchronization edge that is subsumed by (x_a, y_b), then

$$\rho_{\tilde{G}}(src(s), x_a) + \rho_{\tilde{G}}(y_b, snk(s)) \leq delay(s). \tag{15}$$

From the definition of v_a, we have that $c \leq b$, and thus, that $\rho_{\tilde{G}}(y_c, y_b) = 0$. It follows from (15) that

$$\rho_{\tilde{G}}(src(s), x_a) + \rho_{\tilde{G}}(y_c, snk(s)) \leq delay(s), \tag{16}$$

and thus, that v_a subsumes s. Hence, v_a subsumes all synchronization edges that (x_a, y_b) contributes to the elimination of, and we can conclude that R' is a valid resynchronization of $\tilde{G}$.

From the definition of v_a, we know that $t_{pred}(x_a) + t_{succ}(y_c) \leq L_{max}$, and thus since R is an LCR, we have from Observation 5 that R' is an LCR. ■

From Fact 3 and the assumption that the members of S are all delayless, an optimal LCR of $\tilde{G}$ consists only of delayless synchronization edges. Thus from Observation 6, we know that there exists an optimal LCR that consists only of members of the form $d_0(v_i)$ (but not necessarily of all $d_0(v_i)$s). Furthermore, from Observation 5, we know that a collection V of v_is is an LCR if

and only if $\bigcup_{v \in V} \chi(v) = \{s_1, s_2, \ldots, s_n\}$, where $\chi(v)$ is the set of synchronization edges that are subsumed by v. The following observation completes the correspondence between 2LCR and interval covering.

Observation 7: Let $s_1', s_2', \ldots, s_n'$ be the ordering of $s_1, s_2, \ldots, s_n$ specified by

$$(x_a = src(s_i'), x_b = src(s_j'), a < b) \Rightarrow (i < j). \tag{17}$$

That is the s_i''s are ordered according to the order in which their respective source actors execute on the source processor. Suppose that for some $j \in \{1, 2, \ldots, p\}$, some $m > 1$, and some $i \in \{1, 2, \ldots, n - m\}$, we have $s_i' \in \chi(v_j)$ and $s_{i+m}' \in \chi(v_j)$. Then $s_{i+1}', s_{i+2}', \ldots, s_{i+m-1}' \in \chi(v_j)$.

In Figure 15(a), the ordering specified by (17) is

$$s_1' = (x_1, y_2), s_2' = (x_2, y_4), s_3' = (x_3, y_6), s_4' = (x_5, y_7), s_5' = (x_7, y_8), \tag{18}$$

and thus from (14), we have

$$\chi(v_1) = \{s_1'\}, \chi(v_2) = \{s_1', s_2'\}, \chi(v_3) = \{s_1', s_2', s_3'\}, \chi(v_4) = \{s_2', s_3'\}$$
$$\chi(v_5) = \{s_2', s_3', s_4'\}, \chi(v_6) = \{s_3', s_4'\}, \chi(v_7) = \{s_3', s_4', s_5'\}, \chi(v_8) = \{s_4', s_5'\}, \tag{19}$$

which is clearly consistent with Observation 7.

Proof of Observation 7: Let $v_j = (x_j, y_l)$, and suppose k is a positive integer such that $i < k < i + m$. Then from (17), we know that $\rho_{\tilde{G}}(src(s_k'), src(s_{i+m}')) = 0$. Thus, since $s_{i+m}' \in \chi(v_j)$, we have that

$$\rho_{\tilde{G}}(src(s_k'), x_j) = 0. \tag{20}$$

Now clearly

$$\rho_{\tilde{G}}(snk(s_i'), snk(s_k')) = 0, \tag{21}$$

since otherwise $\rho_{\tilde{G}}(snk(s_k'), snk(s_i')) = 0$ and thus (from 17) s_k' subsumes s_i', which contradicts the assumption that the members of S are irredundant. Finally, since $s_i' \in \chi(v_j)$, we know that $\rho_{\tilde{G}}(y_l, snk(s_i')) = 0$. Combining this with (21) yields

$$\rho_{\tilde{G}}(y_l, snk(s_k')) = 0, \tag{22}$$

and (20) and (22) together yield that $s_k' \in \chi(v_j)$. ■

From Observation 7 and the preceding discussion, we conclude that an optimal LCR of $\tilde{G}$ can obtained by the following steps.

(a) Construct the ordering $s_1', s_2', \ldots, s_n'$ specified by (17).

(b) For $i = 1, 2, \ldots, p$, determine whether or not v_i exists, and if it exists, compute v_i.

(c) Compute $\chi(v_j)$ for each value of j such that v_j exists.

(d) Find a minimal cover C for S given the family of subsets $\{\chi(v_j) | v_j \text{ exists}\}$.

(e) Define the resynchronization $R = \{v_j | \chi(v_j) \in C\}$.

Steps (a), (b), and (e) can clearly be performed in $O(N)$ time, where N is the number of vertices in $\tilde{G}$. If the algorithm outlined in Section 8.1 is employed for step (d), then from the discussion in Section 8.1 and Observation 8(e) in Section 8.3, it can be easily verified that the time complexity of step (d) is $O(N^2)$. Step (c) can also be performed in $O(N^2)$ time using the observation that if $v_i = (x_i, y_j)$, then $\chi(v_i) \equiv \{(x_a, y_b) \in S | a \le i \textbf{ and } b \ge j\}$, where $S = \{s_1, s_2, \ldots, s_n\}$ is the set of synchronization edges in $\tilde{G}$. Thus, we have the following result.

Theorem 6: Polynomial-time solutions (quadratic in the number of synchronization graph ver-

tices) exist for the delayless, two-processor latency-constrained resynchronization problem.

Note that solutions more efficient than the $O(N^2)$ approach described above may exist.

From (19), we see that there are two possible solutions that can result if we apply Steps (a)-(e) to Figure 15(a) and use the technique described in Subsection 8.1 for interval covering. These solutions correspond to the interval covers $C_1 = \{\chi(v_3), \chi(v_7)\}$ and $C_2 = \{\chi(v_3), \chi(v_8)\}$. The synchronization graph that results from the interval cover C_1 is shown in Figure 15(b).

8.3 Taking delays into account

If delays exist on one or more edges of the original synchronization graph, then the correspondence defined in the previous subsection between 2LCR and interval covering does not necessarily hold. For example, consider the synchronization graph in Figure 16. Here, the numbers beside the actors specify execution times; a "D" on top of an edge specifies a unit delay; the latency input and latency output are respectively x_1 and y_q; and the latency constraint is $L_{max} = 12$. It is easily verified that v_i exists for $i = 1, 2, \ldots, 6$, and from (13), we obtain

$$v_1 = (x_1, y_3), v_2 = (x_2, y_4), v_3 = (x_3, y_6), v_4 = (x_4, y_8), v_5 = (x_5, y_8), v_6 = (x_6, y_8). \quad (23)$$

Now if we order the synchronization edges as specified by (17), then

$$s_i' = (x_i, y_{i+4}) \text{ for } i = 1, 2, 3, 4, \text{ and } s_i' = (x_i, y_{i-4}) \text{ for } i = 5, 6, 7, 8, \quad (24)$$

and if the correspondence between delayless 2LCR and interval covering defined in the previous section were to hold for general 2LCR, then we would have that

$$\text{each subset } \chi(v_i) \text{ is of the form } \{s_a', s_{a+1}', \ldots, s_b'\}, 1 \le a \le b \le 8. \quad (25)$$

However, computing the subsets $\chi(v_i)$, we obtain

$$\chi(v_1) = \{s_1', s_7', s_8'\}, \chi(v_2) = \{s_1', s_2', s_8'\}, \chi(v_3) = \{s_2', s_3'\}$$

$$\chi(v_4) = \{s_4'\}, \chi(v_5) = \{s_4', s_5'\}, \chi(v_6) = \{s_4', s_5', s_6'\}, \tag{26}$$

and these subsets are clearly not all consistent with the form specified in (25). Thus, the algorithm developed in Subsection 8.2 does not apply directly to handle delays.

However, the technique developed in the previous section can be extended to solve the general 2LCR problem in polynomial time. This extension is based on separating the subsumption relationships between the v_i's and the synchronization edges into two categories: if $v_i = (x_i, y_j)$ subsumes the synchronization edge $s = (x_k, y_l)$ then we say that v_i **1-subsumes** s if $i < k$, and we say that v_i **2-subsumes** s if $i \geq k$. For example in Figure 16(a), $v_1 = (x_1, y_3)$ 1-subsumes

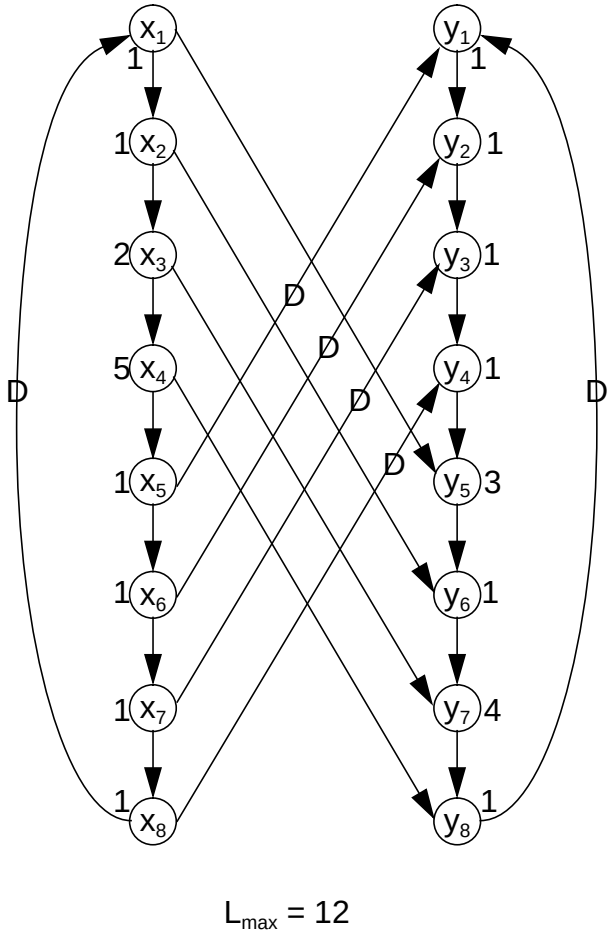


Figure 16. A synchronization graph with unit delays on some of the synchronization edges.

both (x_7, y_3) and (x_8, y_4), and $v_5 = (x_5, y_8)$ 2-subsumes (x_4, y_8) and (x_5, y_1).

Observation 8: Assuming the same notation for a generic instance of 2LRC that was defined in the previous subsection, the initial synchronization graph $\tilde{G}$ satisfies the following conditions:

(a) Each synchronization edge has at most one unit of delay ($delay(s_i) \in \{0, 1\}$).

(b) If (x_i, y_j) is a zero-delay synchronization edge and (x_k, y_l) is a unit-delay synchronization edge, then $i < k$ and $j > l$.

(c) If v_i 1-subsumes a unit-delay synchronization edge (x_i, y_j), then v_i also 1-subsumes all unit-delay synchronization edges s that satisfy $src(s) = x_{i+n}, n > 0$.

(d) If v_i 2-subsumes a unit-delay synchronization edge (x_i, y_j), then v_i also 2-subsumes all unit-delay synchronization edges s that satisfy $src(s) = x_{i-n}, n > 0$.

(e) If (x_i, y_j) and (x_k, y_l) are both distinct zero-delay synchronization edges or they are both distinct unit-delay synchronization edges, then $i \neq k$ and $(i < k) \Leftrightarrow (j < l)$.

(f) If (x_i, y_j) 1-subsumes a unit delay synchronization edge (x_k, y_l), then $l \geq j$.

Proof outline: From Fact 5, we know that $\rho(x_1, y_q) = 0$. Thus, there exists at least one delayless synchronization edge in $\tilde{G}$. Let e be one such delayless synchronization edge. Then it is easily verified from the structure of $\tilde{G}$ that for all x_i, y_j, there exists a path $p_{i,j}$ in $\tilde{G}$ directed from x_i to y_j such that $p_{i,j}$ contains e, $p_{i,j}$ contains no other synchronization edges, and $Delay(p_{i,j}) \leq 2$. It follows that any synchronization edge e' whose delay exceeds unity would be redundant in $\tilde{G}$. Thus, part (a) follows from the assumption that the synchronization edges in $\tilde{G}$ are all irredundant.

The other parts can be verified easily from the structure of $\tilde{G}$, including the assumption that all synchronization edges in $\tilde{G}$ are irredundant. We omit the details. ■

Resynchronizations for instances of general 2LCR can be partitioned into two categories — **category A** consists of all resynchronizations that contain at least one synchronization edge having nonzero delay, and **category B** consists of all resynchronizations that consist entirely of delayless synchronization edges. An optimal category A solution (a category A solution whose cost is less than or equal to the cost of all category A solutions) can be derived by simply applying

the optimal solution described in Subsection 8.2 to "rearrange" the delayless resynchronization edges, and then replacing all synchronization edges that have nonzero delay with a single unit delay synchronization edge directed from x_p, the last actor scheduled on the source processor to y_1, the first actor scheduled on the sink processor. We refer to this approach as **Algorithm A**.

An example is shown in Figure 17. Figure 17(a) shows an example where for general 2LCR, the constraint that all synchronization edges have zero delay is too restrictive to permit a globally optimal solution. Here, the latency constraint is assumed to be $L_{max} = 2$. Under this constraint, it is easily seen that no zero-delay resynchronization edges can be added without violating the latency constraint. However, if we allow resynchronization edges that have delay, then we can apply Algorithm A to achieve a cost of two synchronization edges. The resulting synchronization graph, with redundant synchronization edges removed, is shown in Figure 17(b). Observe that this resynchronization is an LCR since only delayless synchronization edges affect the latency of a transparent synchronization graph.

Now suppose that $\tilde{G}$ (our generic instance of 2LCR) contains at least one unit-delay synchronization edge, suppose that G_b is an optimal category B solution for $\tilde{G}$, and let R_b denote

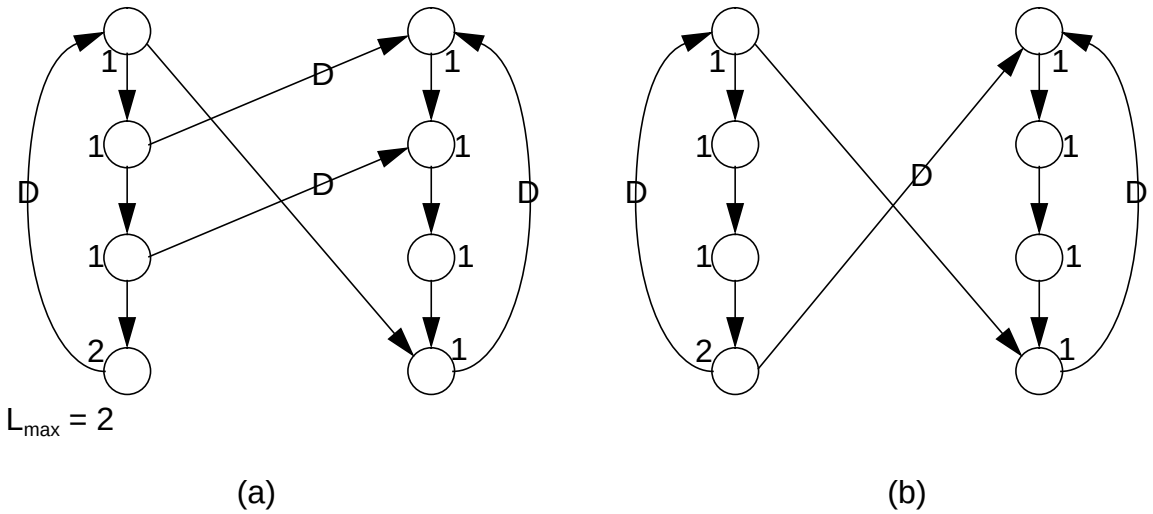


Figure 17. An example in which constraining all resynchronization edges to be delayless precludes the ability to derive an optimal resynchronization.

the set of resynchronization edges in G_b. Let $ud(\tilde{G})$ denote the set of synchronization edges in $\tilde{G}$ that have unit delay, and let $(x_{k_1}, y_{l_1}), (x_{k_2}, y_{l_2}), \ldots, (x_{k_M}, y_{l_M})$ denote the ordering of the members of $ud(\tilde{G})$ that corresponds to the order in which the source actors execute on the source processor — that is, $(i < j) \Rightarrow (k_i < k_j)$. Note from Observation 8(a) that $ud(\tilde{G})$ is the set of all synchronization edges in $\tilde{G}$ that are not delayless. Also, let $1subs(\tilde{G}, G_b)$ denote the set of unit-delay synchronization edges in $\tilde{G}$ that are 1-subsumed by resynchronization edges in G_b. That is,

$$1subs(\tilde{G}, G_b) \equiv \{s \in ud(\tilde{G}) \,|\, (\exists((z_1, z_2) \in R_b)) \text{ s.t } ((z_1, z_2) \text{ 1-subsumes } s \text{ in} \tilde{G})\}.$$

If $1subs(\tilde{G}, G_b)$ is not empty, define

$$r = min(\{j \,|\, (x_{k_j}, y_{l_j}) \in 1subs(\tilde{G}, G_b)\}). \tag{27}$$

Suppose $(x_m, y_{m'}) \in 1subs(\tilde{G}, G_b)$. Then by definition of r, $m' \geq l_r$, and thus $\rho_{\tilde{G}}(y_{l_r}, y_{m'}) = 0$. Furthermore, since x_m and x_1 execute on the same processor, $\rho_{\tilde{G}}(x_m, x_1) \leq 1$. Hence $\rho_{\tilde{G}}(x_m, x_1) + \rho_{\tilde{G}}(y_{l_r}, y_{m'}) \leq 1 = delay(x_m, y_{m'})$, so we have that (x_1, y_{l_r}) subsumes $(x_m, y_{m'})$ in $\tilde{G}$. Since $(x_m, y_{m'})$ is an arbitrary member of $ud(\tilde{G})$, we conclude that

$$\text{Every member of } 1subs(\tilde{G}, G_b) \text{ is subsumed by } (x_1, y_{l_r}). \tag{28}$$

Now, if $\Gamma \equiv (ud(\tilde{G}) - 1subs(\tilde{G}, G_b))$ is not empty, then define

$$u = max(\{j \,|\, (x_{k_j}, y_{l_j}) \in \Gamma\}), \tag{29}$$

and suppose $(x_m, y_{m'}) \in \Gamma$. By definition of u, $m \leq k_u$ and thus $\rho_{\tilde{G}}(x_m, x_{k_u}) = 0$. Furthermore, since $y_{m'}$ and y_q execute on the same processor, $\rho_{\tilde{G}}(y_q, y_{m'}) \leq 1$. Hence,

$$\rho_{\tilde{G}}(x_m, x_{k_u}) + \rho_{\tilde{G}}(y_q, y_{m'}) \leq 1 = delay(x_m, y_{m'}),$$

and we have that

$$\text{Every member of } \Gamma \text{ is subsumed by } (x_{k_u}, y_q). \tag{30}$$

Observe also that from the definitions of r and u, and from Observation 8(c),

$$((1subs(\tilde{G}, G_b) \neq \varnothing) \textbf{ and } (\Gamma \neq \varnothing)) \Rightarrow (u = r - 1); \tag{31}$$

$$(1subs(\tilde{G}, G_b) = \varnothing) \Rightarrow (u = M); \tag{32}$$

and

$$(\Gamma = \varnothing) \Rightarrow (r = 1). \tag{33}$$

Now we define the synchronization graph $Z(\tilde{G})$ by $Z(\tilde{G}) = (V, (E - ud(\tilde{G})) + P)$, where V and E are the sets of vertices and edges in $\tilde{G}$; $P = \{d_0(x_1, y_{l_r}), d_0(x_{k_u}, y_q)\}$, if both $1subs(\tilde{G}, G_b)$ and Γ are non-empty; $P = \{d_0(x_1, y_{l_r})\}$ if Γ is empty; and $P = \{d_0(x_{k_u}, y_q)\}$ if $1subs(\tilde{G}, G_b)$ is empty.

Theorem 7: G_b is a resynchronization of $Z(\tilde{G})$.

Proof: The set of synchronization edges in $Z(\tilde{G})$ is $E_0 + P$, where E_0 is the set of delayless synchronization edges in $\tilde{G}$. Since G_b is a resynchronization of $\tilde{G}$, it suffices to show that for each $e \in P$,

$$\rho_{G_b}(src(e), snk(e)) = 0. \tag{34}$$

If $1subs(\tilde{G}, G_b)$ is non-empty then from (27) (the definition of r) and Observation 8(f), there must be a delayless synchronization edge e' in G_b such that $snk(e') = y_w$ for some $w \leq l_r$. Thus,

$$\rho_{G_b}(x_1, y_{l_r}) \leq \rho_{G_b}(x_1, src(e')) + \rho_{G_b}(snk(e'), y_{l_r}) = 0 + \rho_{G_b}(y_w, y_{l_r}) = 0,$$

and we have that (34) is satisfied for $e = (x_1, y_{l_r})$.

Similarly if Γ is non-empty, then from (29) (the definition of u) and from the definition of *2-subsumes*, there exists a delayless synchronization edge e' in G_b such that $src(e') = x_w$ for some $w \geq k_u$. Thus,

$$\rho_{G_b}(x_{k_u}, y_q) \leq \rho_{G_b}(x_{k_u}, src(e')) + \rho_{G_b}(snk(e'), y_q) = \rho_{G_b}(x_{k_u}, x_w) + 0 = 0;$$

hence, we have that (34) is satisfied for $e = (x_{k_u}, y_q)$.

From the definition of P, it follows that (34) is satisfied for every $e \in P$. ■

Corollary 1: The latency of $Z(\tilde{G})$ is no greater than L_{max}. That is, $L_{Z(\tilde{G})}(x_1, y_q) \leq L_{max}$.

Proof: From Theorem 7, we know that G_b preserves $Z(\tilde{G})$. Thus, from Lemma 1, it follows that $L_{Z(\tilde{G})}(x_1, y_q) \leq L_{G_b}(x_1, y_q)$. Furthermore, from the assumption that G_b is an optimal category B LCR, we have $L_{G_b}(x_1, y_q) \leq L_{max}$. We conclude that $L_{Z(\tilde{G})}(x_1, y_q) \leq L_{max}$. ■

Theorem 7, along with (31)-(33), tells us that an optimal category B LCR of $\tilde{G}$ is always a resynchronization of

(1) a synchronization graph of the form

$$(V, ((E - ud(\tilde{G})) + \{d_0(x_1, y_{l_\alpha}), d_0(x_{k_{\alpha-1}}, y_q)\})), \ 1 < \alpha \leq M, \tag{35}$$

or

(2) of the graph $(V, ((E - ud(\tilde{G})) + \{d_0(x_1, y_{l_1})\}))$, (36)

or

(3) of the graph $(V, ((E - ud(\tilde{G})) + \{d_0(x_{k_M}, y_q)\}))$. (37)

Thus, from Corollary 1, an optimal resynchronization can be computed by examining each of the $(M + 1) = (|ud(\tilde{G})| + 1)$ synchronization graphs defined by (35)-(37), computing an optimal LCR for each of these graphs whose latency is no greater than L_{max}, and returning one of the optimal LCRs that has the fewest number of synchronization edges. This is straightforward since these graphs contain only delayless synchronization edges, and thus the algorithm of Section 8.2 can be used.

Recall the example of Figure 16(a). Here,

$$ud(\tilde{G}) = \{(x_5, y_1), (x_6, y_2), (x_7, y_3), (x_8, y_4)\},$$

and the set of synchronization graphs that correspond to (35)-(37) are shown in Figure 18(a)-(e). The latencies of the graphs in Figure 18(a)-(e) are respectively 14, 13, 12, 13, and 14. Since $L_{max} = 12$, we only need to compute an optimal LCR for the graph of Figure 18(c) (from Corollary 1). This is done by first removing redundant edges from the graph (yielding the graph in Fig-

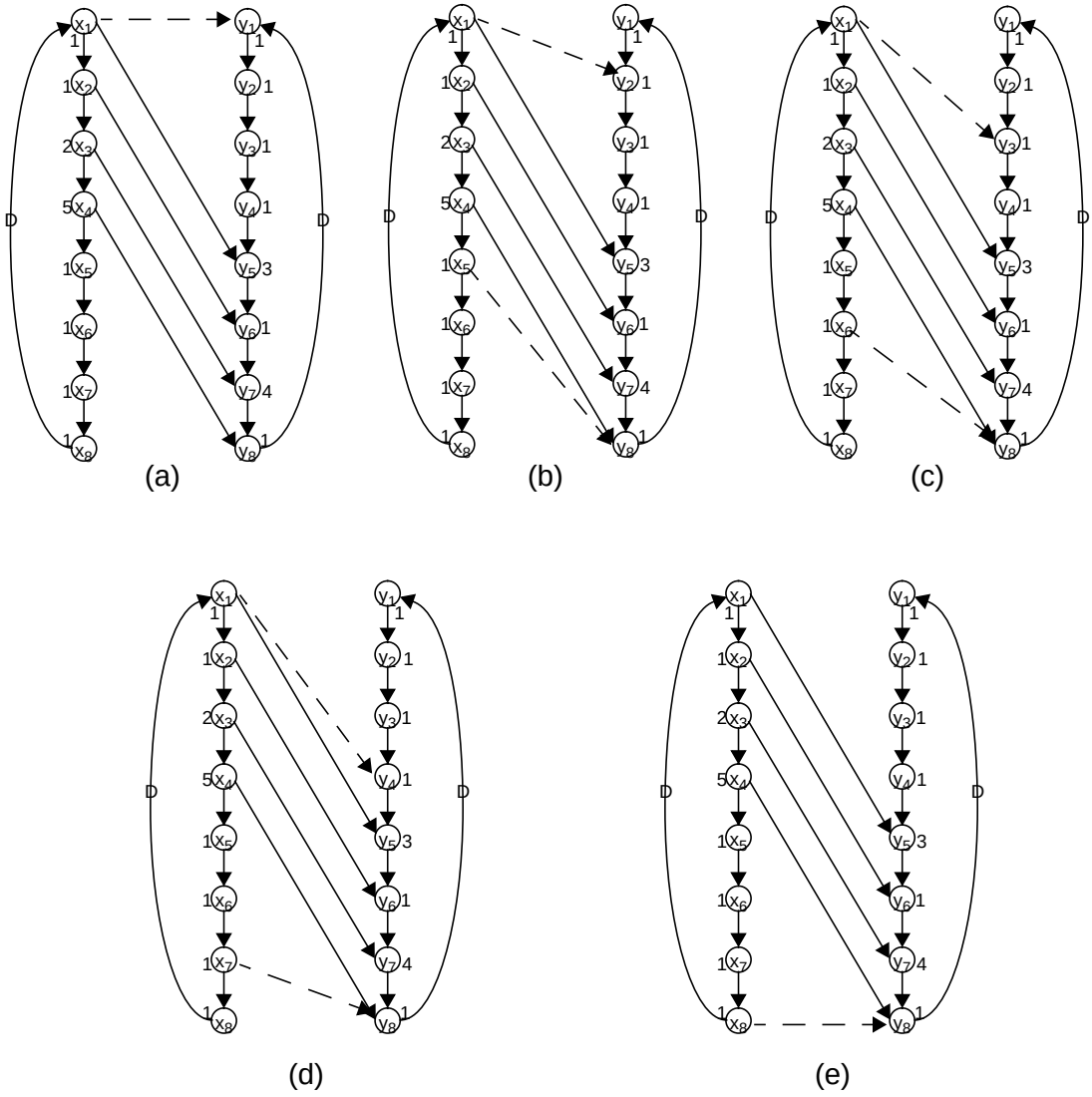


Figure 18. The synchronization graphs considered in Algorithm B for the example in Figure 16.

ure 19(b)) and then applying the algorithm developed in Section 8.2. For the synchronization graph of Figure 19(b), and $L_{max} = 12$, it is easily verified that the set of v_is is

$$v_1 = (x_1, y_3), v_2 = (x_2, y_4), v_3 = (x_3, y_6), v_4 = (x_4, y_8), v_5 = (x_5, y_8), v_6 = (x_6, y_8).$$

If we let

$$s_1 = (x_1, y_3), s_2 = (x_2, y_6), s_3 = (x_3, y_7), s_4 = (x_6, y_8), \tag{38}$$

then we have,

$$\chi(v_1) = \{s_1\}, \chi(v_2) = \{s_2\}, \chi(v_3) = \{s_2, s_3\}, \chi(v_4) = \chi(v_5) = \varnothing, \chi(v_6) = \{s_4\}. \tag{39}$$

From (39), the algorithm outlined in Subsection 8.1 for interval covering can be applied to obtain an optimal resynchronization. This results in the resynchronization $R = \{v_1, v_3, v_6\}$. The resulting synchronization graph is shown in Figure 19(c). Observe that the number of synchronization edges has been reduced from 8 to 3, while the latency has increased from 10 to $L_{max} = 12$. Also, none of the original synchronization edges in $\tilde{G}$ are retained in the resynchronization.

We say that **Algorithm B** for general 2LCR is the approach of constructing the

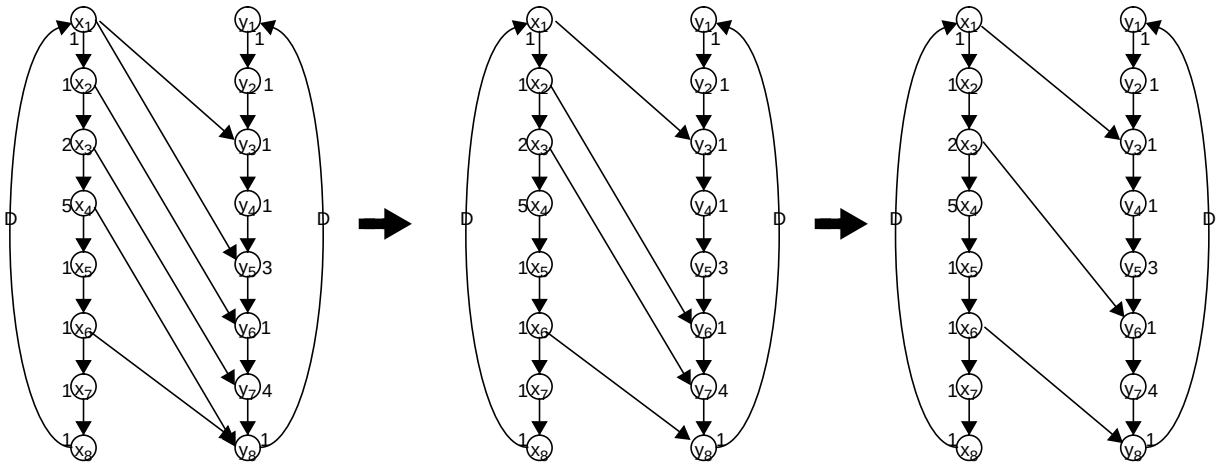


Figure 19. Derivation of an optimal LCR for the synchronization graph of Figure 18(c).

($|ud(\tilde{G})| + 1$**)** synchronization graphs corresponding to (35)-(37), computing an optimal LCR for each of these graphs whose latency is no greater than L_{max}, and returning one of the optimal LCRs that has the fewest number of synchronization edges. We have shown above that Algorithm B leads to an optimal LCR *under the constraint that all resynchronization edges have zero delay.*

Thus, given an instance of a general 2LCR, a globally optimal solution can be derived by applying Algorithm A and Algorithm B and retaining the best of the resulting two solutions. The time complexity of this two phased approach is dominated by the complexity of Algorithm B, which is $O(|ud(\tilde{G})|N^2)$ (a factor of $|ud(\tilde{G})|$ greater than the complexity of the technique for delayless 2LCR that was developed in Section 8.2), where N is the number of vertices in $\tilde{G}$. Since $|ud(\tilde{G})| \leq N$ from Observation 8(e), the complexity is $O(N^3)$.

Theorem 8: Polynomial-time solutions exist for the general two-processor latency-constrained resynchronization problem.

The example in Figure 17 shows how it is possible for Algorithm A to produce a better result than Algorithm B. Conversely, the ability of Algorithm B to outperform Algorithm A can be demonstrated through the example of Figure 16. From Figure 19(c), we know that the result computed by Algorithm B has a cost of 3 synchronization edges. The result computed by Algorithm A can be derived by applying interval covering to the subsets specified in (26) with all of the unit-delay edges (s_5', s_6', s_7', s_8') removed:

$$\chi(v_1) = \{s_1'\}, \chi(v_2) = \{s_1', s_2'\}, \chi(v_3) = \{s_2', s_3'\}$$
$$\chi(v_4) = \chi(v_5) = \chi(v_6) = \{s_4'\}. \quad (40)$$

A minimal cover for (40) is achieved by $\{\chi(v_2), \chi(v_3), \chi(v_4)\}$, and the corresponding synchronization graph computed by Algorithm A is shown in Figure 20. This solution has a cost of 4 synchronization edges, which is one greater than that of the result computed by Algorithm B for this example.

9. A heuristic for latency-constrained resynchronization

In this section, we present the application of a well-known set covering heuristic to the latency-constrained resynchronization problem, and we illustrate the ability of this method to systematically trade off latency for synchronization overhead. Our heuristic for latency-constrained resynchronization is based on the simple greedy approximation algorithm for set covering that repeatedly selects a subset that covers the largest number of *remaining elements*, where a remaining element is an element that is not contained in any of the subsets that have already been selected. In [18, 20] it is shown that this set covering technique is guaranteed to compute a solu-

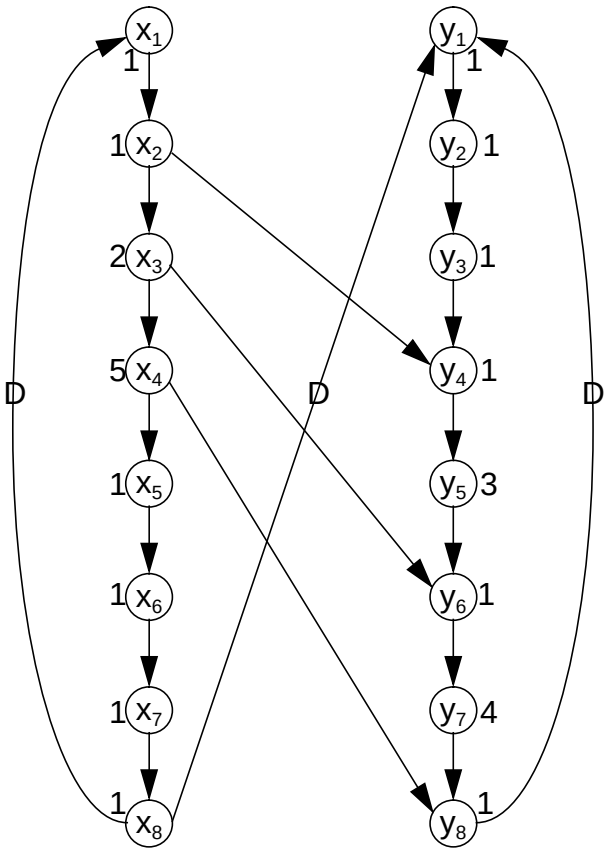


Figure 20. The solution derived by Algorithm A when it is applied to the example of Figure 16.

tion whose cardinality is no greater than $(\ln(|X|) + 1)$ times that of the optimal solution, where X is the set that is to be covered.

To adapt this set covering technique to latency-constrained resynchronization, we construct an instance of set covering by choosing the set X, the set of elements to be covered, to be the set of synchronization edges, and choosing the family of subsets to be

$$T \equiv \{\chi(v_1, v_2) \mid ((v_1, v_2 \in V) \textbf{ and } (\rho_G(v_2, v_1) = \infty) \textbf{ and } (L'(v_1, v_2) \leq L_{max}))\}, \qquad (41)$$

where $G = (V, E)$ is the input synchronization graph, and L' is the latency of the synchronization graph $(V, \{E + \{(v_1, v_2)\}\})$ that results from adding the resynchronization edge (v_1, v_2) to G. Assuming that $T_{\mathit{fi}(G)}(x, y)$ has been determined for all $x, y \in V$, L' can easily be computed from

$$L'(v_1, v_2) = max(\{(T_{\mathit{fi}(G)}(\upsilon, v_1) + T_{\mathit{fi}(G)}(v_2, o_L)), L_G\}) \ , \qquad (42)$$

where υ is the source actor in $\mathit{fi}(G)$, o_L is the latency output, and L_G is the latency of G.

The middle constraint, $\rho_G(v_2, v_1) = \infty$, in (41) ensures that inserting the resynchronization edge (v_2, v_1) does not introduce a cycle, and thus that it does not reduce the estimated throughput.

Our heuristic for latency-constrained resynchronization determines the family of subsets specified by (41), chooses a member of this family that has maximum cardinality, inserts the corresponding delayless resynchronization edge, removes all synchronization edges that it subsumes, and updates the values $T_{\mathit{fi}(G)}(x, y)$ and $\rho_G(x, y)$ for the new synchronization graph that results. This process is then repeated on the new synchronization graph, and it continues until it arrives at a synchronization graph for which the computation defined by (41) produces the empty set — that is, the algorithm terminates when no more resynchronization edges can be added without increasing the latency beyond L_{max}. Figure 21 gives a pseudocode specification of this algorithm (with some straightforward modifications to improve the running time). Fact 1 guarantees that the result returned by *latency-constrained-resynchronization* is always a valid resynchronization.

Clearly, each time a delayless resynchronization edge is added to a synchronization graph, the *connectivity* of the graph is increased by at least one, where the connectivity is defined to be

function latency-constrained-resynchronization
input: a synchronization graph $G = (V, E)$
output: an alternative synchronization graph that preserves G.

compute $\rho_G(x, y)$ for all actor pairs $x, y \in V$
compute $T_{fi(G)}(x, y)$ for all actor pairs $x, y \in (V \cup \{\upsilon\})$
$complete$ = FALSE
while not $(complete)$
    $best = NULL$, $M = 0$
    for $x, y \in V$
        if $((\rho_G(y, x) = \infty)$ **and** $(L'(x, y) \leq L_{max}))$
            $\chi^* = \chi((x, y))$
            if $(|\chi^*| > M)$
                $M = |\chi^*|$
                $best = (x, y)$
            end if
        end if
    end for
    if $(best = NULL)$
        $complete = TRUE$
    else
        $E = E - \chi(best) + \{d_0(best)\}$
        $G = (V, E)$
        for $x, y \in (V \cup \{\upsilon\})$ /* update $T_{fi(G)}$ */
            $T_{new}(x, y) = max(\{T_{fi(G)}(x, y), T_{fi(G)}(x, src(best)) + T_{fi(G)}(snk(best), y)\})$
        end for
        for $x, y \in V$ /* update ρ_G */
            $\rho_{new}(x, y) = min(\{\rho_G(x, y), \rho_G(x, src(best)) + \rho_G(snk(best), y)\})$
        end for
        $\rho_G = \rho_{new}$, $T_{fi(G)} = T_{new}$
    end if
end while
return G
end function

Figure 21. A heuristic for latency-constrained resynchronization.

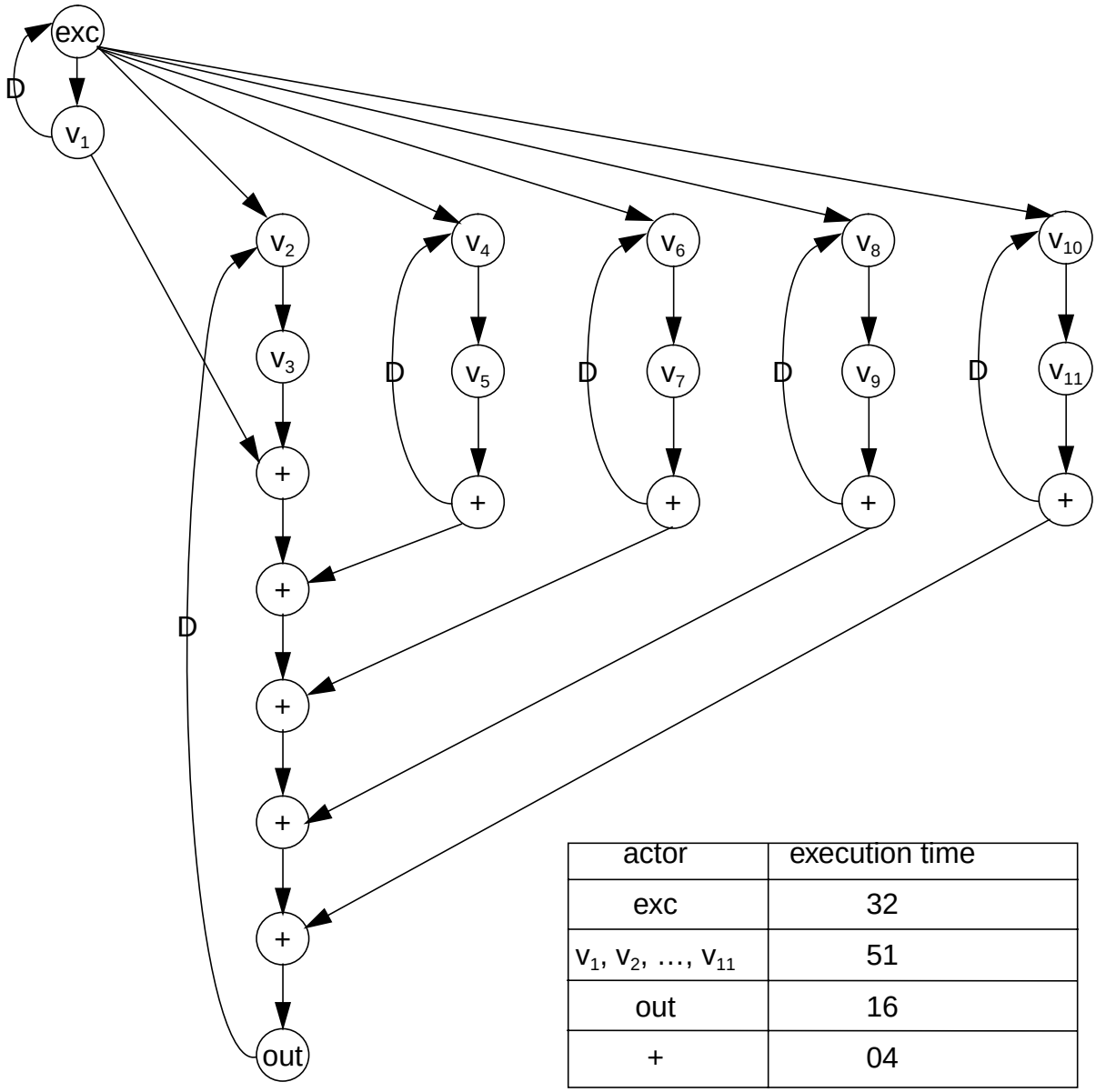
number of ordered vertex pairs (x, y) that satisfy $\rho_G(x, y) = 0$. Thus, the number of iterations of the while loop in Figure 21 is bounded above by $|V|^2$. The complexity of one iteration of the while loop is dominated by the computation required to update the longest and shortest path quantities $T_{fi(G)}$ and ρ_G respectively, both of which can be accomplished in $O(|V|^2)$ time. Thus, the time-complexity of the overall algorithm is $O(|V|^4)$. In practice, however, the number of resynchronization steps is much lower than $|V|^2$ since the constraints on the latency and on the introduction of cycles severely limit the number of resynchronization steps. Thus, our $O(|V|^4)$ bound can be viewed as a very conservative estimate.

9.1 Example

Figure 22 shows the synchronization graph topology that results from a six-processor schedule of a synthesizer for plucked-string musical instruments in 11 voices based on the Karplus-Strong technique. Here, exc represents the excitation input, each v_i represents the computation for the ith voice, and the actors marked with "+" signs specify adders. Execution time estimates for the actors are shown in the table at the bottom of the figure. In this example, exc and out are respectively the latency input and latency output, and the latency is 170. There are ten synchronization edges shown, and these are all irredundant.

Figure 23 shows how the number of synchronization edges in the result computed by our heuristic changes as the latency constraint varies. If just over 50 units of latency can be tolerated beyond the original latency of 170, then the heuristic is able to eliminate a single resynchronization edge. No further improvement can be obtained unless roughly another 50 units are allowed, at which point the number of synchronization edges drops to 8, and then down to 7 for an additional 8 time units of allowable latency. If the latency constraint is weakened to 382, just over twice the original latency, then the heuristic is able to reduce the number of synchronization edges to 6. No further improvement is achieved over the relatively long range of $(383 – 644)$. When $L_{max} \geq 645$, the latency of the pipelined solution (see Section 6) is permissible, and this allows the minimal cost of 5 synchronization edges for this system, which is half that of the original synchronization graph.

Figure 24 illustrates how the placement of synchronization edges changes as the heuristic



actor	execution time
exc	32
$v_1, v_2, \ldots, v_{11}$	51
out	16
+	04

Figure 22. The synchronization graph that results from a six processor schedule of a music synthesizer based on the Karplus-Strong technique.

L_{max}	# synch edges
$170 \leq L_{max} \leq 220$	10
$221 \leq L_{max} \leq 267$	9
$268 \leq L_{max} \leq 275$	8
$276 \leq L_{max} \leq 381$	7
$382 \leq L_{max} \leq 644$	6
$645 \leq L_{max} \leq \infty$	5

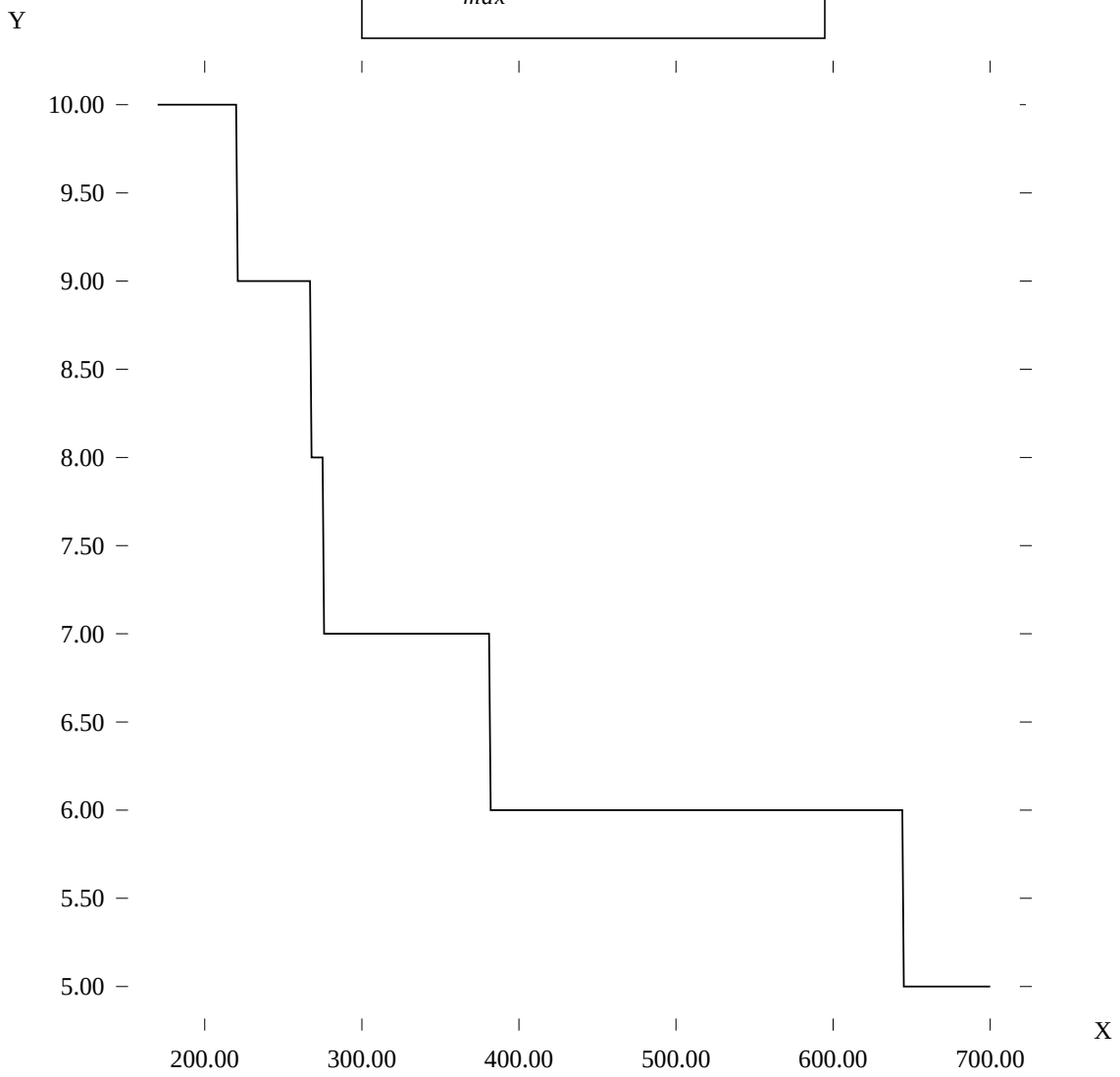


Figure 23. Performance of the heuristic on the example of Figure 22.

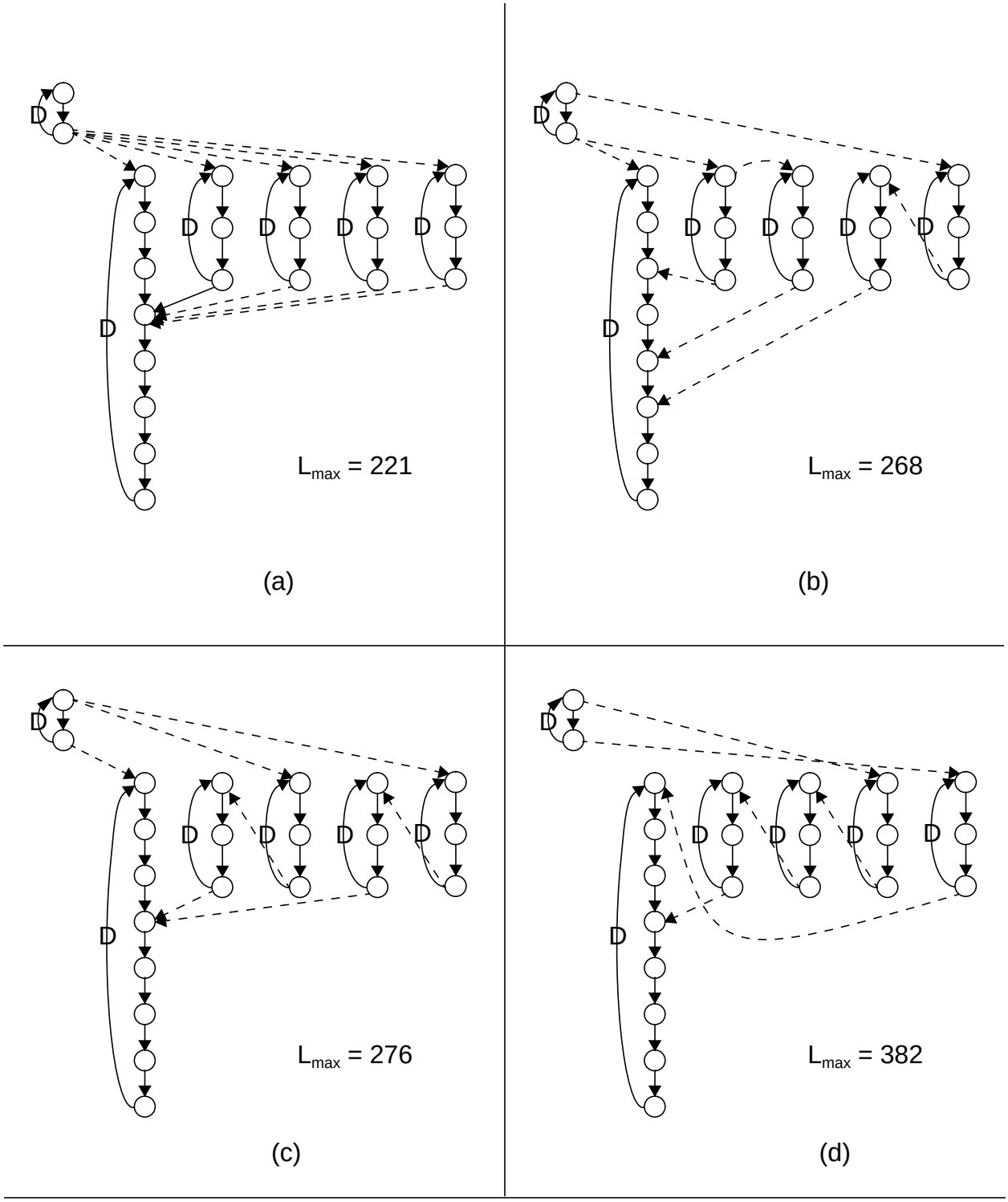


Figure 24. Synchronization graphs computed by the heuristic for different values of L_{max}.

is able to attain lower synchronization costs. Each of the four topologies corresponds to a break-point in the plot of Figure 23. For example Figure 24(a) shows the synchronization graph computed by the heuristic for the lowest latency constraint value for which it computes a solution that has 9 synchronization edges.

Note that synchronization graphs computed by the heuristic are not necessarily identical over any of the L_{max} ranges in Figure 23 in which the number of synchronization edges is constant. In fact, they can be significantly different. This is because even when there are no resynchronization candidates available that can reduce the net synchronization cost $(|\chi(*)| > 1)$, the heuristic attempts to insert resynchronization edges for the purpose of increasing the connectivity; this increases the chance that subsequent resynchronization candidates will be generated for which $|\chi(*)| > 1$. For example, Figure 25, shows the synchronization graph computed when L_{max} is just below the amount needed to permit the pipelined solution. Comparison with the graph shown in Figure 24(d) shows that even though these solutions have the same synchronization cost, the heuristic had much more room to pursue further resynchronization opportunities

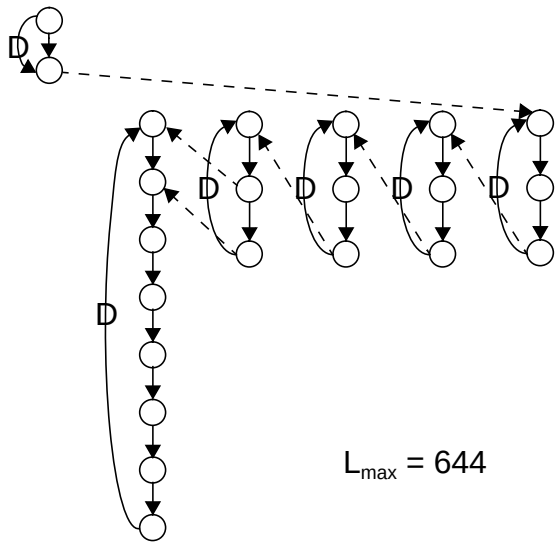


Figure 25. The synchronization graph computed by the heuristic for $L_{max} = 644$.

with $L_{max} = 644$, and thus, the graph of Figure 25 is more similar to the pipelined solution than it is to the solution of Figure 24(d).

Earlier, we mentioned that our $O(|V|^4)$ complexity expression is conservative since it is based on a $|V|^2$ bound on the number of iterations of the while loop in Figure 21, while in practice, the actual number of while loop iterations can be expected to be much less than $|V|^2$. This claim is supported by our music synthesis example, as shown in the graph of Figure 26. Here, the X-axis corresponds again to the latency constraint L_{max}, and the Y-coordinates give the number of while loop iterations that were executed by the heuristic. We see that between 5 and 13 iterations were required for each execution of the algorithm, which is not only much less than $|V|^2 = 484$, it is even less than $|V|$. This suggests that perhaps a significantly tighter bound on the number of while loop iterations can be derived.

10. Conclusions

This paper develops a post-optimization called resynchronization for self-timed, embedded multiprocessor implementations. The goal of resynchronization is to introduce new synchronizations in such a way that the number of additional synchronizations that become redundant exceeds the number of new synchronizations that are added, and thus the net synchronization cost is reduced. Two specific resynchronization problems are addressed. In the first context, which we refer to here as *unbounded-latency resynchronization*, we impose the constraint that resynchronization cannot degrade the throughput of the original schedule, and to ensure this constraint efficiently, we restrict resynchronization to involve only feedforward synchronization edges. In the second context, called *latency-constrained resynchronization*, we assume the same constraints and we consider a user-specified upper bound on latency that the resynchronized solution must respect.

We show that the general forms of both problems are intractable by deriving reductions from the classic set covering problem. For unbounded-latency resynchronization, we define a broad class of systems for which optimal resynchronization can be performed in polynomial time by a method that is roughly equivalent to pipelining. For latency-constrained resynchronization,

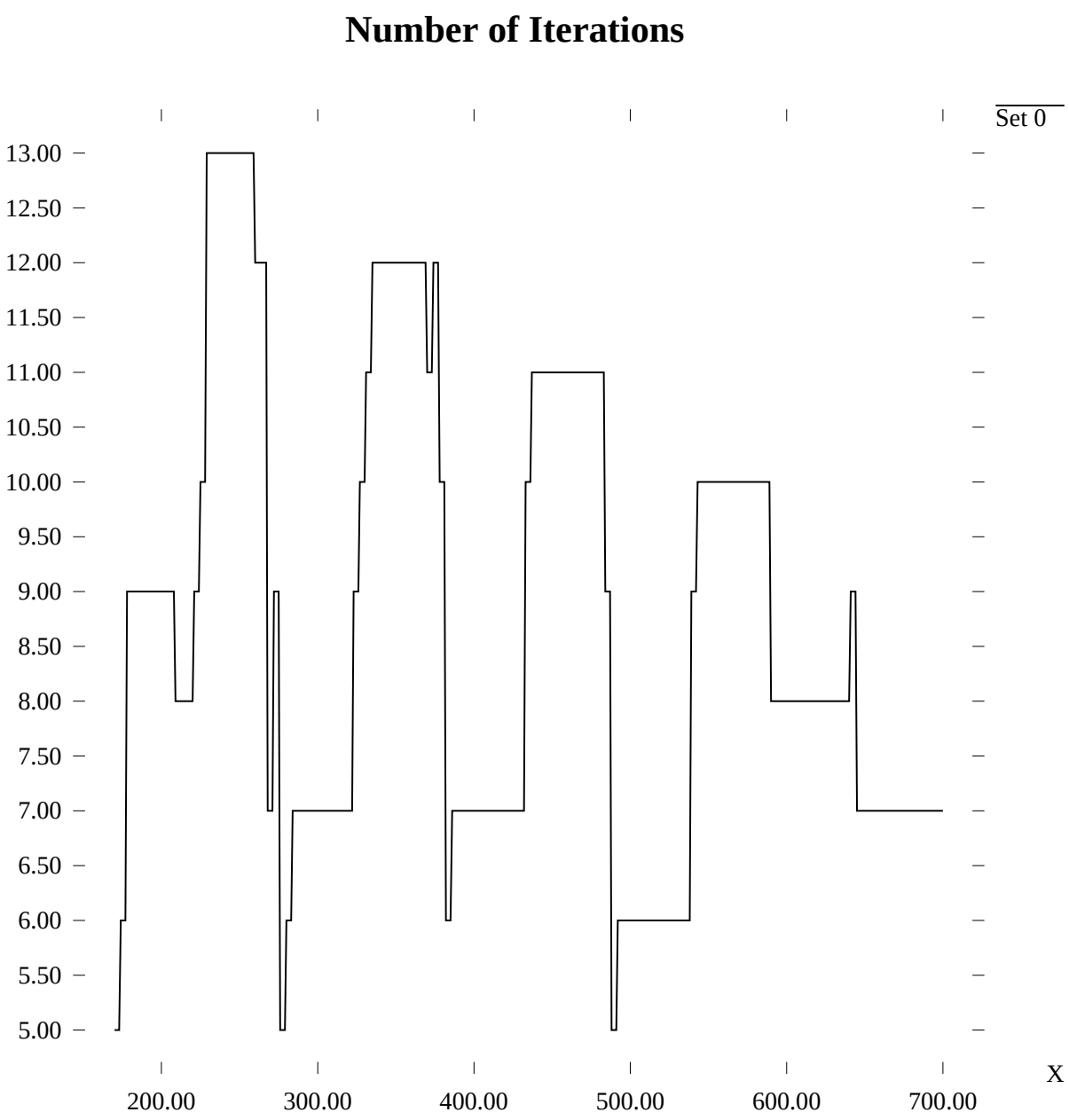


Figure 26. Number of resynchronization iterations versus L_{max} for the example of Figure 22.

we develop polynomial time techniques to optimally resynchronize two-processor systems. We also develop heuristic techniques for both problem contexts. For the unbounded-latency case, we show that a heuristic framework emerges naturally from the correspondence to set covering. Given an arbitrary heuristic for set covering, this framework generates a heuristic for unbounded-latency resynchronization. Based on a simple approximation algorithm for set covering that was developed previously, we propose a heuristic for latency-constrained resynchronization, and through an example of a music synthesis application, we illustrate the performance of our implementation of this heuristic. The results demonstrate that the technique can efficiently trade off between synchronization overhead and latency.

Several useful directions for future work emerge from our study. These include investigating whether efficient techniques can be developed that consider resynchronization opportunities within strongly connected components, rather than just accross feedforward edges; and developing more general measures of latency that can be computed efficiently. There may also be considerable room for improvement over our proposed heuristics, which are straightforward adaptations of existing set covering algorithms. In particular, it may be interesting to search for properties of practical synchronization graphs that could be exploited in addition to the correspondence with set covering. A related direction for further study is the exploration of additional useful special cases that can be resynchronized optimally, for both the unbounded-latency and latency-constrained contexts.

Appendix A

Proof of Argument #6 in Figure 12. By contraposition, we show that (w, z) cannot contribute to the elimination of any synchronization edge of G, and thus from Fact 2, it follows from the optimality of R that $(w, z) \notin R$. Suppose that (w, z) contributes to the elimination of some synchronization edge s. Then

$$\rho_{R(G)}(src(s), w) = \rho_{R(G)}(z, snk(s)) = 0. \quad (43)$$

From the matrix in Figure 13, we see that no resynchronization edge can have z as the source vertex. Thus, $snk(s) \in \{z, out\}$. Now, if $snk(s) = z$, then $s = (v, z)$, and thus from (43), there is a zero delay path from v to w in $R(G)$. However, the existence of such a path in $R(G)$ implies the existence of a path from in to out that traverses actors v, w, st_1, sx_1, which in turn implies that $L_{R(G)}(in, out) \geq 104$, and thus that R is not a valid LCR.

On the other hand, if $snk(s) = out$, then $src(s) \in \{z, sx_1, sx_2, sx_3, sx_4\}$. Now from (43), $src(s) = z$ implies the existence of a zero delay path from z to w in $R(G)$, which implies the existence of a path from in to out that traverses v, w, z, st_1, sx_1, which in turn implies that $L_{max} \geq 204$. On the other hand, if $src(s) = sx_i$ for some i, then since from Figure 13, there are no resynchronization edges that have an sx_i as the source, it follows from (43) that there must be a zero delay path in $R(G)$ from out to w. The existence of such a path, however, implies the existence of a cycle in $R(G)$ since $\rho_G(w, out) = 0$. Thus, $snk(s) = out$ implies that R is not an LCR. ■

Appendix B

Proof of Observation 3: Since R is an optimal LCR, we know that e must contribute to the elimination of at least one synchronization edge (from Fact 2). Let s be some synchronization edge such that e contributes to the elimination of s. Then

$$\rho_{R(G)}(src(s), src(e)) = \rho_{R(G)}(snk(e), snk(s)) = 0. \quad (44)$$

Now from Figure 13, it is apparent that there are no resynchronization edges in R that have sx_i or out as their source actor. Thus, from (44), $snk(s) = sx_i$ or $snk(s) = out$. Now, if $snk(s) = out$, then $src(s) = sx_k$ for some $k \neq i$, or $src(s) = z$. However, since no resynchronization edge has a member of $\{sx_1, sx_2, sx_3, sx_4\}$ as its source, we must (from 44) rule out $src(s) = sx_k$. Similarly, if $src(s) = z$, then from (44) there exists a zero delay path in $R(G)$ from z to st_j, which in turn implies that $L_{R(G)}(in, out) > 140$. But this is not possible since our assumption that R is an LCR guarantees that $L_{R(G)}(in, out) \leq 103$. Thus, we conclude that

$snk(s) \neq out$, and thus, that $snk(s) = sx_i$.

Now $(snk(s) = sx_i)$ implies that (a) $s = ex_i$ or (b) $s = (st_k, sx_i)$ for some k such that $x_i \in t_k$ (recall that $x_i \notin t_j$, and thus, that $k \neq j$). If $s = (st_k, sx_i)$, then from (44), $\rho_{R(G)}(st_k, st_j) = 0$. It follows that for any member $x_l \in t_j$, there is a zero delay path in $R(G)$ that traverses st_k, st_j and sx_l. Thus, $s = (st_k, sx_i)$ does not hold since otherwise $L_{R(G)}(in, out) \geq 140$.

Thus, we are left only with possibility (a) — $s = ex_i$. ■

Glossary

$\|S\|$:	The number of members in the finite set S.
$\rho(x, y)$:	Same as ρ_G with the DFG G understood from context.
$\rho_G(x, y)$:	If there is no path in G from x to y, then $\rho_G(x, y) = \infty$; otherwise, $\rho_G(x, y) = Delay(p)$, where p is any minimum-delay path from x to y.
$delay(e)$:	The delay on a DFG edge e.
$Delay(p)$:	Given a path p, $Delay(p)$ is the sum of the edge delays over all edges in p.
$d_n(u, v)$:	An edge whose source and sink vertices are u and v, respectively, and whose delay is equal to n.
λ_{max}:	The maximum cycle mean of a DFG.
$\chi(p)$:	The set of synchronization edges that are subsumed by the ordered pair of actors p.
$\langle(p_1, p_2, \ldots, p_k)\rangle$:	The *concatenation* of the paths $p_1, p_2, \ldots, p_k$.
2LCR:	Two-processor latency-constrained resynchronization.
connectivity:	The connectivity of a DFG G is the number of ordered vertex pairs (x, y) in G that satisfy $\rho_G(x, y) = 0$.
contributes to the elimination:	If G is a synchronization graph, s is a synchronization edge in G, R is a resynchronization of G, $s' \in R$, $s' \neq s$, and there is a path p from $src(s)$ to $snk(s)$ in $R(G)$ such that p contains s' and $Delay(p) \leq delay(s)$, then we say that s' contributes to the elimination of s

critical cycle: A fundamental cycle in a DFG whose cycle mean is equal to the maximum cycle mean of the DFG.

cycle mean: The cycle mean of a cycle C in a DFG is equal to T/D, where T is the sum of the execution times of all vertices traversed by C, and D is the sum of delays of all edges in C.

eliminates: If G is a synchronization graph, R is a resynchronization of G, and s is a synchronization edge in G, we say that R eliminates s if $s \notin R$.

$end(v, k)$: The time at which invocation k of actor v completes execution.

estimated throughput: Given a DFG with execution time estimates for the actors, the estimated throughput is the reciprocal of the maximum cycle mean.

execution source: In a synchronization graph, any actor that has no input edges or has non-zero delay on all input edges is called an execution source.

FBS: Feedback synchronization. A synchronization protocol that may be used for feedback edges in a synchronization graph. This protocol requires two synchronization accesses per schedule period.

feedback edge: An edge that is contained in at least one cycle.

feedforward edge: An edge that is not contained in a cycle.

FFS: Feedforward synchronization. A synchronization protocol that may be used for feedforward edges of the synchronization graph. This protocol requires four synchronization accesses per iteration period.

LCR: Latency-constrained resynchronization. Given a synchronization graph G, a latency input/output pair, and a positive integer L_{max}, a resynchronization R of G is an LCR if the latency of $R(G)$ is less than or equal to L_{max}.

maximum cycle mean: Given a DFG, the maximum cycle mean is the largest cycle mean over all cycles in the DFG.

resynchronization edge: Given a synchronization graph G and a resynchronization R, a resynchronization edge of R is any member of R that is not contained in G.

$R(G)$: If G is a synchronization graph and R is a resynchronization of G, then $R(G)$ denotes the graph that results from the resynchronization R.

SCC: Strongly connected component.

self loop:	An edge whose source and sink vertices are identical.
$start(v, k)$:	The time at which invocation k of actor v commences execution.
subsumes:	Given a synchronization edge (x_1, x_2) and an ordered pair of actors (y_1, y_2), (y_1, y_2) subsumes (x_1, x_2) if $\rho(x_1, y_1) + \rho(y_2, x_2) \leq delay((x_1, x_2))$.
$t(v)$:	The execution time or estimated execution time of actor v.
$T_{fi(G)}(x, y)$:	The sum of the actor execution times along a path from x to y in the first iteration graph of G that has maximum cumulative execution time.

References

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

[2] S. Banerjee, T. Hamada, P. M. Chau, and R. D. Fellman, "Macro Pipelining Based Scheduling on High Performance Heterogeneous Multiprocessor Systems," *IEEE Transactions on Signal Processing*, Vol. 43, No. 6, pp. 1468-1484, June, 1995.

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

[4] S. S. Bhattacharyya, S. Sriram, and E. A. Lee, "Minimizing Synchronization Overhead in Statically Scheduled Multiprocessor Systems," *Proceedings of the 1995 International Conference on Application Specific Array Processors*, Strasbourg, France, July, 1995.

[5] S. S. Bhattacharyya, S. Sriram, and E. A. Lee, *Optimizing Synchronization in Multiprocessor Implementations of Iterative Dataflow Programs*, Memorandum No. UCB/ERL M95/3, Electronics Research Laboratory, University of California at Berkeley, January, 1995.

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

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

[8] E. G. Coffman, Jr., *Computer and Job Shop Scheduling Theory*, Wiley, 1976.

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

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

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

[12] D. Filo, D. C. Ku, and G. De Micheli, “Optimizing the Control-unit through the Resynchronization of Operations,” *INTEGRATION, the VLSI Journal*, Vol. 13, pp. 231-258, 1992.

[13] G. R. Gao, R. Govindarajan, and P. Panangaden, “Well-Behaved Dataflow Programs for DSP Computation,” *Proceedings of the International Conference on Acoustics, Speech and Signal Processing*, March, 1992.

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

[15] P. Hoang, *Compiling Real Time Digital Signal Processing Applications onto Multiprocessor Systems*, Memorandum No. UCB/ERL M92/68, Electronics Research Laboratory, University of California at Berkeley, June, 1992.

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

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

[18] D. S. Johnson, “Approximation Algorithms for Combinatorial Problems,” *Journal of Computer and System Sciences*, Vol. 9, pp. 256-278, 1974.

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

[20] L. Lovasz, “On the Ratio of Optimal Integral and Fractional Covers,” *Discrete Mathematics*, Vol. 13, pp. 383-390, 1975.

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

[22] D.C. Ku, G. De Micheli, “Relative scheduling under timing constraints: algorithms for high-level synthesis of digital circuits,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol.11, No.6, pp. 696-718, June, 1992.

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

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

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

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

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

[28] B. Lim, A. Agarwal, “Waiting Algorithms for Synchronization in Large-Scale Multiprocessors,” *ACM Transactions on Computer Systems*, Vol. 11, No. 3, pp. 253-294, August, 1993.

[29] D. M. Nicol, “Optimal Partitioning of Random Programs Across Two Processors,” *IEEE Transactions on Computer*, Vol. 15, No. 2, February, pp. 134-141, 1989.

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

[31] K. K. Parhi, “High-Level Algorithm and Architecture Transformations for DSP Synthesis,” Journal of VLSI Signal Processing, January, 1995.

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

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

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

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

[36] 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.

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

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

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

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

[41] S. Sriram and E. A. Lee, “Statically Scheduling Communication Resources in Multiproces-

sor DSP architectures," *Proceedings of the Asilomar Conference on Signals, Systems, and Computers*, November, 1994.

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

[43] W. K. Stewart and S. A. Ward, "A Solution to the Special Case of the Synchronization Problem," *IEEE Transactions on Computers*, Vol. 37, No. 1, pp. 123-125, January, 1988.

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