Technical report, Digital Signal Processing Laboratory, University of Maryland at College Park, July, 1998. Revised from Memorandum UCB/ERL 96/56, Electronics Research Laboratory, University of California at Berkeley, October, 1996.

RESYNCHRONIZATION FOR MULTIPROCESSOR DSP SYSTEMS — PART 2: LATENCY-CONSTRAINED RESYNCHRONIZATION[1]

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

1. Abstract

The companion paper [5] introduced the concept of resynchronization, a post-optimization for static multiprocessor schedules in which extraneous synchronization operations are introduced in such a way that the number of original synchronizations that consequently become *redundant* significantly exceeds the number of additional synchronizations. Redundant synchronizations are synchronization operations whose corresponding sequencing requirements are enforced completely by other synchronizations in the system. The amount of run-time overhead required for synchronization can be reduced significantly by eliminating redundant synchronizations [4, 30]. Thus, effective resynchronization reduces the net synchronization overhead in the implementation of a multiprocessor schedule, and improves the overall throughput.

However, since additional serialization is imposed by the new synchronizations, resynchronization can produce significant increase in latency. The companion paper [5] develops fundamental properties of resynchronization and studies the problem of optimal resynchronization under the assumption that arbitrary increases in latency can be tolerated ("maximum-throughput resynchronization"). Such an assumption is valid, for example, in a wide variety of simulation applications. This paper addresses the problem of computing an optimal resynchronization among all resynchronizations that do not increase the latency beyond a prespecified upper bound L_{max}.

1. S. S. Bhattacharyya is with the Department of Electrical Engineering and the Institute for Advanced Computer Studies, University of Maryland, College Park, (ssb@eng.umd.edu).

S. Sriram is with the DSP R&D Research Center, Texas Instruments, Dallas, Texas, (sriram@hc.ti.com).

E. A. Lee is with the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, (eal@eecs.berkeley.edu).

This research is part of the Ptolemy project, which is supported by the Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory, the State of California MICRO program, and the following companies: The Alta Group of Cadence Design Systems, Hewlett Packard, Hitachi, Hughes Space and Communications, NEC, Philips, and Rockwell.

Our study is based in the context of self-timed execution of iterative dataflow specifications, which is an implementation model that has been applied extensively for digital signal processing systems.

Latency constraints become important in interactive applications such as video conferencing, games, and telephony, where beyond a certain point latency becomes annoying to the user. This paper demonstrates how to obtain the benefits of resynchronization while maintaining a specified latency constraint.

2. Introduction

In a shared-memory multiprocessor system, it is possible that certain synchronization operations are *redundant,* which means that their sequencing requirements are enforced entirely by other synchronizations in the system. It has been demonstrated that the amount of run-time overhead required for synchronization can be reduced significantly by detecting and eliminating redundant synchronizations [4, 30].

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 greater that the number of new synchronizations. Thus, effective resynchronization improves the overall throughput of a multiprocessor implementation by decreasing the average rate at which synchronization operations are performed. However, since additional serialization is imposed by the new synchronizations, resynchronization can produce significant increase in latency. For some applications, such an increased latency is tolerable; examples are video and audio playback from media such as Digital Video Disk (DVD). For other applications, teleconferencing for example, an increase in latency may not be tolerable. In voice telephony, for example, a round trip delay greater than 40 milliseconds is perceived as an annoying echo. Such a limit on tolerable latency sets a *latency constraint* L_{max}. This paper addresses the problem of computing an optimal resynchronization among all resynchronizations that do not increase the latency beyond a specified upper bound L_{max}. This enables us to realize some of the benefits of reduced synchronization

overhead due to resynchronization, while maintaining the required latency constraint.

We address this problem in the context of *self-timed* scheduling of *iterative synchronous dataflow* [17] specifications on multiprocessor systems. Please refer to Section 2 of the companion paper [5] for elaboration of these concepts. Interprocessor communication (**IPC**) and synchronization are assumed to take place through shared memory, which could be global memory between all processors, or it could be distributed between pairs of processors.

3. Synchronization redundancy and resynchronization

Please refer to Sections 3 of the companion paper [5] for a review of relevant background (primarily from graph theory) and notation, and to Section 4 of the companion paper [5] for a description of the *synchronization protocols* (FFS and FBS) assumed in this paper and a review of our *synchronization graph* modeling methodology for analyzing self-timed execution of iterative dataflow specifications.

Any transformations that we perform on the synchronization graph must respect the synchronization constraints implied by the original synchronization graph. If we ensure this, then we only need to implement the synchronization edges of the optimized synchronization graph. 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), we say that G_1 **preserves** G_2 if for all $e \in E_2$ such that $e \notin E_1$, we have $\rho_{G_1}(src(e), snk(e)) \leq delay(e)$. The following theorem, developed in [4], underlies the validity of resynchronization.

Theorem 1: The synchronization constraints of G_1 imply the constraints of G_2 if G_1 preserves G_2.

A synchronization edge is **redundant** in a synchronization graph G if its removal yields a graph that preserves G. If all redundant edges in a synchronization graph are removed, then the resulting graph preserves the original synchronization graph [4].

Given a synchronization graph G, a synchronization edge (x_1, x_2) in G, and an ordered pair of actors (y_1, y_2) in G, we say that (y_1, y_2) **subsumes** (x_1, x_2) in G if

$$\rho_G(x_1, y_1) + \rho_G(y_2, x_2) \leq delay((x_1, x_2)).$$

Thus, intuitively, (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. If S is the set of synchronization edges in G, and p is an ordered pair of actors in G, then $\chi(p) \equiv \{s \in S | p \text{ subsumes } s\}$.

If $G = (V, E)$ is a synchronization graph and F is the set of feedforward edges in G, then a **resynchronization** of G is a 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 $e_1', e_2', \ldots, e_m'$ are feedforward edges in the DFG $G^* \equiv (V, (E - F) + R)$, and G^* preserves G. Each member of R that is not in E is a **resynchronization edge**, G^* is called the **resynchronized graph** associated with R, and this graph is denoted by $\Psi(R, G)$.

Our concept of resynchronization considers the rearrangement of synchronizations only across feedforward edges. We impose this restriction so that the serialization imposed by resynchronization does not degrade the estimated throughput [5].

4. Elimination of synchronization edges

In this section, we introduce a number of useful properties that pertain to the process by which resynchronization can make certain synchronization edges in the original synchronization graph become redundant. The following definition is fundamental to these properties.

Definition 1: If G is a synchronization graph, s is a synchronization edge in G that is not redundant, R is a resynchronization of G, and s is not contained in R, then we say that R **eliminates** s. If R eliminates s, $s' \in R$, and there is a path p from $src(s)$ to $snk(s)$ in $\Psi(R, G)$ 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 s "by itself". In such cases, it is the contribution of all of the resynchronization edges within the path p that enables the elimination of s. This motivates our choice of terminology in Definition 1. An example is shown in Figure 1.

The following two facts follow immediately from Definition 1.

Fact 1: 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.

Fact 2: Suppose that G is a synchronization 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 2(a). Figure 2(b) shows a resynchronization R of G. In the resynchronized graph of Figure 2(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 1 guarantees that $R' \equiv R - \{(x_4, y_3)\}$, illustrated in Figure 2(c), is also a resynchronization of G. In Figure 2(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 Fact 1, we have that $R'' \equiv R' - \{(x_5, y_4)\} + \{(x_5, y_5)\}$, illustrated in Figure 2(d), is a also resynchronization of G.

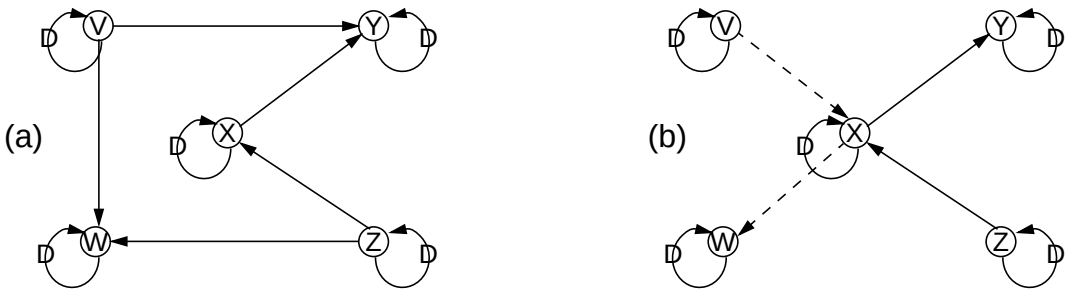


Figure 1. An illustration of Definition 1. Here each processor executes a single actor. A resynchronization of the synchronization graph in (a) is illustrated in (b). In this resynchronization, the resynchronization edges (V, X) and (X, W) both contribute to the elimination of (V, W).

5. Latency-constrained resynchronization

As discussed in Section 3, 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.

Definition 2: Suppose G_0 is an application DFG, G is a synchronization graph that results from a multiprocessor schedule for G_0, x is an execution source (an actor that has no input edges or has nonzero delay on all input edges) in G, and y is an actor in G other than x. 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**

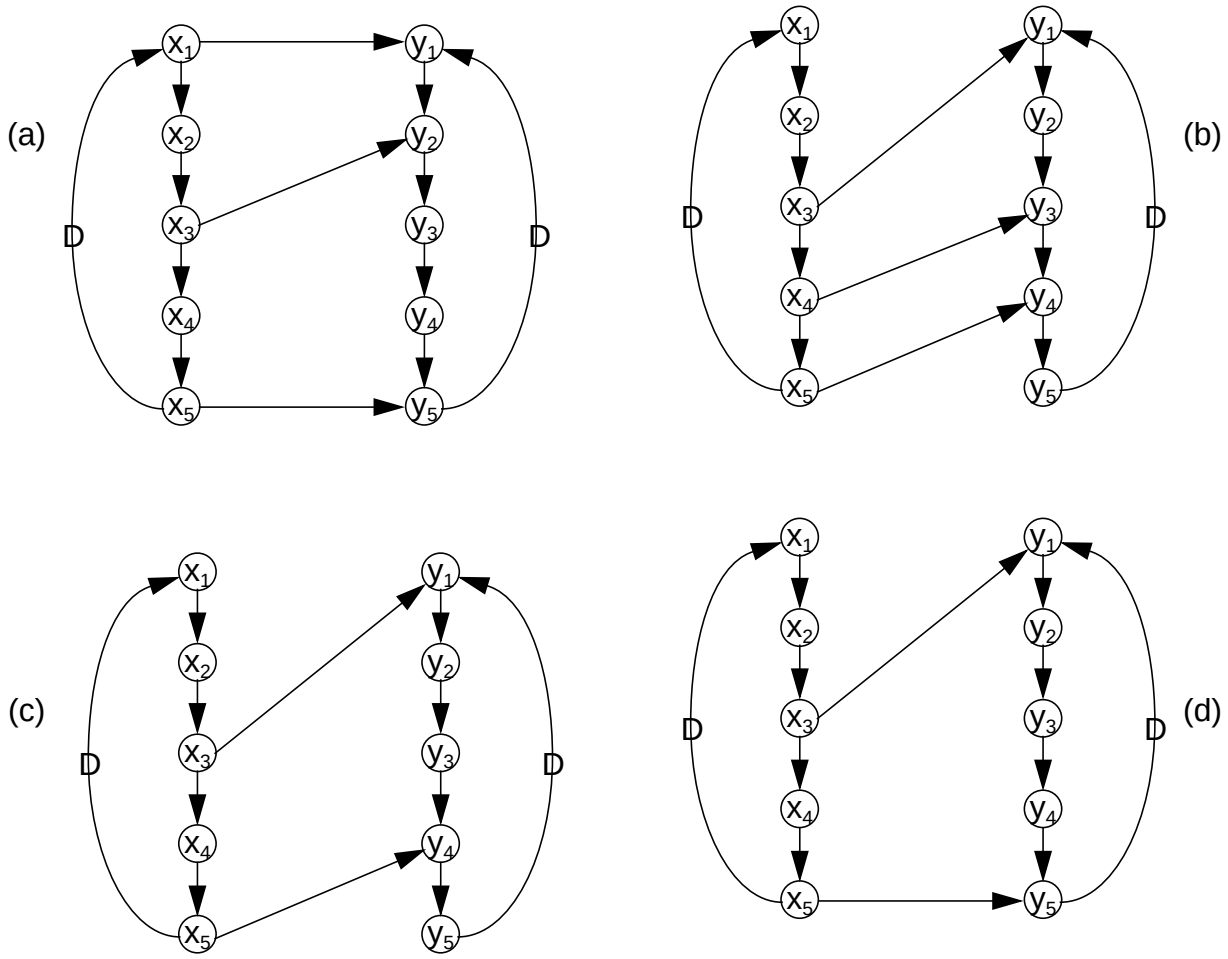


Figure 2. Properties of resynchronization.

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, and thus the latency corresponds to the critical path in the dataflow implementation to the first output invocation that is influenced by the input. This interpretation of the latency as the critical path is widely used in VLSI signal processing [14, 20].

In general, the latency can be computed by performing a simple simulation of the ASAP execution for G through the $(1 + \rho_{G_0}(x, y))$ th execution of y. Such a simulation can be performed as a functional simulation of a DFG G_{sim} that has the same topology (vertices and edges) as G, and that maintains the simulation time of each processor in the values of data tokens. Each initial token (delay) in G_{sim} is initialized to have the value 0, since these tokens are all present at time 0. Then, a data driven simulation of G_{sim} is carried out. In this simulation, an actor may execute whenever it has sufficient data, and the value of the output token produced by the invocation of any actor z in the simulation is given by

$$max(\{v_1, v_2, \ldots, v_n\}) + t(z), \tag{1}$$

where $\{\{v_1, v_2, \ldots, v_n\}\}$ is the set of token values consumed during the actor execution. In such a simulation, the i th token value produced by an actor z gives the completion time of the i th invocation of z in the ASAP execution of G. Thus, the latency can be determined as the value of the $(1 + \rho_{G_0}(x, y))$ th output token produced by y. With careful implementation of the functional simulator described above, the latency can be determined in $O(d \times max(\{|V|, s\}))$ time, where $d = 1 + \rho_{G_0}(x, y)$, and s denotes the number of synchronization edges in G. The simulation approach described above is similar to approaches described in [32]

For a broad class of synchronization graphs, latency can be analyzed even more efficiently during resynchronization. This is the class of synchronization graphs in which the first invocation of the latency output is influenced by the first invocation of the latency input. Equivalently, it is

1. Recall from the companion paper [5] 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.

the class of graphs that contain at least one delayless path in the corresponding application DFG directed from the latency input to the latency output. For transparent synchronization graphs, we can directly apply well-known longest-path based techniques for computing latency.

Definition 3: Suppose that G_0 is an application DFG, x is a source actor in G_0, and y is an actor in G_0 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-iteration graph** of G, denoted $\phi(G)$, is constructed by removing all edges from G that have non-zero-delay; adding a vertex υ, which represents the beginning of execution; 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 3 illustrates the derivation of $\phi(G)$.

Given two vertices x and y in $\phi(G)$ such that there is a path in $\phi(G)$ 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 $T_{\phi(G)}(x, y)$. That is,

$$T_{\phi(G)}(x, y) = max\left(\sum_{p \text{ traverses z}} t(z) \,\middle|\, (p \text{ is a path from } x \text{ to } y \text{ in } \phi(G)) \right). \quad (2)$$

If there is no path from x to y, then we define $T_{\phi(G)}(x, y)$ to be $-\infty$. Note that for all x, y $T_{\phi(G)}(x, y) < +\infty$, since $\phi(G)$ is acyclic. The values $T_{\phi(G)}(x, y)$ for all pairs x, y can be com-

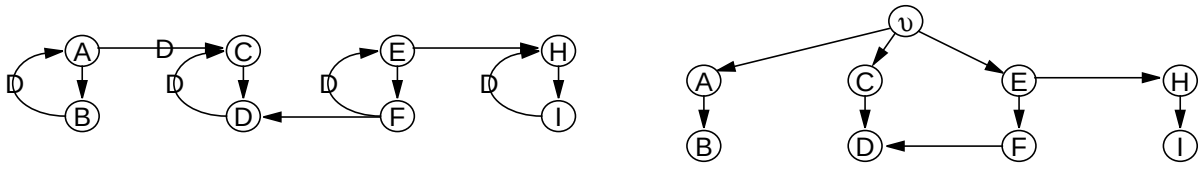


Figure 3. An example used to illustrate the construction of $\phi(G)$. The graph on the right is $\phi(G)$ if G is the left-side graph.

puted 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 [10].

Fact 3: 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 G is a resynchronization G_s. Then $\rho_G(x, y) = 0$, and thus $T_{\mathcal{fi}(G)}(x, y) \geq 0$ (i. e. $T_{\mathcal{fi}(G)}(x, y) \neq -\infty$).

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 in the companion paper [5] that $\rho_G(x, y) = 0$. *QED.*

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

Theorem 2: Suppose that G is a synchronization graph that is transparent with respect to latency input x and latency output y. Then $L_G(x, y) = T_{\mathcal{fi}(G)}(\upsilon, y)$.

Proof: By induction, we show that for every actor w in $\mathcal{fi}(G)$,

$$end(w, 1) = T_{\mathcal{fi}(G)}(\upsilon, w), \tag{3}$$

which clearly implies the desired result.

First, let $mt(w)$ denote the maximum number of actors that are traversed by a path in $\mathcal{fi}(G)$ (over all paths in $\mathcal{fi}(G)$) that starts at υ and terminates at w. If $mt(w) = 1$, then clearly $w = \upsilon$. Since both the LHS and RHS of (3) are identically equal to $t(\upsilon) = 0$ when $w = \upsilon$, we have that (3) holds whenever $mt(w) = 1$.

Now suppose that (3) holds whenever $mt(w) \leq k$, for some $k \geq 1$, and consider the sce-

nario $mt(w) = k + 1$. Clearly, in the self-timed (ASAP) execution of G, invocation w_1, the first invocation of w, commences as soon as all invocations in the set

$$Z = \{z_1 | (z \in P_w)\}$$

have completed execution, where z_1 denotes the first invocation of actor z, and P_w is the set of predecessors of w in $fi(G)$. All members $z \in P_w$ satisfy $mt(z) \leq k$, since otherwise $mt(w)$ would exceed $(k + 1)$. Thus, from the induction hypothesis, we have

$$start(w, 1) = max(end(z, 1) | (z \in P_w)) = max(T_{fi(G)}(\upsilon, z) | (z \in P_w)),$$

which implies that

$$end(w, 1) = max(T_{fi(G)}(\upsilon, z) | (z \in P_w)) + t(w). \quad (4)$$

But, by definition of $T_{fi(G)}$, the RHS of (4) is clearly equal to $T_{fi(G)}(\upsilon, w)$, and thus we have that $end(w, 1) = T_{fi(G)}(\upsilon, w)$.

We have shown that (3) 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, (3) holds for all values of $mt(w)$. *QED.*

In the context of resynchronization, the main 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 $T_{fi(G)}(a, b)$ for all actor pairs (a, b). We will discuss this further in Section 9.

Since many practical application graphs contain delayless paths from input to output and these graphs admit a particularly efficient means for computing latency, we have targeted our implementation of latency-constrained resynchronization to the class of transparent synchronization graphs. However, the overall resynchronization framework described in this paper does not depend on any particular method for computing latency, and thus, it can be fully applied to general graphs (with a moderate increase in complexity) using the ASAP simulation approach mentioned above. Our framework can also be applied to subclasses of synchronization graphs other than transparent graphs for which efficient techniques for computing latency are discovered.

Definition 4: An instance of the **latency-constrained resynchronization problem** consists of a 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 such that 1) $L_{\Psi(R, G)}(x, y) \leq L_{max}$, 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 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 $L_{\Psi(R, G)}(x, y) \leq L_{max}$. Thus, the latency-constrained resynchronization problem is the problem of determining a minimal LCR.

6. Related work

In [4], an 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 the more efficient FBS protocol. 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 overhead required to implement the new edges that are added by *Convert-to-SC-graph* can be significantly less than the net overhead that is eliminated by converting all uses of FFS to FBS. However, this technique may increase the latency.

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 transparent 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.

Trade-offs between latency and throughput have been studied by Potkonjac and Srivastava in the context of transformations for dedicated implementation of linear computations [26]. Because this work is based on synchronous implementations, it does not address the synchronization issues and opportunities that we encounter in our self-timed dataflow context.

7. Intractability of LCR

In this section we show that the latency-constrained resynchronization problem is NP-hard even for the very restricted subclass of synchronization graphs in which each SCC corresponds to a single actor, and all synchronization edges have zero delay.

As with the maximum-throughput resynchronization problem [5], the intractability of this special case of latency-constrained resynchronization can be established by a reduction from set covering. To illustrate this reduction, we 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 $t_3 = \{x_2, x_4\}$. Figure 4 illustrates the instance of latency-constrained resynchronization that we derive from the instance of set covering specified by (X, T). Here, each actor corresponds to a single processor and the self loop edge for each actor is 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 4, 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 6, with redundant resynchronization edges removed. Since the resynchronization candidates $(v, st_1), (v, st_3)$ were chosen to obtain the solution shown in Figure 6, this solution corre-

sponds 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 4 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

$$(v, st_i),\ i \in \{1, 2, 3\}, \text{ or of the form } (st_j, sx_i),\ x_i \notin t_j. \tag{5}$$

The first observation is immediately apparent from inspection of Figure 4. A proof of the second observation follows.

Proof of Observation 2 We must show that no other resynchronization edges can be contained in an optimal LCR of G. Figure 6 specifies arguments with which we can discard all possibilities

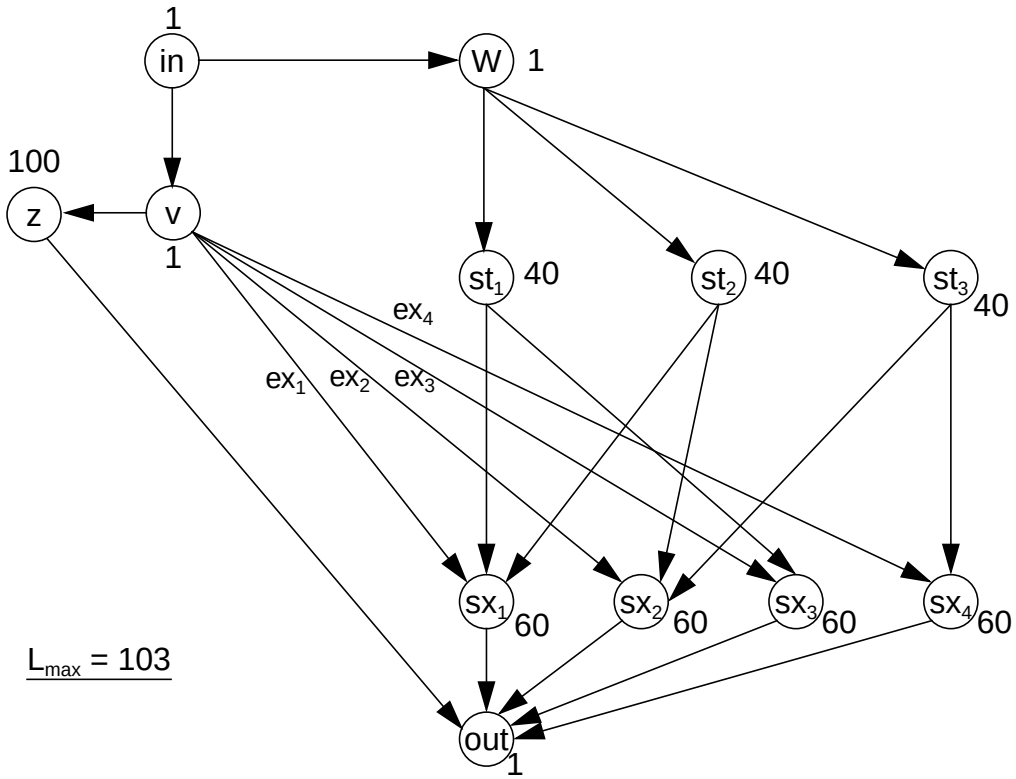


Figure 4. An instance of latency-constrained resynchronization that is derived from an instance of the set covering problem.

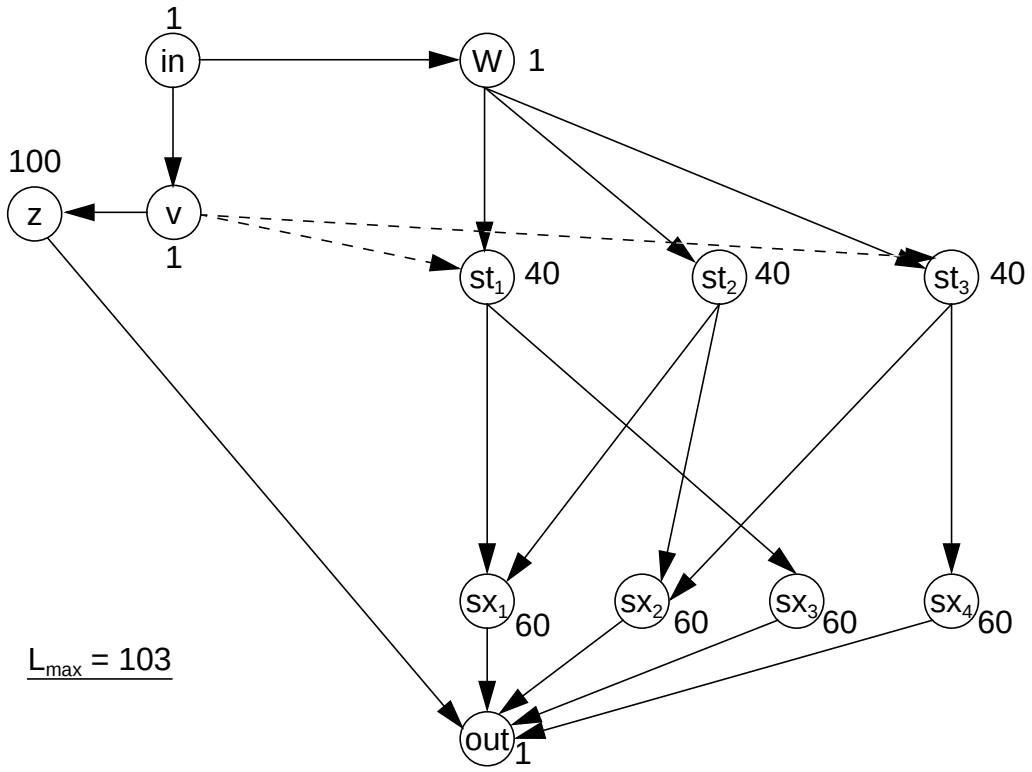


Figure 5. The synchronization graph that results from a solution to the instance of latency-constrained resynchronization shown in Figure 4.

	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 in [5]).
5. Introduces a delayless self loop.
6. Proof is given below.

Figure 6. Arguments that support Observation 2.

other than those given in (5). In the matrix shown in Figure 6, 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 4. A proof of argument #6 follows shortly within this same section.

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 would be increased to at least 204; $(in, z) \notin R$ from Lemma 2 in the companion paper [5] 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 6 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 6 marked *OK* are simply those that correspond to (5), and thus we have justified Observation 2. *QED.*

In the proof of Observation 2, we deferred the proof of Argument #6 for Figure 1. We now present the proof of this argument.

Proof of Argument #6 in Figure 1. By contraposition, we show that (w, z) cannot contribute to the elimination of any synchronization edge of G, and thus from Fact 1, 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

$$\rho_{\tilde{G}}(src(s), w) = \rho_{\tilde{G}}(z, snk(s)) = 0, \tag{6}$$

where

$$\tilde{G} \equiv \Psi(R, G). \tag{7}$$

From the matrix in Figure 6, 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 (6), there is a zero delay path from v to w in $\tilde{G}$. However, the existence of such a path in $\tilde{G}$ implies the existence of a path from in to out that traverses actors v, w, st_1, sx_1, which in turn implies that $L_{\tilde{G}}(in, out) \geq 104$, 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 (6), $src(s) = z$ implies the existence of a zero delay path from z to w in $\tilde{G}$, 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 6, there are no resynchronization edges that have an sx_i as the source, it follows from (6) that there must be a zero delay path in $\tilde{G}$ from out to w. The existence of such a path, however, implies the existence of a cycle in $\tilde{G}$ since $\rho_G(w, out) = 0$. Thus, $snk(s) = out$ implies that R is not an LCR. *QED.*

The following observation states that a resynchronization edge of the form (st_j, sx_i) contributes to the elimination of exactly one synchronization edge, which is the edge ex_i.

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: Since R is an optimal LCR, we know that e must contribute to the elimination of at least one synchronization edge (from Fact 1). Let s be some synchronization edge such that e contributes to the elimination of s. Then

$$\rho_{R(G)}(src(s), src(e)) = \rho_{R(G)}(snk(e), snk(s)) = 0. \quad (8)$$

Now from Figure 6, it is apparent that there are no resynchronization edges in R that have sx_i or out as their source actor. Thus, from (8), $snk(s) = sx_i$ 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 8) rule out

$src(s) = sx_k$. Similarly, if $src(s) = z$, then from (8) 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 (8), $\rho_{R(G)}(st_k, st_j) = 0$. 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) — $s = ex_i$. *QED.*

Now, suppose that we are given an optimal LCR R of G. From Observation 3 and Fact 1, 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 all resynchronization edges in R' are of the form (v, st_i).

For each $x_i \in X$ such that

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

we have that $ex_i \notin R'$. This is because R' is assumed to be optimal, and thus, $\Psi(R, G)$ 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

each resynchronization edge in R'' is of the form (v, st_i), and (10)

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

It is easily verified that the set of synchronization edges eliminated by R'' is $\{ex_i \,|\, x_i \in X\}$. Thus,

the set $T' \equiv \{t_j | (v, t_j)$ 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 4) that given an arbitrary cover T_a for X, the resynchronization defined by

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

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.

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). This example of the reduction from set covering to latency-constrained resynchronization 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 7.

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

- 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 4 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 7. A procedure for constructing an instance I_{lr} 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}.

follow the steps in Figure 7. This generalization is not conceptually difficult (although it is rather tedious) since it is easily verified that all of the arguments in Figure 7 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.

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 simple cycle. This reveals a pattern of complexity that is analogous to the classic nonpreemptive 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 [9, 13]. 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 [21].

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 non-redundant 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 $L_{\Psi(R, G)}(x_1, y_q) \leq L_{max}$. In the remainder of this section, we denote the synchronization graph corresponding to our generic instance of 2LCR by $\tilde{G}$.

We assume that $delay(s_i) = 0$ for all s_i, and we refer to the subproblem that results from this restriction as **delayless 2LCR**. In this section we present an algorithm that solves the delayless 2LCR problem in $O(N^2)$ time, where N is the number of vertices in $\tilde{G}$. We also give an extension of this algorithm to the general 2LCR problem (arbitrary delays can be present).

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 $b_n = N$.

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 $\Psi(R, \tilde{G})$, there can be at most one synchronization edge in any path directed from $src(s)$ to $snk(s)$. *QED.*

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

$$L_{\Psi(R, \tilde{G})}(x_1, y_q) = max(\{t_{pred}(src(s')) + t_{succ}(snk(s')) \,|\, s' \in R\}), \text{ where}$$
$$t_{pred}(x_i) \equiv \sum_{j \le i} t(x_j) \text{ for } i = 1, 2, \ldots, p, \text{ and } t_{succ}(y_i) \equiv \sum_{j \ge i} t(y_j) \text{ for } i = 1, 2, \ldots, q.$$

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. *QED.*

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

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

Consider the example shown in Figure 8. Here, we assume that $t(z) = 1$ for each actor z, and $L_{max} = 10$. From (13), we have

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

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 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 $\tilde{G}$, and suppose that (x_a, y_b) is a delayless syn-

chronization 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 and the assumption that (x_a, y_b) is delayless, 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)

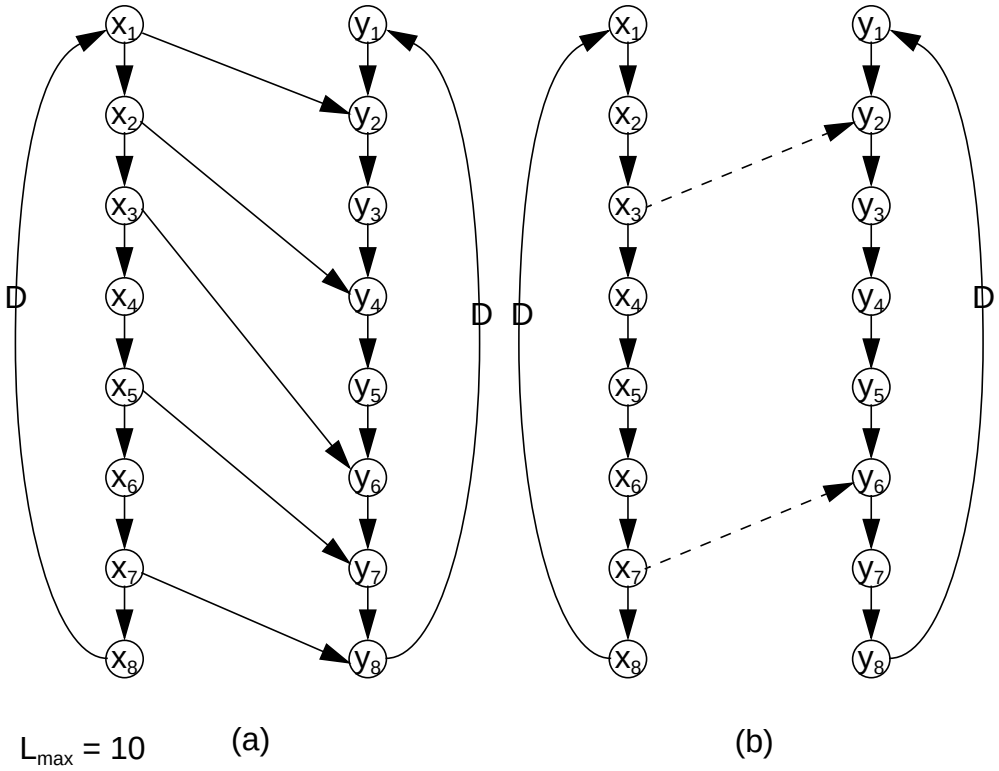


Figure 8. An instance of delayless, two-processor latency-constrained resynchronization. In this example, the execution times of all actors are identically equal to unity.

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. *QED.*

From Fact 2 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 $d_0(v_i)$. 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 8(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

$$\chi(v_1) = \{s_1'\}, \chi(v_2) = \{s_1', s_2'\}, \chi(v_3) = \{s_1', s_2', s_3'\}, \chi(v_4) = \{s_2', s_3'\}$$

$$\chi(v_5) = \{s_2', s_3', s_4'\}, \chi(v_6) = \{s_3', s_4'\}, \chi(v_7) = \{s_3', s_4', s_5'\}, \chi(v_8) = \{s_4', s_5'\}, \quad (19)$$

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

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

Now clearly

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

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 not redundant. 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

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

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

From Observation 7 and the preceding discussion, we conclude that an optimal LCR of $\tilde{G}$ can be 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 $\chi(v_i) \equiv \{(x_a, y_b) \in S | a \leq i$ **and** $b \geq j\}$, 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 3: Polynomial-time solutions (quadratic in the number of synchronization graph vertices) 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 8(a) and use the technique described earlier 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 8(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 9. 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

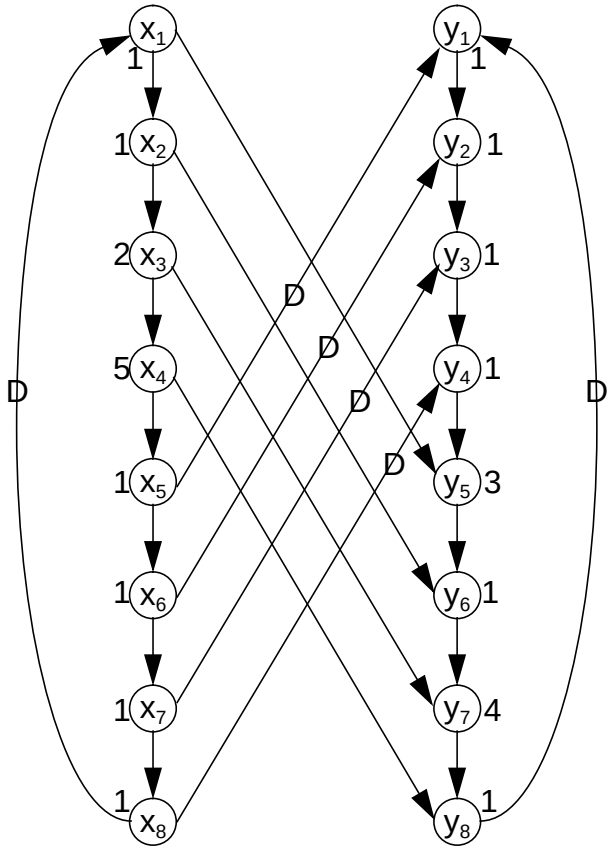
$$\text{each subset } \chi(v_i) \text{ is of the form } \{s_a', s_{a+1}', \ldots, s_b'\}, 1 \leq a \leq b \leq 8. \quad (25)$$

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

$$\chi(v_1) = \{s_1', s_7', s_8'\}, \chi(v_2) = \{s_1', s_2', s_8'\}, \chi(v_3) = \{s_2', s_3'\}$$
$$\chi(v_4) = \{s_4'\}, \chi(v_5) = \{s_4', s_5'\}, \chi(v_6) = \{s_4', s_5', s_6'\}, \quad (26)$$

and these subsets are clearly not all consistent with the form specified in (25). Thus, the algorithm developed in Subsection 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 $v_i = (x_i, y_j)$



$L_{max} = 12$

Figure 9. A synchronization graph with unit delays on some of the synchronization edges.

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 9(a), $v_1 = (x_1, y_3)$ 1-subsumes 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 3, 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 none of the synchronization edges in $\tilde{G}$ are redundant.

The other parts can be verified easily from the structure of $\tilde{G}$, including the assumption that no synchronization edge in $\tilde{G}$ is redundant. 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 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 10. Figure 10(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 10(b). Observe that this resynchronization is an LCR since only delayless synchronization edges affect the

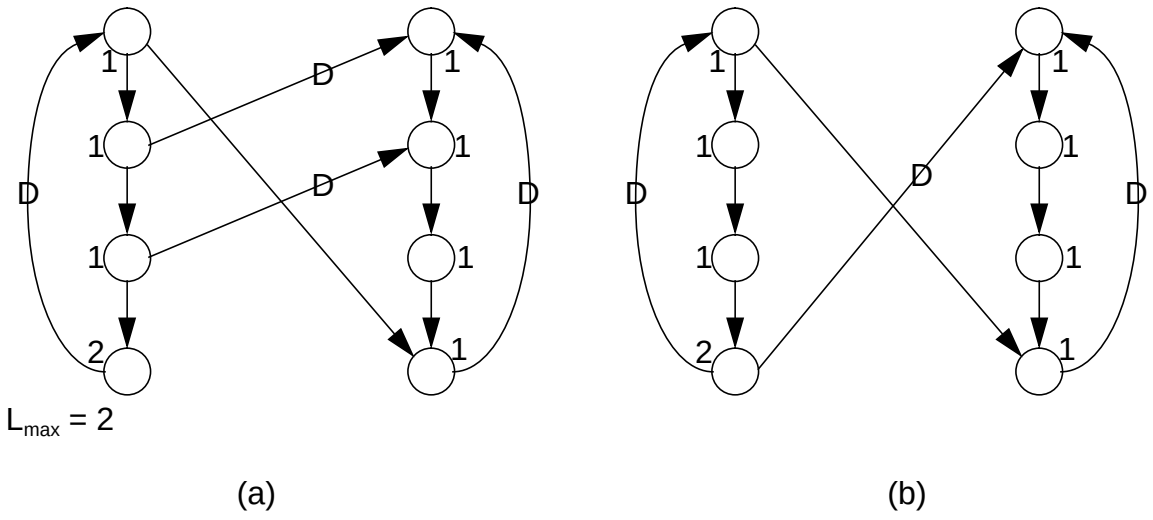


Figure 10. An example in which constraining all resynchronization edges to be delayless precludes the ability to derive an optimal resynchronization.

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 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 4: 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 $e \in P$. n

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 4, we know that G_b preserves $Z(\tilde{G})$. Thus, from Lemma 1 in the companion paper [5], 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 $L_{Z(\tilde{G})}(x_1, y_q) \leq L_{max}$. n

Theorem 4, 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 can

be used.

Recall the example of Figure 9(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 11(a)-(e). The latencies of the graphs in Figure 11(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 11(c) (from Corollary 1). This is done by first removing redundant edges from the graph (yielding the graph in Fig-

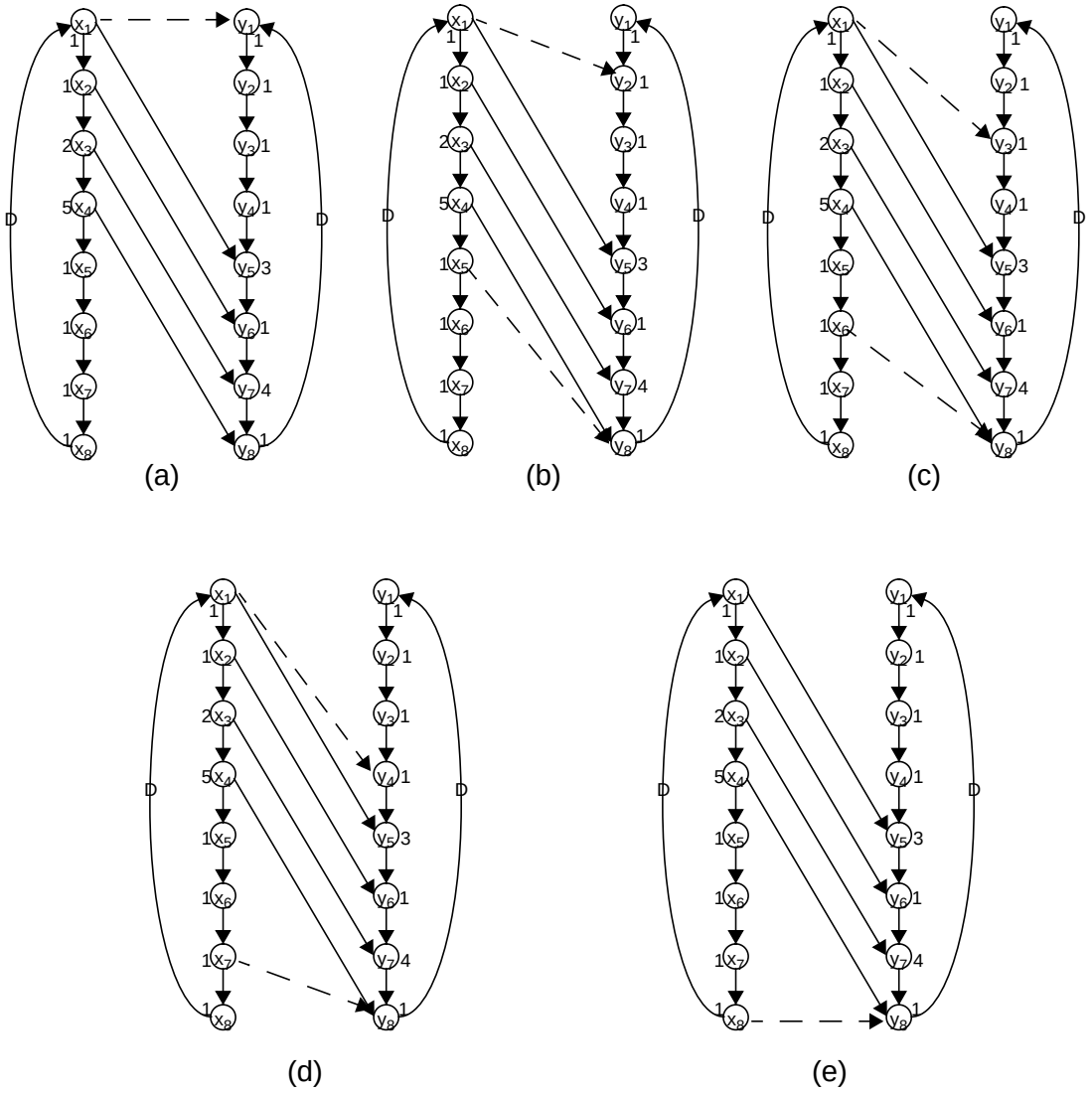


Figure 11. The synchronization graphs considered in Algorithm B for the example in Figure 9.

ure 12(b)) and then applying the algorithm developed in Section . For the synchronization graph of Figure 12(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 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 12(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

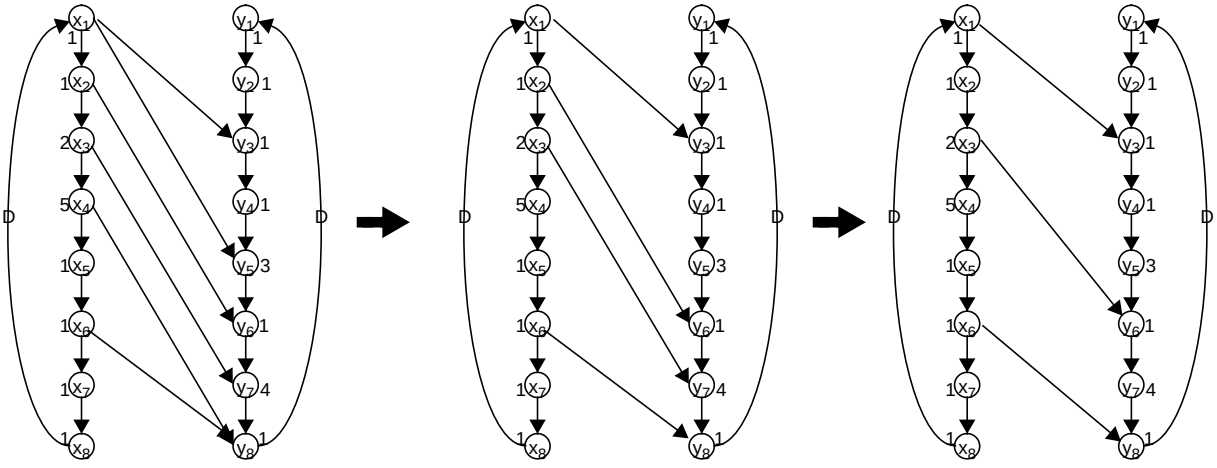


Figure 12. Derivation of an optimal LCR for the synchronization graph of Figure 11(c).

($|ud(\tilde{G})| + 1$) 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), 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 5: Polynomial-time solutions exist for the general two-processor latency-constrained resynchronization problem.

The example in Figure 10 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 9. From Figure 12(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 13. 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 general synchronization graphs

The companion paper [5] presents a heuristic called Global-resynchronize for the maximum-throughput resynchronization problem, which is the problem of determining an optimal resynchronization under the assumption that arbitrary increases in latency can be tolerated. In this section, we extend Algorithm Global-resynchronize to derive an efficient heuristic that addresses the latency-constrained resynchronization problem for general synchronization graphs. Given an input synchronization graph G, Algorithm Global-resynchronize operates by first computing the family of subsets

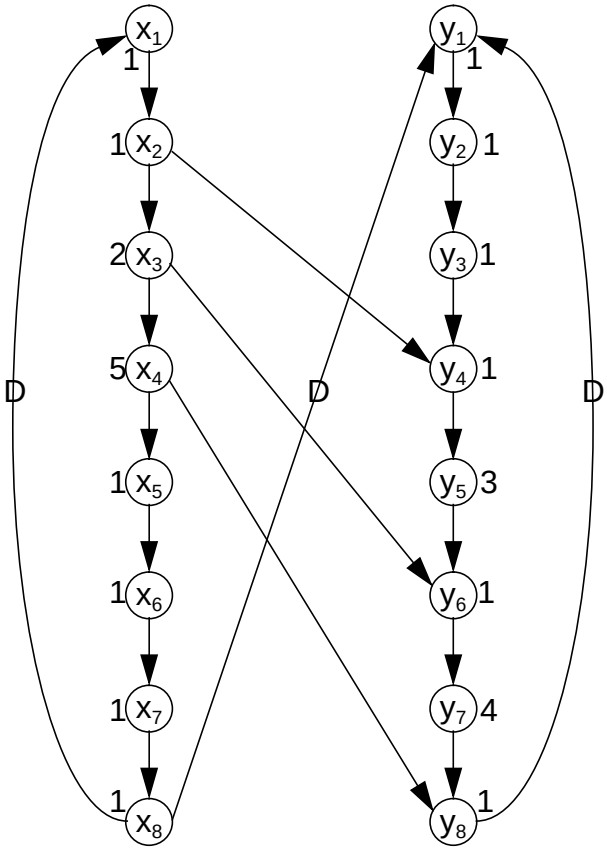


Figure 13. The solution derived by Algorithm A when it is applied to the example of Figure 9.

$$T \equiv \{\chi(v_1, v_2) \mid (((v_1, v_2) \notin E) \textbf{ and } (\rho_G(v_2, v_1) = \infty))\}. \quad (41)$$

After computing the family of subsets specified by (41), Algorithm Global-resynchronize chooses a member of this family that has maximum cardinality, inserts the corresponding delayless resynchronization edge, and removes all synchronization edges that become redundant as a result of inserting this resynchronization edge.

To extend this technique for maximum-throughput resynchronization to the latency-constrained resynchronization problem, we simply replace the subset computation in (41) with

$$T \equiv \{\chi(v_1, v_2) \mid (((v_1, v_2) \notin E) \textbf{ and } (\rho_G(v_2, v_1) = \infty) \textbf{ and } (L'(v_1, v_2) \leq L_{max}))\}, \quad (42)$$

where 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.

A pseudocode specification of our extension of Global-resynchronize to the latency-constrained resynchronization problem, called Algorithm *Global-LCR*, is shown in Figure 14.

9.1 Customization to transparent synchronization graphs

In Section 5, we mentioned that transparent synchronization graphs are advantageous for performing latency-constrained resynchronization. If the input synchronization graph is transparent, then assuming that $T_{\mathit{fi}(G)}(x, y)$ has been determined for all $x, y \in V$, L' in Algorithm Global-LCR can be computed in $O(1)$ time from

$$L'(v_1, v_2) = max(\{(T_{\mathit{fi}(G)}(\upsilon, v_1) + T_{\mathit{fi}(G)}(v_2, o_L)), L_G\}), \quad (43)$$

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

Furthermore, $T_{\mathit{fi}(G)}(x, y)$ can be updated in the same manner as ρ_G. That is once the resynchronization edge, $best$ is chosen, we have that for each $(x, y) \in (V \cup \{\upsilon\})$,

$$T_{new}(x, y) = max(\{T_{\mathit{fi}(G)}(x, y), T_{\mathit{fi}(G)}(x, src(best)) + T_{\mathit{fi}(G)}(snk(best), y)\}), \quad (44)$$

where T_{new} denotes the maximum cumulative execution time between actors in the first iteration

graph after the insertion of the edge $best$ in G. The computations in (44) can be performed by inserting the simple **for** loop shown in Figure 15 at the end of the **else** block in Algorithm Global-LCR. Thus, as with the computation of ρ_G, the cubic-time Bellman-Ford algorithm need only be invoked once, at the beginning of the LCR Algorithm, to initialize $T_{fl(G)}(x, y)$. This loop can be

```
function Global-LCR
input: a reduced synchronization graph G = (V, E)
output: an alternative reduced synchronization graph that preserves G.

compute ρ_G(x, y) for all actor pairs x, y ∈ V
complete = FALSE
while not (complete)
    best = NULL, M = 0
    for x, y ∈ V
        if ((ρ_G(y, x) = ∞) and ((x, y) ∉ E)) and (L'(x, y) ≤ L_max)
            χ* = χ((x, y))
            if (|χ*| > M)
                M = |χ*|
                best = (x, y)
            end if
        end if
    end for
    if (best = NULL)
        complete = TRUE
    else
        E = E − χ(best) + {d_0(best)}
        G = (V, E)
        for x, y ∈ V                    /* update ρ_G */
            ρ_new(x, y) = min({ρ_G(x, y), ρ_G(x, src(best)) + ρ_G(snk(best), y)})
        end for
        ρ_G = ρ_new
    end if
end while
return G
end function
```

Figure 14. A heuristic for latency-constrained resynchronization.

inserted immediately before or after the **for** loop that updates ρ_G.

9.2 Complexity

In the companion paper [5], we showed that Algorithm Global-resynchronize has $O(s_{ff}n^4)$ time-complexity, where n is the number of actors in the input synchronization graph, and s_{ff} is the number of feedforward synchronization edges. Since the longest path quantities $T_{fi(G)}(*, *)$ can be computed initially in $O(n^3)$ time and updated in $O(n^2)$ time, it is easily verified that the $O(s_{ff}n^4)$ bound also applies to our customization of Algorithm Global-LCR to transparent synchronization graphs.

In general, whenever the nested **for** loops in Figure 14 dominate the computation of the **while** loop, the $O(s_{ff}n^4)$ complexity is maintained as long as $(L'(x, y) \leq L_{max})$ can be evaluated in $O(1)$ time. For general (not necessarily transparent) synchronization graphs, we can use the functional simulation approach described in Section 5 to determine $L'(x, y)$ in $O(d \times max(\{n, s\}))$ time, where $d = 1 + \rho_{G_0}(x, y)$, and s denotes the number of synchronization edges in G. This yields a running time of $O(ds_{ff}n^4 max(\{n, s\}))$ for general synchronization graphs.

The complexity bounds derived above are based on a general upper bound of n^2, which is derived in the companion paper [5], on the total number of resynchronization steps (**while** loop iterations). However, this n^2 bound can be viewed as a very conservative estimate since in practice, constraints on the introduction of cycles severely limit the number of possible resynchronization steps [5]. Thus, on practical graphs, we can expect significantly lower average-case complexity than the worst-case bounds of $O(s_{ff}n^4)$ and $O(ds_{ff}n^4 max(\{n, s\}))$.

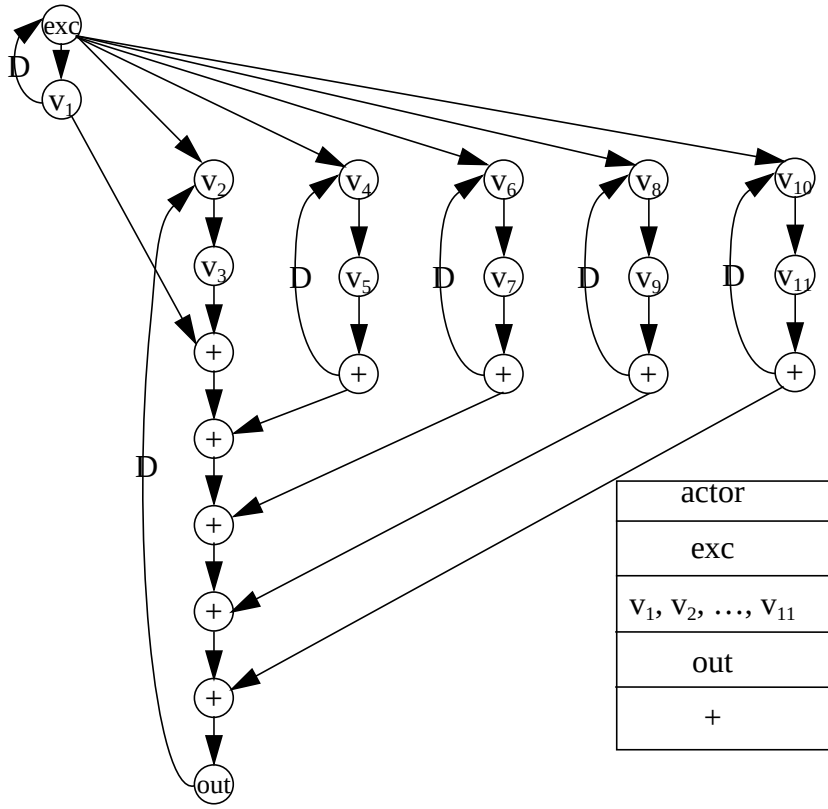
for $x, y \in (V \cup \{\upsilon\})$ /* update $T_{fi(G)}$ */
$T_{new}(x, y) = max(\{T_{fi(G)}(x, y), T_{fi(G)}(x, src(best)) + T_{fi(G)}(snk(best), y)\})$
end for
$T_{fi(G)} = T_{new}$

Figure 15. Pseudocode to update $T_{fi(G)}$ for use in the customization of Algorithm Global-LCR to transparent synchronization graphs.

9.3 Example

Figure 16 shows the synchronization graph that results from a six-processor schedule of a synthesizer for plucked-string musical instruments in 11 voices based on the Karplus-Strong technique, as shown in the companion paper [5]. 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 none of these are redundant.

Figure 17 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 synchronization 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



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

Figure 16. 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

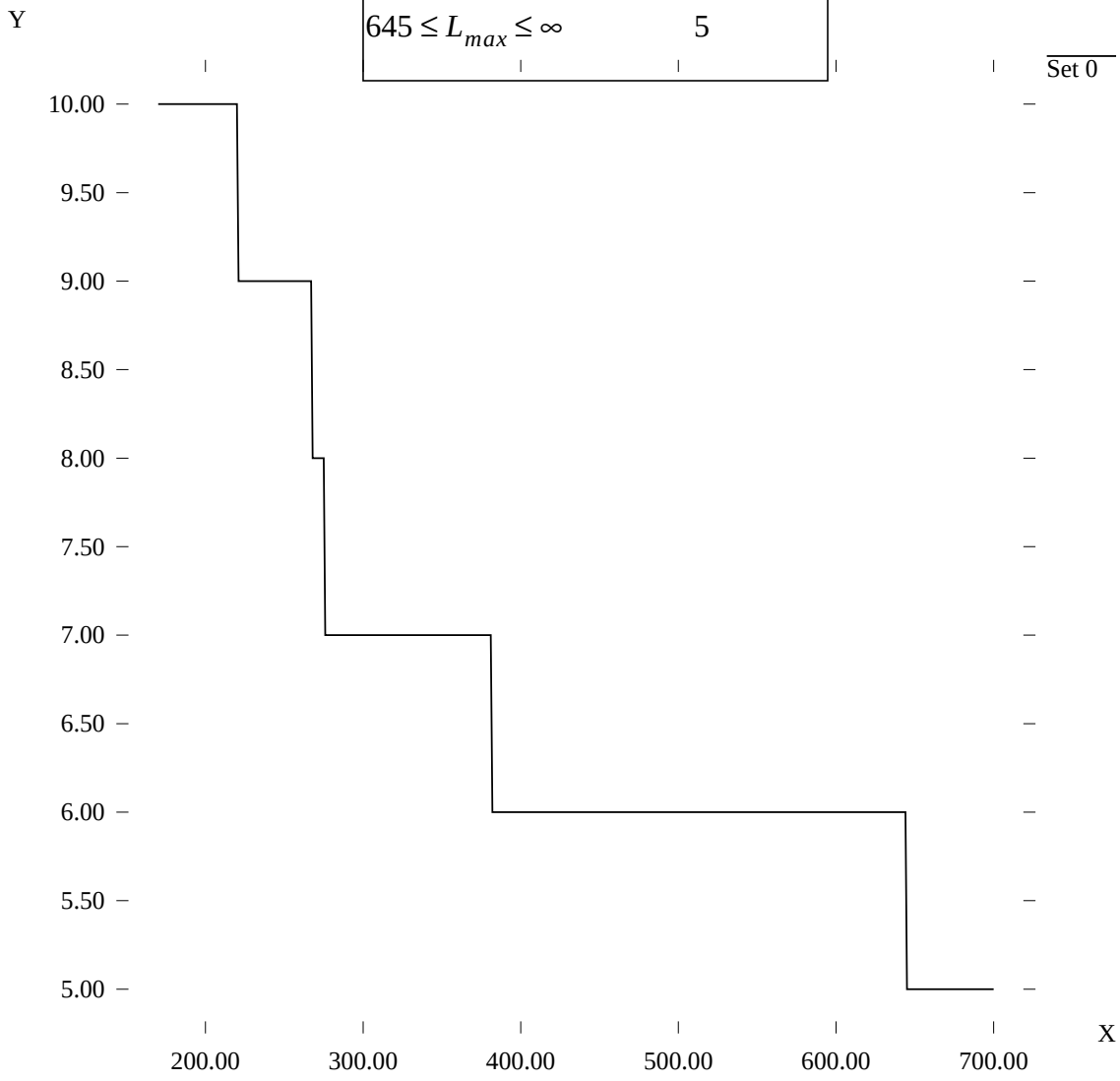


Figure 17. Performance of the heuristic on the example of Figure 16.

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 minimal cost of 5 synchronization edges for this system is attained, which is half that of the original synchronization graph.

Figure 18 and Table 1 show how the average iteration period (the reciprocal of the aver-

Table 1. Performance results for the resynchronization of Figure 16. The first column gives the memory access time; "IP" stands for "average iteration period" (the reciprocal of the average throughput); and "A/P" stands for "memory accesses per graph iteration."

Mem Acc Time	A		F		B		C		D		E	
	IP	A/P	IP	A/P	IP	A/P	IP	A/P	IP	A/P	IP	A/P
0	210	66	184	47	219	59	188	60	200	50	186	47
1	250	67	195	43	274	58	225	58	222	50	222	47
2	292	66	216	43	302	58	262	52	259	50	248	46
3	335	64	249	43	334	58	294	54	298	50	288	45
4	368	63	273	40	373	59	333	53	338	48	321	46
5	408	63	318	43	413	58	375	53	375	49	357	47
6	459	63	350	43	457	58	396	53	419	50	396	47
7	496	63	385	43	502	58	442	53	461	51	431	47
8	540	63	420	43	553	59	480	54	490	50	474	47
9	584	63	455	43	592	58	523	53	528	50	509	47
10	655	65	496	43	641	62	554	54	573	51	551	47

age throughput) varies with different memory access times for various resynchronizations of Figure 16. Here, the column of Table 1 and the plot of Figure 18 labeled A represent the original synchronization graph (before resynchronization); column/plot label B represents the resynchronized result corresponding to the first break-point of Figure 17 ($L_{max} = 221$, 9 synchronization edges); label C corresponds to the second break-point of Figure 17 ($L_{max} = 268$, 8 synchronization edges); and so on for labels D, E and F, whose associated synchronization graphs have 7, 6 and 5 synchronization edges, respectively. Thus, as we go from label A to label F, the number

of synchronization edges in resynchronized solution decreases monotonically. However, as seen in Figure 18, the average iteration period need not exactly follow this trend. For example, even though synchronization graph A has one synchronization edge more than graph B, the iteration period curve for graph B lies slightly above that of A. This is because the simulations shown in the figure model a shared bus, and take bus contention into account. Thus, even though graph B has one less synchronization edge than graph A, it entails higher bus contention, and hence results in a higher average iteration period. A similar anomaly is seen between graph C and graph D, where graph D has one less synchronization edge than graph C, but still has a higher average iteration period. However, we observe such anomalies only within highly localized neighborhoods in which the number of synchronization edges differs by only one. Overall, in a global sense, the figure shows a clear trend of decreasing iteration period with loosening of the latency constraint, and reduction of the number of synchronization edges.

It is difficult to model bus contention analytically, and for precise performance data we must resort to a detailed simulation of the shared bus system. We propose using such a simulation

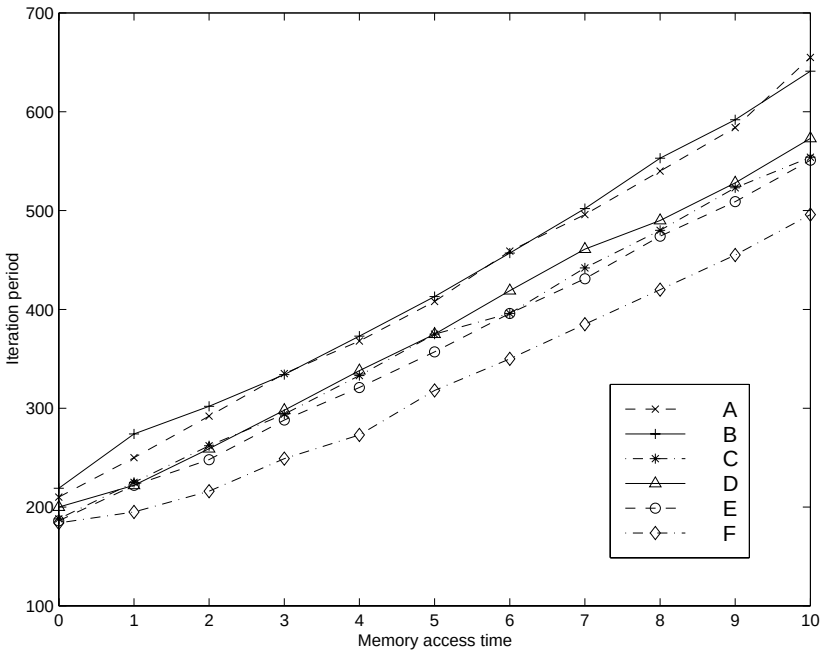


Figure 18. Average iteration period (reciprocal of average throughput) vs. memory access time for various latency-constrained resynchronizations of the music synthesis example in Figure 16.

as a means of verifying that the resynchronization optimization does not result in a performance degradation due to higher bus contention. Our experiments suggest that this needs to be done only for cases where the number of synchronization edges removed by resynchronization is small compared to the total number of synchronization edges (i.e. when the resynchronized solution is within a localized neighborhood of the original synchronization graph).

Figure 19 shows that the average number of shared memory accesses per graph iteration decreases consistently with loosening of the latency constraint. As mentioned in the companion paper, such reduction in shared memory accesses is relevant when power consumption is an important issue, since accesses to shared memory often require significant amounts of energy.

Figure 20 illustrates how the placement of synchronization edges changes as the heuristic is able to attain lower synchronization costs.

Note that synchronization graphs computed by the heuristic are not necessarily identical over any of the L_{max} ranges in Figure 17 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 (that is, no resynchronization candidates for which $(|\chi(*)| > 1)$), the heuristic attempts to insert resynchroni-

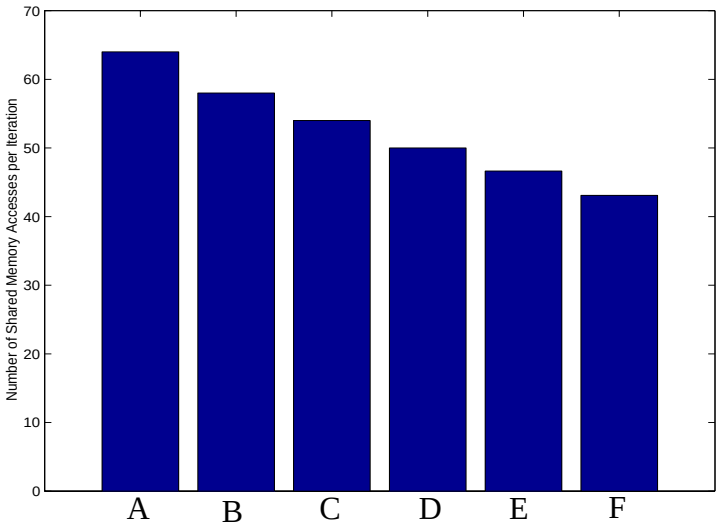


Figure 19. Average number of shared memory accesses per iteration for various latency-constrained resynchronizations of the music synthesis example.

zation edges for the purpose of increasing the connectivity; this increases the chance that subsequent resynchronization candidates will be generated for which $|\chi(*)| > 1$ [5]. For example, Figure 21, shows the synchronization graph computed when L_{max} is just below the amount needed to permit the minimal solution, which requires only five synchronization edges (solution F). Comparison with the graph shown in Figure 20(d) shows that even though these solutions

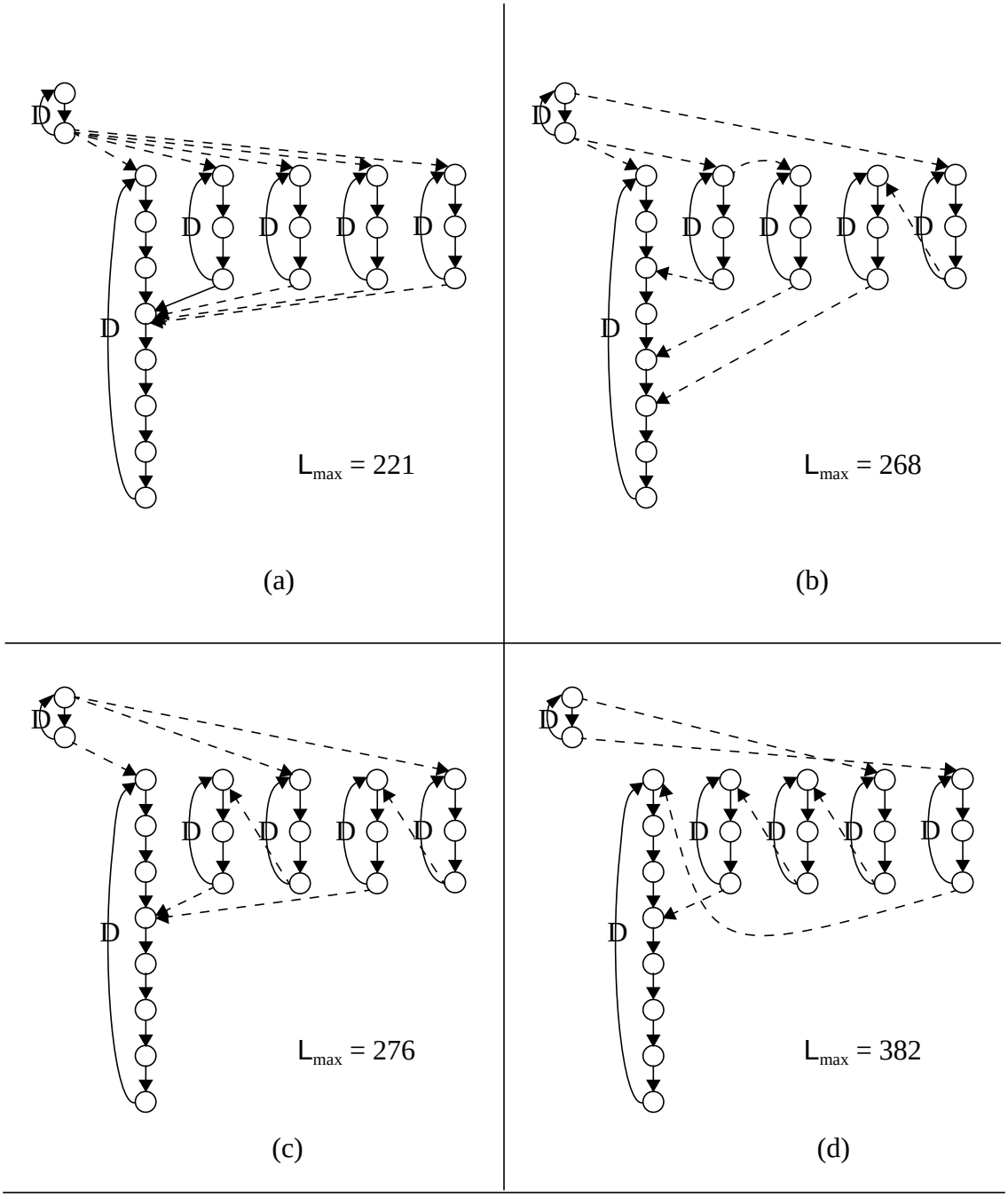


Figure 20. Synchronization graphs computed by the heuristic for different values of L_{max}.

have the same synchronization cost, the heuristic had much more room to pursue further resynchronization opportunities with $L_{max} = 644$, and thus, the graph of Figure 21 is more similar to the minimal solution than it is to the solution of Figure 20(d).

Earlier, we mentioned that our $O(s_{ff}n^4)$ and $O(ds_{ff}n^4 max(\{n, s\}))$ complexity expressions are conservative since they are based on an n^2 bound on the number of iterations of the while loop in Figure 14, while in practice, the actual number of while loop iterations can be expected to be much less than n^2. This claim is supported by our music synthesis example, as shown in the graph of Figure 22. Here, the X-axis corresponds 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 $n^2 = 484$, it is even less than n. This suggests that perhaps a significantly tighter bound on the number of while loop iterations can be derived.

10. Conclusions

This paper has addressed the problem of latency-constrained resynchronization for self-timed implementation of iterative dataflow specifications.

Given an upper bound L_{max} on the allowable latency, the objective of latency-constrained

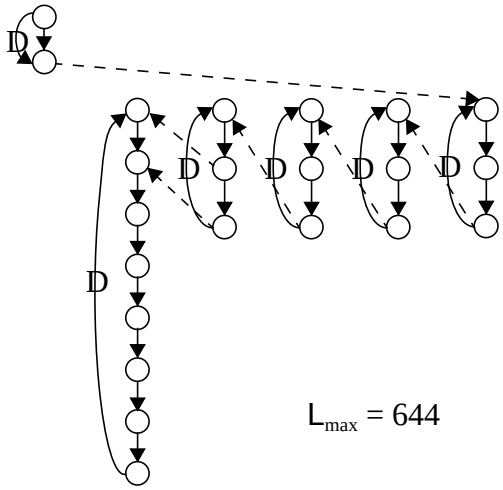


Figure 21. The synchronization graph computed by the heuristic for $L_{max} = 644$.

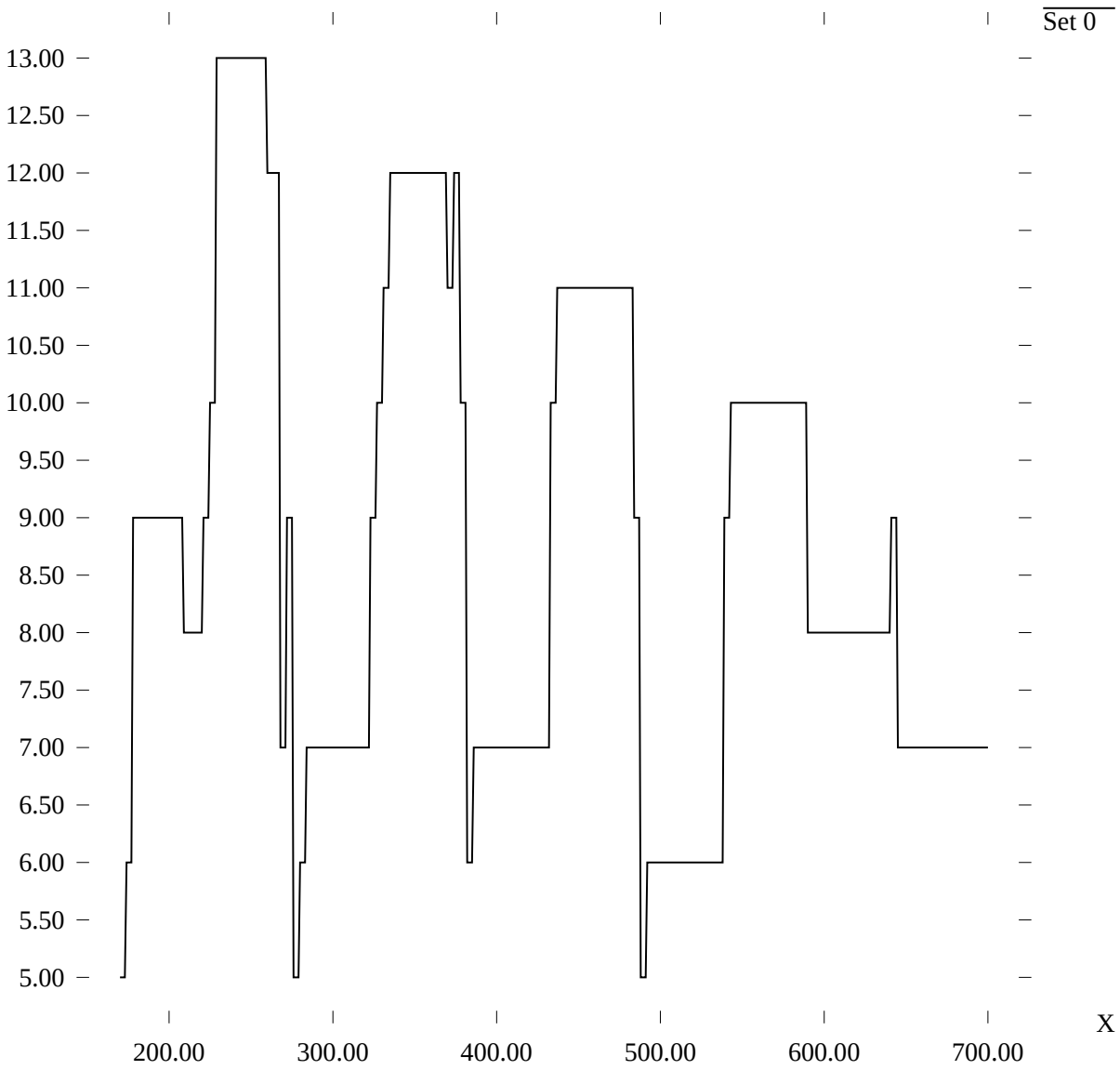


Figure 22. Number of resynchronization iterations versus L_{max} for the example of Figure 16.

resynchronization is to insert extraneous synchronization operations in such a way that a) the number of original synchronizations that consequently become redundant significant exceeds the number of new synchronizations, and b) the serialization imposed by the new synchronizations does not increase the latency beyond L_{max}. To ensure that the serialization imposed by resynchronization does not degrade the throughput, the new synchronizations are restricted to lie outside of all cycles in the final synchronization graph.

We have established that optimal latency-constrained resynchronization is NP-hard even for a very restricted class of synchronization graphs; we have derived an efficient, polynomial-time algorithm that computes optimal latency-constrained resynchronizations for two-processor systems; and we have extended the heuristic presented in the companion paper [5] for maximum-throughput resynchronization to address the problem of latency-constrained resynchronization for general *n*-processor systems. Through an example of a music synthesis system, we have illustrated the ability of this extended heuristic to systematically trade-off between synchronization overhead and latency.

The techniques developed in this paper and the companion paper [5] can be used as a post-processing step to improve the performance of any of the large number of static multiprocessor scheduling techniques for iterative dataflow specifications, such as those described in [1, 2, 8, 11, 12, 19, 24, 27, 31, 33].

11. 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.
$Delay(p)$:	The sum of the edge delays over all edges in the path p.
$d_n(u, v)$:	An edge whose source and sink vertices are u and v, respectively, and whose delay is equal to n.

$\chi(p)$: The set of synchronization edges that are subsumed by the ordered pair of actors p.

2LCR: Two-processor latency-constrained resynchronization.

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 $\Psi(R, G)$ such that p contains s' and $Delay(p) \leq delay(s)$, then we say that s' contributes to the elimination of s.

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 $s \notin R$.

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.

estimated throughput:
The maximum over all cycles C in a DFG of $Delay(C)/T$, where T is the sum of the execution times of all vertices traversed by C.

FBS: Feedback synchronization. A synchronization protocol that may be used for feedback edges in a synchronization graph.

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 in a synchronization graph.

LCR: Latency-constrained resynchronization. Given a synchronization graph G, a resynchronization R of G is an LCR if the latency of $\Psi(R, G)$ is less than or equal to the latency constraint L_{max}.

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.

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

SCC: Strongly connected component.

self loop: An edge whose source and sink vertices are identical.

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.

12. 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. "Optimizing synchronization in multiprocessor DSP systems." *IEEE Transactions on Signal Processing*, vol. 45, no. 6, June 1997.

[5] S. S. Bhattacharyya, S. Sriram, and E. A. Lee, *Resynchronization for Multiprocessor DSP Implementation — Part 1: Maximum-throughput Resynchronization*, Digital Signal Processing Laboratory, University of Maryland at College Park, July, 1998.

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

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

[8] L-F. Chao and E. H-M. Sha, "Static Scheduling for Synthesis of DSP Algorithms on Various Models," *Journal of VLSI Signal Processing*, pp. 207-223, 1995.

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

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

[11] 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.

[12] 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.

[13] T. C. Hu, "Parallel Sequencing and Assembly Line Problems," *Operations Research*, Vol. 9, 1961.

[14] S. Y. Kung. *VLSI Array processors*. Prentice Hall, 1988.

[15] 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.

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

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

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

[19] 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, 1993.

[20] V. Madisetti. *VLSI Digital Signal Processors*. IEEE Press, 1995.

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

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

[23] K. K. Parhi, "High-Level Algorithm and Architecture Transformations for DSP Synthesis," *Journal of VLSI Signal Processing*, January, 1995.

[24] 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.

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

[26] M. Potkonjac and M. B. Srivastava. "Behavioral synthesis of high performance, and low power application specific processors for linear computations." In *Proceedings of the International Conference on Application Specific Array Processors*, 1994, pp. 45–56.

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

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

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

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

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

ing, 1990.

[32] J. Teich, L. Thiele, and E. A. Lee. “Modeling and simulation of heterogeneous real-time systems based on a deterministic discrete event model.” In *Proceedings of the International Symposium on Systems Synthesis*, 1995, pp. 156–161.

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