Memorandum UCB/ERL M96/56, Electronics Research Laboratory, U. C. Berkeley, October, 1996.

RESYNCHRONIZATION OF MULTIPROCESSOR SCHEDULES: PART 2 — LATENCY-CONSTRAINED RESYNCHRONIZATION

Shuvra S. Bhattacharyya, Hitachi America
Sundararajan Sriram, Texas Instruments
Edward A. Lee, UC Berkeley

1. Abstract

The companion paper [7] 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 [5, 32]. 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 [7] develops fundamental properties of resynchronization and studies the problem of optimal resynchronization under the assumption that arbitrary increases in latency can be tolerated (“unbounded-latency 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}. Our study is based in the context of self-timed execution of iterative dataflow programs, which is an implementation model that has been applied extensively for digital signal processing systems.

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 [5, 32].

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. 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}. We address this problem in the context of *iterative synchronous dataflow* [21] programs. An iterative dataflow program consists of a dataflow representation of the body of a loop that is to be iterated infinitely. Iterative synchronous dataflow programming is used extensively for the implementation of digital signal processing (**DSP**) systems. Examples of commercial design environments that use this model are COSSAP by Synopsys, the Signal Processing Worksystem (SPW) by Cadence, and Virtuoso Synchro by Eonic Systems. Examples of research tools developed at various universities that use synchronous dataflow or closely related dataflow models are DESCARTES [31], GRAPE [19], Ptolemy, [11] and the Warp compiler [29].

Our implementation model involves a *self-timed* scheduling strategy [22]. Each processor executes the tasks assigned to it in a fixed order that is specified at compile time. Before firing an actor, a processor waits for the data needed by that actor to become available. Thus, processors are required to perform run-time synchronization only when they communicate data. This provides robustness when the execution times of tasks are not known precisely or when then they may exhibit occasional deviations from their estimates.

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. Details on the assumed synchronization protocols are given in Section 3.1. Thus, in our context, resynchronization achieves its benefit by reducing the rate of accesses to shared memory for the purpose of synchronization.

3. Background on synchronization optimization for iterative dataflow graphs

In **synchronous dataflow** (**SDF**), a program is represented as a directed graph in which vertices (**actors**) represent computational tasks, edges specify data dependences, and the number of data values (**tokens**) produced and consumed by each actor is fixed. Actors can be of arbitrary complexity. In DSP design environments, they typically range in complexity from basic operations such as addition or subtraction to signal processing subsystems such as FFT units and adaptive filters.

Delays on SDF edges represent initial tokens, and specify dependencies between iterations of the actors in iterative execution. For example, if tokens produced by the kth invocation of actor A are consumed by the $(k+2)$th invocation of actor B, then the edge (A, B) contains two delays. We assume that the input SDF graph is *homogeneous*, which means that the numbers of tokens produced and consumed are identically unity. However, since efficient techniques have been developed to convert general SDF graphs into homogeneous graphs [21], our techniques can easily be adapted to general SDF graphs. We refer to a homogeneous SDF graph as a **DFG**. The techniques developed in this paper can be used as a post-processing step to improve the performance of any static multiprocessor scheduling technique for iterative DFGs, such as those described in [1, 2, 12, 16, 17, 23, 27, 29, 33, 34].

A **path** in a DFG is a finite, nonempty sequence $(e_1, e_2, \ldots, e_n)$, where

$$snk(e_1) = src(e_2),\ snk(e_2) = src(e_3),\ \ldots,\ snk(e_{n-1}) = src(e_n).$$

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

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

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

A strongly connected component (**SCC**) of a directed graph is a maximal subgraph in which there is a path from each vertex to every other vertex. A **feedforward edge** is an edge that is not contained in an SCC, and a **feedback edge** is an edge that is contained in at least one SCC. The source and sink actors of an SDF edge e are denoted $src(e)$ and $snk(e)$, and the delay on e is denoted $delay(e)$. An edge e is a **self loop edge** if $src(e) = snk(e)$. We define $d_n(x, y)$ to represent an SDF edge whose source and sink vertices are x and y, respectively, and whose delay is n (assumed non-negative).

An SDF representation of an application is called an **application DFG**. For each task v in a given application DFG G, we assume that an estimate $t(v)$ (a positive integer) of the execution time is available. Given a multiprocessor schedule for G, we derive a data structure called the **IPC graph**, denoted G_{ipc}, by instantiating a vertex for each task, connecting an edge from each task to the task that succeeds it on the same processor, and adding an edge that has unit delay from the last task on each processor to the first task on the same processor. Also, for each edge (x, y) in G that connects tasks that execute on different processors, an *IPC edge* having identical delay is instantiated in G_{ipc} from x to y. Figure 1(c) shows the IPC graph that corresponds to the application DFG of Figure 1(a) and the processor assignment / actor ordering of Figure 1(b).

Each edge (v_j, v_i) in G_{ipc} represents the **synchronization constraint**

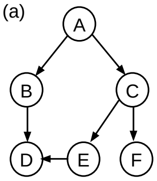
$$start(v_i, k) \ge end(v_j, k - delay((v_j, v_i))) , \qquad (2)$$

where $start(v, k)$ and $end(v, k)$ respectively represent the time at which invocation k of actor v begins execution and completes execution.

Initially, an IPC edge in G_{ipc} represents two functions — reading and writing of tokens into the corresponding buffer, and synchronization between the sender and the receiver. To differentiate these functions, we define another graph called the **synchronization graph**, in which edges between tasks assigned to different processors, called **synchronization edges**, represent *synchronization constraints only.* An **execution source** of a synchronization graph is any actor that has nonzero delay on each input edge.

Initially, the synchronization graph is identical to G_{ipc}. However, resynchronization modifies the synchronization graph by adding and deleting synchronization edges. After resynchronization, the IPC edges in G_{ipc} represent buffer activity, and must be implemented as buffers in shared memory, whereas the synchronization edges represent synchronization constraints, and are implemented by updating and testing flags in shared memory as described in Section 3.1 below. If there is an IPC edge as well as a synchronization edge between the same pair of actors, then the synchronization protocol is executed before the buffer corresponding to the IPC edge is accessed so as to ensure sender-receiver synchronization. On the other hand, if there is an IPC edge between two actors in the IPC graph, but there is no synchronization edge between the two, then no synchronization needs to be done before accessing the shared buffer. If there is a synchronization edge between two actors but no IPC edge, then no shared buffer is allocated between the two actors; only the corresponding synchronization protocol is invoked.

(a)



(b)

Processor	Actor ordering
Proc. 1	B, D, F
Proc. 2	A, C, E

(c)

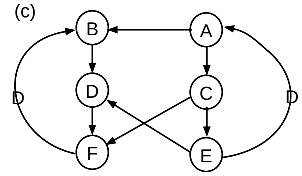


Figure 1. Part (c) shows the IPC graph that corresponds to the DFG of part (a) and the processor assignment / actor ordering of part (b). A “D” on top of an edge represents a unit delay.

3.1 Synchronization protocols

Given a synchronization graph (V, E), and a synchronization edge $e \in E$, if e is a feedforward edge then we apply a synchronization protocol called **feedforward synchronization (FFS)**, which guarantees that $snk(e)$ never attempts to read data from an empty buffer (to prevent underflow), and $src(e)$ never attempts to write data into the buffer unless the number of tokens already in the buffer is less than some pre-specified limit, which is the amount of memory allocated to that buffer (to prevent overflow). This involves maintaining a count of the number of tokens currently in the buffer in a shared memory location. This count must be examined and updated by each invocation of $src(e)$ and $snk(e)$.

If e is a feedback edge, then we use a more efficient protocol, called **feedback synchronization (FBS)**, that only explicitly ensures that underflow does not occur. Such a simplified protocol is possible because each feedback edge has a buffer requirement that is bounded by a constant, called the **self-timed buffer bound** of the edge. The self-timed buffer bound of a feedback edge can be computed efficiently from the synchronization graph topology [5]. In the FBS protocol we allocate a shared memory buffer of size equal to the self-timed buffer bound of e, and rather than maintaining the token count in shared memory, we maintain a copy of the *write pointer* into the buffer (of the source actor). After each invocation of $src(e)$, the write pointer is updated locally (on the processor that executes $src(e)$), and the new value is written to shared memory. It is easily verified that to prevent underflow, it suffices to block each invocation of the sink actor until the *read pointer* (maintained locally on the processor that executes $snk(e)$) is found to be not equal to the current value of the write pointer. For a more detailed discussion of the FFS and FBS protocols, the reader is referred to [9].

3.2 Estimated throughput

If the execution time of each actor v is a fixed constant $t^*(v)$ for all invocations of v, and the time required for IPC is ignored (assumed to be zero), then as a consequence of Reiter's analysis in [30], the throughput (number of DFG iterations per unit time) of a synchronization graph

G is given by

$$\tau = \underset{\text{cycle } C \text{ in } G}{min}\left\{\frac{Delay(C)}{\sum_{v \in C} t^*(v)}\right\}. \tag{3}$$

Since in our problem context, we only have execution time estimates available instead of exact values, we replace $t^*(v)$ with the corresponding estimate $t(v)$ in (3) to obtain the **estimated throughput**. The objective of resynchronization is to increase the *actual throughput* by reducing the rate at which synchronization operations must be performed, while making sure that the estimated throughput is not degraded.

3.3 Synchronization redundancy and resynchronization

Any transformation that we perform on the synchronization graph must respect the synchronization constraints implied by G_{ipc}. If we ensure this, then we only need to implement the synchronization edges of the optimized synchronization graph. 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 $\rho_{G_1}(src(e), snk(e)) \le delay(e)$. The following theorem, developed in [9], underlies the validity of resynchronization.

Theorem 1: The synchronization constraints (as specified by (2)) of G_1 imply the constraints of G_2 if G_1 preserves G_2.

Intuitively, Theorem 1 is true because if G_1 preserves G_2, then for every synchronization edge e in G_2, there is a path in G_1 that enforces the synchronization constraint specified by e.

A synchronization edge is **redundant** in a synchronization graph G if its removal yields a graph that preserves G. For example, in Figure 1(c), the synchronization edge (C, F) is redundant due to the path $((C, E), (E, D), (D, F))$. In [5], it is shown that if all redundant edges in a synchronization graph are removed, then the resulting graph preserves the original synchronization graph.

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

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

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 $\Psi(R, G)$.

For example $R = \{(E, B)\}$ is a resynchronization of the synchronization graph shown in Figure 1(c).

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. Rearrangement of feedforward edges does not reduce the estimated throughput because such edges do not affect the value derived from (3).

We will use the following lemma, which is established [7].

Lemma 1: Suppose that G is a synchronization graph; R is a resynchronization of G; and (x, y) is a resynchronization edge such that $\rho_G(x, y) = 0$. Then (x, y) is redundant in $\Psi(R, G)$. Thus, a minimal resynchronization (fewest number of elements) has the property that $\rho_G(x', y') > 0$ for each resynchronization edge (x', y').

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: Suppose that G is a synchronization graph and R is a resynchronization of G. If s is a synchronization edge in G that is not contained in R, we say that R **eliminates** s. If 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 $Delay(p) \le delay(s)$, 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 $Delay(p) \le delay(s)$. 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 the our choice of terminology in Definition 1. An example is shown in Figure 2.

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.

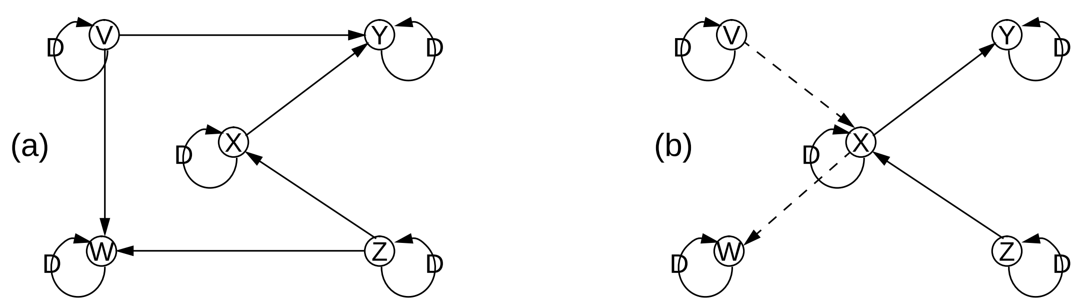


Figure 2. An illustration of Definition 1. Here each processor executes a single actor. A resynchronization of the synchronization graph in (a) is illustrated in (b). In this resynchronization, the resynchronization edges (V, X) and (X, W) both contribute to the elimination of (V, W).

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 3(a). Figure 3(b) shows a resynchronization R of G. In the resynchronized graph of Figure 3(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 3(c), is also a resynchronization of G. In Figure 3(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 3(d), is a also resynchronization of G.

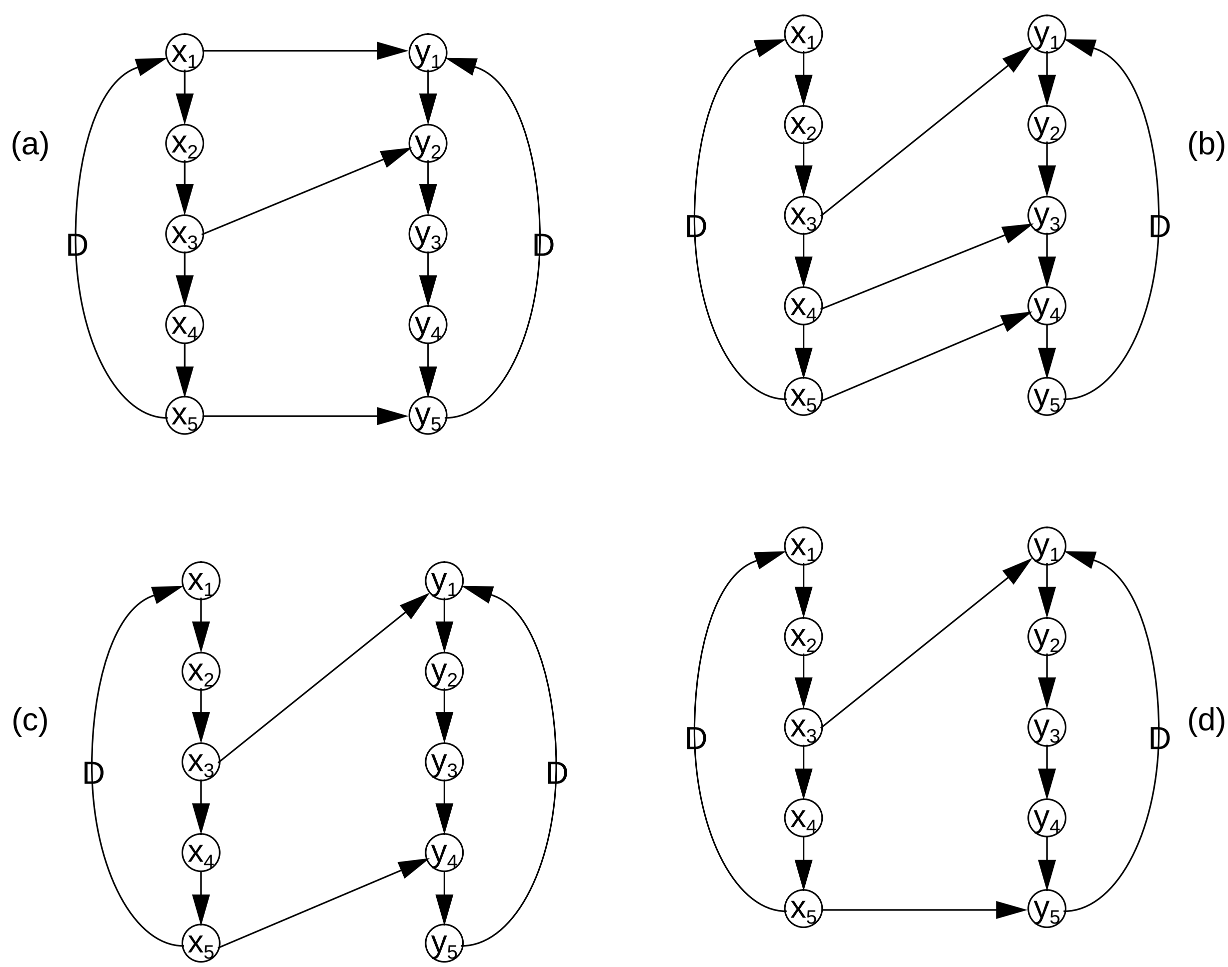


Figure 3. Properties of resynchronization.

5. Latency-constrained resynchronization

As discussed in Section 3.3, 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 in G, and y is an actor in G other than x, then we define the **latency** from x to y by $L_G(x, y) \equiv end(y, 1 + \rho_{G_0}(x, y))$ [1]. We refer to x as the **latency input** associated with this measure of latency, and we refer to y as the **latency output**.

Intuitively, the latency is the time required for the first invocation of the latency input to influence the associated latency output. Thus, our measure of latency is explicitly concerned only with the time that it takes for the *first* input to propagate to the output, and does not in general give an upper bound on the time for subsequent inputs to influence subsequent outputs. Extending our latency measure to maximize over all pairs of “related” input and output invocations would yield the alternative measure L_G' defined by

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

Currently, there are no known tight upper bounds on L_G' that can be computed efficiently from the synchronization graph for any useful subclass of graphs, and thus, we use the lower bound approximation L_G, which corresponds to the critical path, when attempting to analyze and optimize the input-output propagation delay of a self-timed system. The heuristic that we present in Section 9 for latency-constrained resynchronization can easily be adapted to handle arbitrary

---

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

latency measures; however, the efficiency of the heuristic depends on the existence of an algorithm to efficiently compute the change in latency that arises from inserting a single new synchronization edge. The exploration of incorporating alternative measures — or estimates — of latency in this heuristic framework, would be a useful area for further study.

In our study of latency-constrained resynchronization, we restrict our attention to a class of synchronization graphs for which the latency can be computed efficiently. 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 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.

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 $fi(G)$, 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 4 illustrates the derivation of $fi(G)$.

Given two vertices x and y in $fi(G)$ such that there is a path in $fi(G)$ from x to y, we

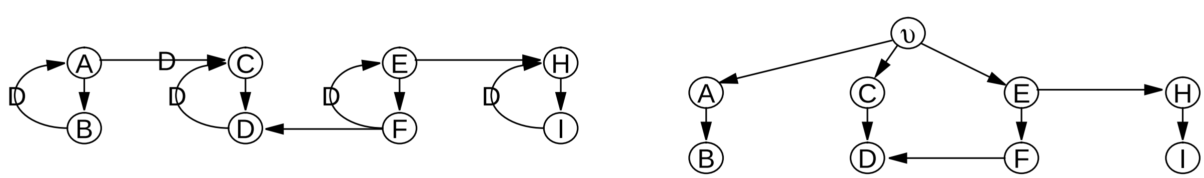


Figure 4. An example used to illustrate the construction of $fi(G)$. The graph on the right is $fi(G)$ when G is the left-side graph.

denote the sum of the execution times along a path from x to y that has maximum cumulative execution time by $T_{fi(G)}(x, y)$. That is,

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

If there is no path from x to y, then we define $T_{fi(G)}(x, y)$ to be $-\infty$. Note that for all x, y $T_{fi(G)}(x, y) < +\infty$, since $fi(G)$ is acyclic. The values $T_{fi(G)}(x, y)$ for all pairs x, y can be computed in $O(n^3)$ time, where n is the number of actors in G, by using a simple adaptation of the Floyd-Warshall algorithm specified in [14].

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 $L_G(x, y) = T_{fi(G)}(\upsilon, y)$.

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

$$end(w, 1) = T_{fi(G)}(\upsilon, w), \qquad (6)$$

which clearly implies the desired result.

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

Now suppose that (6) holds whenever $mt(w) \le k$, for some $k \ge 1$, and consider the scenario $mt(w) = k + 1$. 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 $mt(z) \le k$, 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

$$end(w, 1) = max(T_{fi(G)}(\upsilon, z) | (z \in P_w)) + t(w). \quad (7)$$

But, by definition of $T_{fi(G)}$, the RHS of (7) 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 (6) holds for $mt(w) = 1$, and that whenever it holds for $mt(w) = k \geq 1$, it must hold for $mt(w) = (k+1)$. Thus, (6) holds for all values of $mt(w)$. *QED*✎

Theorem 2 shows that latency can be computed efficiently for transparent synchronization graphs. A further benefit of transparent synchronization graphs is that the change in latency induced by adding a new synchronization edge (a “resynchronization operation”) can be computed in $O(1)$ time, given $T_{fi(G)}(a, b)$ for all actor pairs (a, b). We will discuss this further, as well as its application to developing an efficient resynchronization heuristic, in Section 9.

Definition 4: An instance of the **latency-constrained synchronization problem** consists of a transparent synchronization graph G with latency input x and latency output y, and a *latency constraint* $L_{max} \geq L_G(x, y)$. A solution to such an instance is a resynchronization R 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 transparent synchronization graph G with latency input x and latency output y, and a latency constraint L_{max}, we say that a resynchronization R of G is a **latency-constrained resynchronization (LCR)** if $L_{\Psi(R, G)}(x, y) \leq L_{max}$. Thus, the latency-constrained resynchronization problem is the problem of determining a minimal LCR.

6. Related work

In [5], a simple, efficient algorithm, called *Convert-to-SC-graph*, is described for introducing new synchronization edges so that the synchronization graph becomes strongly connected, which allows all synchronization edges to be implemented with 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 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.

Shaffer presented an algorithm that removes redundant synchronizations in the self-timed execution of a non-iterative DFG [32]. This technique was subsequently extended to handle iterative execution and DFG edges that have delay [5]. These approaches differ from the techniques of this paper in that they only consider the redundancy induced by the *original* synchronizations; they do not consider the addition of new synchronizations.

Resynchronization has been studied earlier in the context of hardware synthesis [15]. However in this work, the scheduling model and implementation model are significantly different from the structure of self-timed multiprocessor implementations, and as a consequence, the analysis techniques and algorithmic solutions do not apply to our context, and vice-versa [7].

Partial summaries of the material in this paper and the companion paper have been presented in [4] and [8], respectively.

7. Intractability of LCR

In this section we show that 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 unbounded-latency resynchronization problem [7], 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 5 illustrates the instance of latency-constrained resynchronization that we

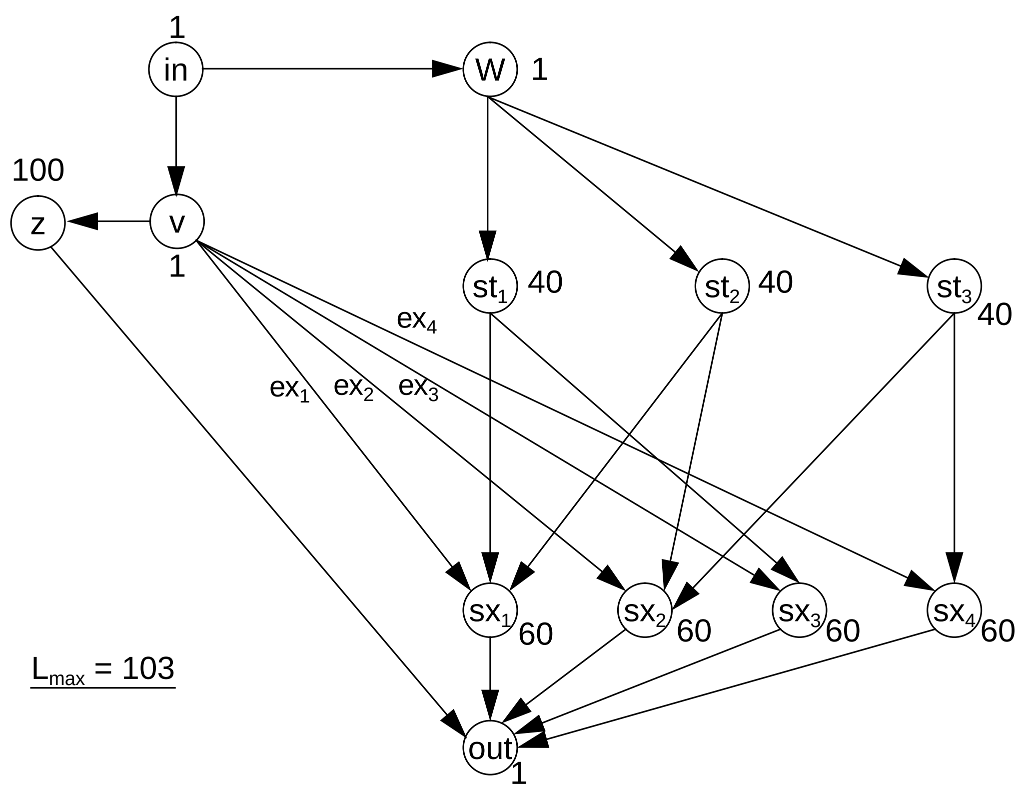


Figure 5. An instance of latency-constrained resynchronization that is derived from an instance of the set covering problem.

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 5, 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 corresponds to the solution of (X, T) that consists of the subfamily $\{t_1, t_3\}$.

A correspondence between the set covering instance (X, T) and the instance of latency-

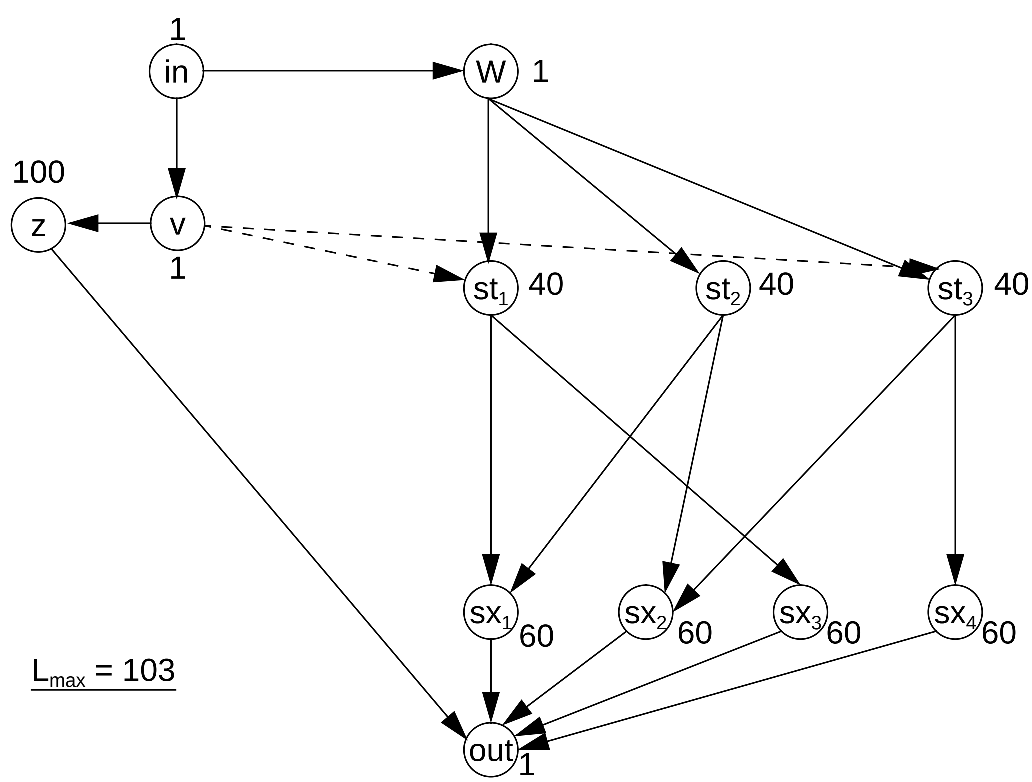


Figure 6. The synchronization graph that results from a solution to the instance of latency-constrained resynchronization shown in Figure 5.

constrained resynchronization defined by Figure 5 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

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

The first observation is immediately apparent from inspection of Figure 5.

Proof of Observation 2 We must show that no other resynchronization edges can be contained in an optimal LCR of G. Figure 7 specifies arguments with which we can discard all possibilities other than those given in (8). In the matrix shown in Figure 7, 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 5. A proof of argument #6 can be found in Appendix A of [6].

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

	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 1).
5. Introduces a delayless self loop.
6. See Appendix A in [6].

Figure 7. Arguments that support Observation 2.

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 1 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 7 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 7 marked *OK* are simply those that correspond to (8), and thus we have justified Observation 2. *QED*✎

The following observation, which is proven in Appendix B of [6], 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.

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

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

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

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

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

It is easily verified that the set of synchronization edges eliminated by R'' is $\{ex_i | x_i \in X\}$. 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 5) 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 8.

The main task in establishing our general correspondence between latency-constrained resynchronization and set covering is generalizing Observation 2 to apply to all constructions that follow the steps in Figure 8. This generalization is not conceptually difficult (although it is rather tedious) since it is easily verified that all of the arguments in Figure 8 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 [13, 18]. 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 [24].

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

- 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 5 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 8. 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}.

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 $L_{\Psi(R, G)}(x_1, y_q) \le L_{max}$. 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 $\tilde{G}$. An extension of this algorithm to the general 2LCR problem (arbitrary delays can be present) can be found in [6].

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 $\{w_a, w_{a+1}, \ldots, w_b\}, 1 \le a \le b \le N$. 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 $\{w_{a_2}, w_{a_2+1}, \ldots, w_{b_2}\}$, $a_2 \le b_1 + 1$, 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 $\{w_{a_3}, w_{a_3+1}, \ldots, w_{b_3}\}$, $a_3 \le b_2 + 1$, where

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

and so on until $b_n = N$.

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

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

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

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

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

Consider the example shown in Figure 9. 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 synchronization edge in R such that $(x_a, y_b) \neq v_a$. Then $(R - \{(x_a, y_b)\} + \{d_0(v_a)\})$ is an LCR of R.

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

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

From Observation 4 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

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

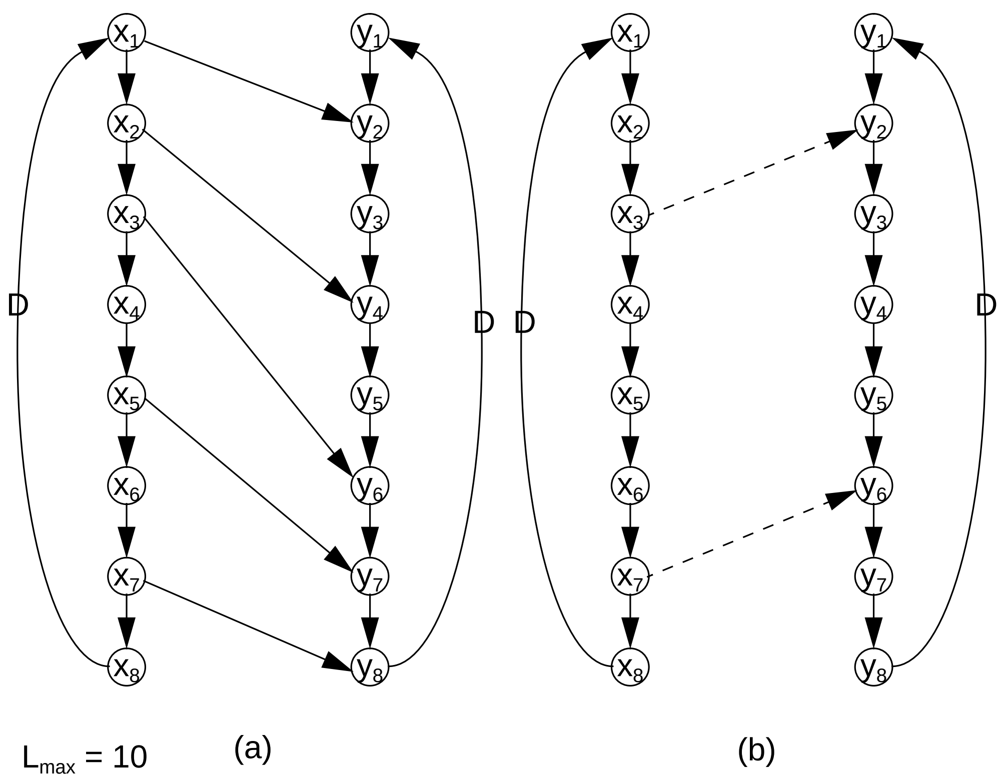


Figure 9. An instance of delayless, two-processor latency-constrained resynchronization. In this example, the execution times of all actors are identically equal to unity.

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

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

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 $t_{pred}(x_a) + t_{succ}(y_c) \le L_{max}$, 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 9(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 $R = \{v_j | \chi(v_j) \in C\}$.

The time-complexity of this optimal algorithm for delayless 2LCR is $O(N^2)$, where N is the number of vertices in $\tilde{G}$ [6].

From (19), we see that there are two possible solutions that can result if we apply Steps (a)-(e) to Figure 9(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 9(b).

9. A heuristic for general transparent synchronization graphs

The companion paper [7] presents a heuristic called Global-resynchronize for the unbounded-latency 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, transparent synchronization graphs. Given an input synchronization graph G, Algorithm Global-resynchronize operates by first computing the family of subsets

$$T \equiv \{\chi(v_1, v_2) \mid ((v_1, v_2 \in V) \textbf{ and } (\rho_G(v_2, v_1) = \infty))\}. \tag{23}$$

The second constraint in (23), $\rho_G(v_2, v_1) = \infty$, ensures that inserting the candidate resynchronization edge (v_1, v_2) does not introduce a cycle, and thus that it does not reduce the estimated throughput or produce deadlock.

After computing the family of subsets specified by (23), 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 unbounded-latency resynchronization to the latency-constrained resynchronization problem, we replace the subset computation in (23) with

$$T \equiv \{\chi(v_1, v_2) \mid ((v_1, v_2 \in V) \textbf{ and } (\rho_G(v_2, v_1) = \infty) \textbf{ and } (L'(v_1, v_2) \le L_{max}))\}, \tag{24}$$

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. Assuming that $T_{fi(G)}(x, y)$ has been determined for all $x, y \in V$, L' can be computed from

$$L'(v_1, v_2) = max(\{(T_{fi(G)}(\upsilon, v_1) + T_{fi(G)}(v_2, o_L)), L_G\}), \tag{25}$$

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

A pseudocode specification of our extension of Global-resynchronize to the latency-constrained resynchronization problem, called Algorithm Global-LCR is shown in Figure 9.

In the companion paper [7], we showed that Algorithm Global-resynchronize has $O(sn^4)$ time-complexity, where n is the number of actors in the input synchronization graph, and s is the number of feedforward synchronization edges. Since the longest path quantities $T_{fi(G)}(*, *)$ can be computed in $O(n^3)$ time and updated in $O(n^2)$ time, it is easily verified that the $O(sn^4)$ bound also applies to our extension to latency-constrained resynchronization; that is, the time complexity of Algorithm Global-LCR is $O(sn^4)$.

Figure 11 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. Here, exc represents the excitation input, each v_i represents the computation for the ith voice, and the actors marked with “+” signs specify adders. Execution time estimates for the actors are shown in the table at the bottom of the figure. In this example, exc and out are respectively the latency input and latency output, and the latency is 170. There are ten synchronization edges shown, and none of these are redundant.

Figure 12 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 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.

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

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

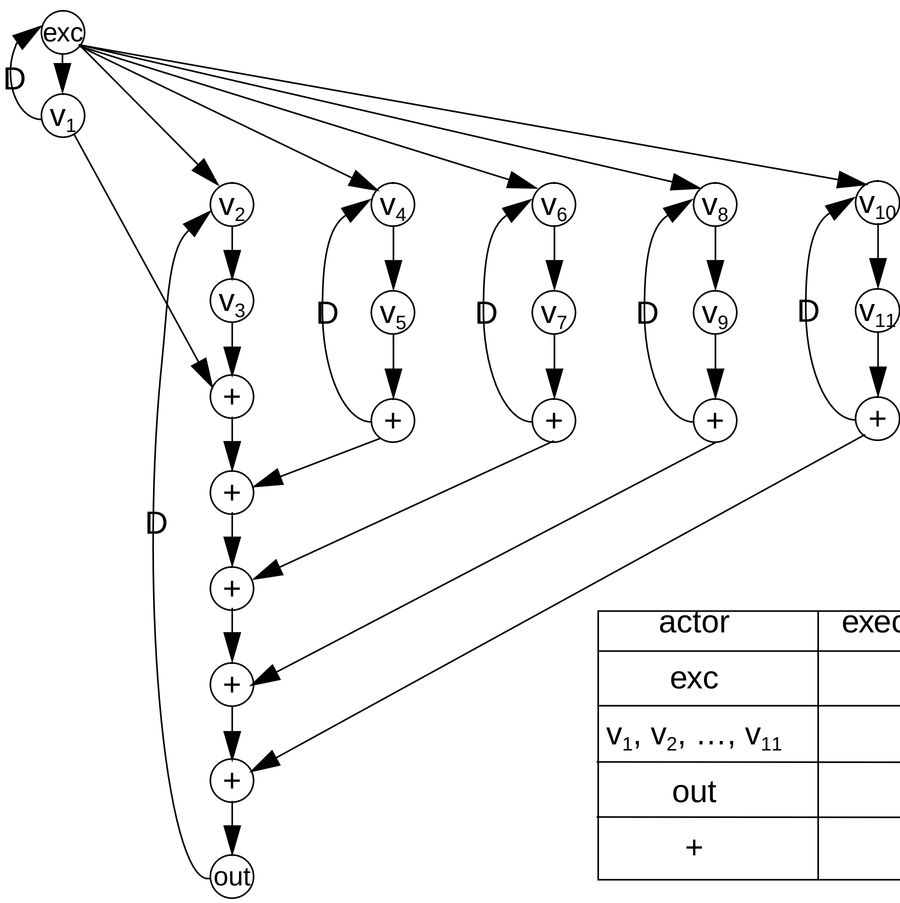
Figure 10. A heuristic for latency-constrained resynchronization.

Figure 13 illustrates how the placement of synchronization edges changes as the heuristic is able to attain lower synchronization costs. Each of the four topologies corresponds to a breakpoint in the plot of Figure 12. For example Figure 13(a) shows the synchronization graph computed by the heuristic for the lowest latency constraint value for which it computes a solution that has 9 synchronization edges.

10. Conclusions

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

We have focused on a restricted class of dataflow graphs, called *transparent* graphs, that permits efficient computation of latency, and efficient evaluation of the change in latency that



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

Figure 11. The synchronization graph that results from a six processor schedule of a music synthesizer based on the Karplus-Strong technique.

arises from the insertion of a new synchronization operation. A transparent dataflow graph is an application graph that contains at least one delay-free path from the input to the output.

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

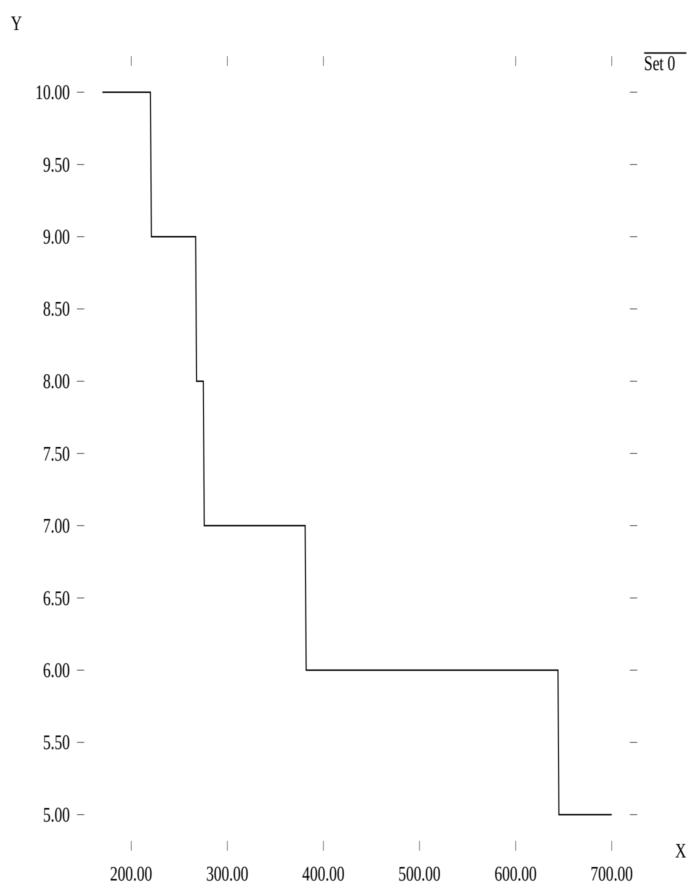


Figure 12. Performance of the heuristic on the example of Figure 11. The vertical axis gives the number of synchronization edges; the horizontal axis corresponds to the latency constraint.

for a very restricted class of transparent graphs; we have derived an efficient, polynomial-time

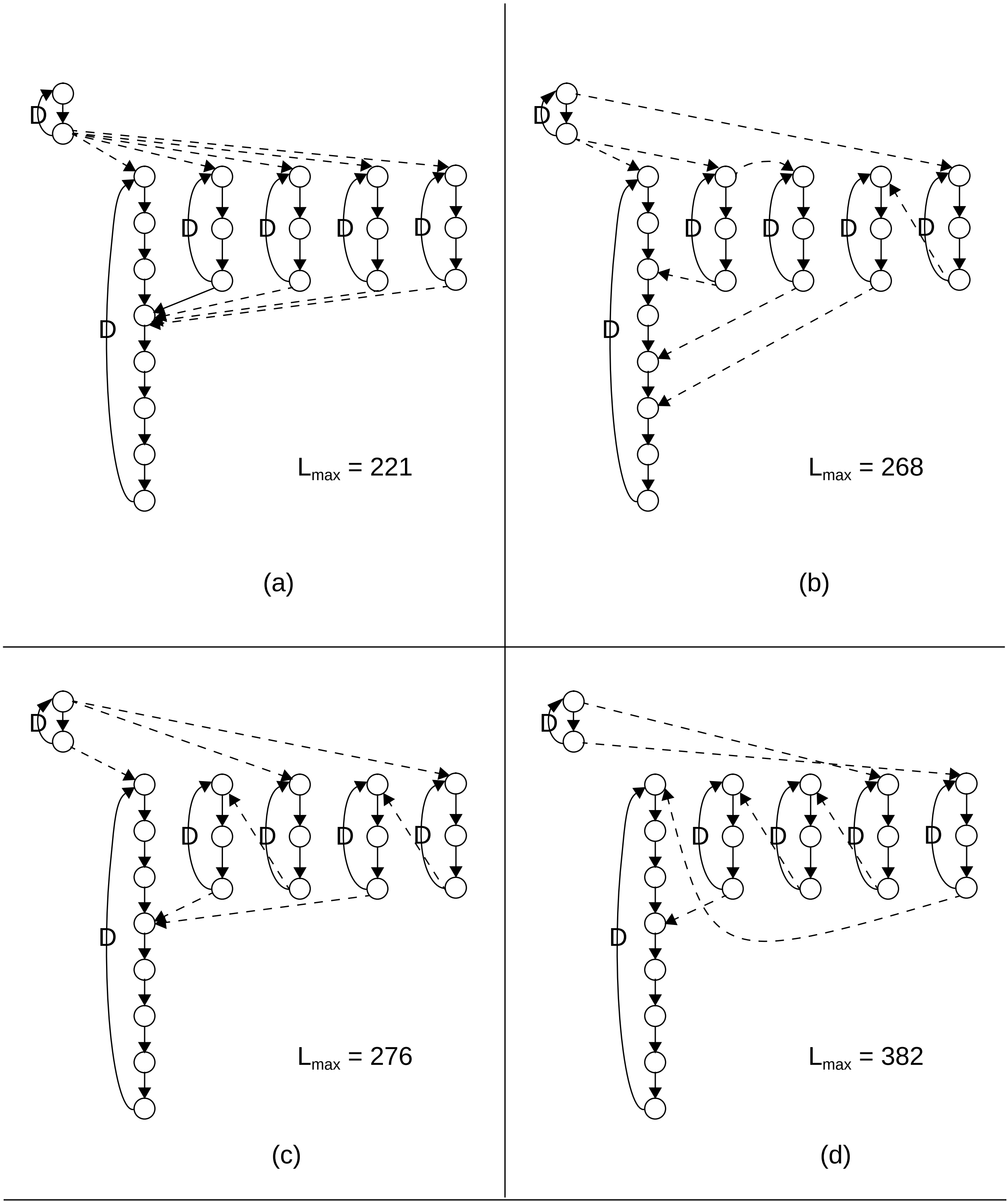


Figure 13. Synchronization graphs computed by the heuristic for different values of L_{max}.

algorithm that computes optimal latency-constrained resynchronizations for two-processor systems; and we have extended the heuristic presented in the companion paper [7] for unbounded-latency 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 can be used as a post-processing step to improve the performance of any of the large number of static multiprocessors scheduling techniques for iterative dataflow programs, such as those described in [1, 2, 12, 16, 17, 23, 27, 29, 33, 34].

11. Acknowledgments

A portion of this research was undertaken as part of the Ptolemy project, which is supported by the Advanced Research Projects Agency and the U.S. Air Force (under the RASSP program, contract F33615-93-C-1317), the State of California MICRO program, and the following companies: Bell Northern Research, Cadence, Dolby, Hitachi, Lucky-Goldstar, Mentor Graphics, Mitsubishi, Motorola, NEC, Philips, and Rockwell.

The authors are grateful to Jeanne-Anne Whitacre of Hitachi America for her great help in formatting this document.

12. Glossary

$\lvert S\rvert$:	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 $Delay(p) \le delay(s)$, 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 nonzero 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

$$\rho(x_1, y_1) + \rho(y_2, x_2) \le delay((x_1, x_2)).$$

$t(v)$: 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.

13. 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, "Latency-Constrained Resynchronization For Multiprocessor DSP Implementation," *Proceedings of the 1996 International Conference on Application-Specific Systems, Architectures and Processors*, August, 1996.

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

[6] S. S. Bhattacharyya, S. Sriram, and E. A. Lee, *Resynchronization for Embedded Multiprocessors*, Memorandum UCB/ERL M95/70, Electronics Research Laboratory, University of California at Berkeley, September, 1995.

[7] S. S. Bhattacharyya, S. Sriram, and E. A. Lee, *Resynchronization of Multiprocessor Schedules — Part 1: Fundamental Concepts and Unbounded-latency Analysis*, Memorandum UCB/ERL M96/55, Electronics Research Laboratory, University of California at Berkeley, October, 1996.

[8] S. S. Bhattacharyya, S. Sriram, and E. A. Lee, "Self-Timed Resynchronization: A Post-Optimization for Static Multiprocessor Schedules," *Proceedings of the International Parallel Processing Symposium*, 1996.

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

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

[11] 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.

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

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

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

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

[16] 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.

[17] 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.

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

[19] 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.

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

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

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

[23] 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.

[24] 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.

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

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

[27] 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.

[28] 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.

[29] 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.

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

[31] 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.

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

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

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