Memorandum UCB/ERL M96/55, Electronics Research Laboratory, U. C. Berkeley, October, 1996.

RESYNCHRONIZATION OF MULTIPROCESSOR SCHEDULES: PART 1 — FUNDAMENTAL CONCEPTS AND UNBOUNDED-LATENCY ANALYSIS

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

1. Abstract

This paper introduces a technique, called *resynchronization*, for reducing synchronization overhead in embedded multiprocessor implementations. The technique exploits the well-known observation [35] that in a given multiprocessor implementation, certain synchronization operations may be *redundant* in the sense that their associated sequencing requirements are ensured by other synchronizations in the system. The goal of resynchronization is to introduce new synchronizations in such a way that the number of additional synchronizations that become redundant exceeds the number of new synchronizations that are added, and thus the net synchronization cost is reduced.

Our study is based in the context of *self-timed* execution of *iterative dataflow* programs. An iterative dataflow program consists of a dataflow representation of the body of a loop that is to be iterated infinitely; dataflow programming in this form has been employed extensively, particularly in the context of software for digital signal processing applications. Self-timed execution refers to a combined compile-time/run-time scheduling strategy in which processors synchronize with one another only based on inter-processor communication requirements, and thus, synchronization of processors at the end of each loop iteration does not generally occur.

After reviewing our model for the analysis of synchronization overhead, we define the general form of our resynchronization problem; we show that optimal resynchronization is intractable by establishing a correspondence to the *set covering* problem; and based on this correspondence, we develop an efficient heuristic for resynchronization. Also, we show that for a broad class of iterative dataflow graphs, optimal resynchronizations can be computed by means of an efficient polynomial-time algorithm. We demonstrate the utility of our resynchronization tech-

niques through a practical example of a music synthesis system.

2. Introduction

This paper develops a technique called *resynchronization* for reducing the rate at which synchronization operations must be performed in a shared memory multiprocessor system. Resynchronization is based on the concept that there can be redundancy in the synchronization functions of a given multiprocessor implementation [35]. Such redundancy arises whenever the objective of one synchronization operation is guaranteed as a side effect of other synchronizations in the system. In the context of noniterative execution, Shaffer showed that the amount of run-time overhead required for synchronization can be reduced significantly by detecting redundant synchronizations and implementing only those synchronizations that are found not to be redundant; an efficient, optimal algorithm was also proposed for this purpose [35], and this algorithm was subsequently extended to handle iterative computations [5].

The objective of resynchronization is to introduce new synchronizations in such a way that the number of original synchronizations that consequently become redundant is significantly less than number of new synchronizations. We study this problem in the context of self-timed execution of iterative *synchronous dataflows* (SDF) programs. An iterative dataflow program consists of a dataflow representation of the body of a loop that is to be iterated infinitely; SDF programming in this form has proven to be a useful model for representing a significant class of digital signal processing (DSP) algorithms, and it has been used as the foundation for numerous DSP design environments, in which DSP applications are represented as hierarchies of block diagrams. Examples of commercial tools that employ SDF are the Signal Processing Worksystem (SPW) by Cadence, COSSAP by Synopsys, and Virtuoso Synchro by Eonic Systems. Research tools developed at universities that use SDF and related models include DESCARTES [33], GRAPE [20], Ptolemy [10], and the Warp compiler [31]. A wide variety of techniques have been developed to schedule SDF programs for efficient multiprocessor implementation, such as those described in [1, 2, 11, 15, 16, 24, 28, 31, 36, 38]. The techniques developed in this paper can be used as a post-

processing step to improve the performance of implementations that use any of these scheduling techniques.

In 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. This form of "synchrony" should not be confused with the use of "synchronous" in synchronous languages [3]. The task represented by an actor can be of arbitrary complexity. In DSP design environments, it typically ranges in complexity from a basic operation such as addition or subtraction to a signal processing subsystem such as an FFT unit or an adaptive filter.

Each SDF edge has associated a non-negative integer *delay*. These delays represent initial tokens, and specify dependencies between iterations of 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 [22], our techniques can easily be adapted to general SDF graphs. We refer to a homogeneous SDF graph as a **dataflow graph** (**DFG**).

Our implementation model involves a *self-timed* scheduling strategy [23]. 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 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**) between processors is assumed to take place through shared memory, which could be global memory between all processors, or it could be distributed between pairs of processors (for example, hardware first-in-first-out (FIFO) queues or dual ported memory). Sender-receiver synchronization is performed by setting and testing flags in shared

memory; Section 4.2 provides details on the assumed synchronization protocols. Interfaces between hardware and software are typically implemented using memory-mapped registers in the address space of the programmable processor, which can be viewed as a kind of shared memory, and synchronization is achieved using flags that can be tested and set by the programmable component, and the same can be done by an interface controller on the hardware side [17]. Thus, in our context, effective resynchronization results in a significantly reduced rate of accesses to shared memory for the purpose of synchronization.

The resynchronization techniques developed in this paper are designed to improve the throughput of multiprocessor implementations. Frequently in real-time signal processing systems, latency is also an important issue, and although resynchronization improves the throughput, it generally degrades (increases) the latency. In this paper, we address the problem of resynchronization under the assumption that an arbitrary increase in latency is acceptable. Such a scenario arises, for example, in a wide variety of simulation applications. The companion paper [8] examines the relationship between resynchronization and latency, and addresses the problem of optimal resynchronization when only a limited increase in latency is tolerable. Partial summaries of the material in this paper and the companion paper have been presented in [9] and [4], respectively.

3. Background

We represent a DFG by an ordered pair (V, E), where V is the set of vertices (actors) and E is the set of edges. We refer to the source and sink actors of a DFG edge e by $src(e)$ and $snk(e)$, we denote the delay on e by $delay(e)$, and we frequently represent e by the ordered pair $(src(e), snk(e))$. We say that e is an **output edge** of $src(e)$, and that e is an **input edge** of $snk(e)$. Edge e is **delayless** if $delay(e) = 0$, and it is a **self loop** if $src(e) = snk(e)$.

Given $x, y \in V$, we say that x is a **predecessor** of y if there exists $e \in E$ such that $src(e) = x$ and $snk(e) = y$; we say that x is a **successor** of y if y is a predecessor of x. A **path** in (V, E) is a finite, nonempty sequence $(e_1, e_2, \ldots, e_n)$, where each e_i is a member of E, and $snk(e_1) = src(e_2)$, $snk(e_2) = src(e_3)$, …, $snk(e_{n-1}) = src(e_n)$. We say that the path

$p = (e_1, e_2, \ldots, e_n)$ **contains** each e_i and each contiguous subsequence of $(e_1, e_2, \ldots, e_n)$; p is **directed from** $src(e_1)$ **to** $snk(e_n)$; and each member of

$$\{src(e_1), src(e_2), \ldots, src(e_n), snk(e_n)\}$$

is **traversed by** p. A path that is directed from some vertex to itself is called a **cycle**, and a **simple cycle** is a cycle of which no proper subsequence is a cycle.

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

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

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

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

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

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".

By a **subgraph** of (V, E), we mean the directed graph formed by any $V' \subseteq V$ together with the set of edges $\{e \in E \mid src(e), snk(e) \in V'\}$. We denote the subgraph associated with the vertex-subset V' by $subgraph(V')$. We say that (V, E) is **strongly connected** if for each pair of distinct vertices x, y, there is a path directed from x to y and there is a path directed from y to x. We say that a subset $V' \subseteq V$ is strongly connected if $subgraph(V')$ is strongly connected. A **strongly connected component (SCC)** of (V, E) is a strongly connected subset $V' \subseteq V$ such

that no strongly connected subset of V properly contains V'. If V' is an SCC, then when there is no ambiguity, we may also say that $subgraph(V')$ is an SCC. If C_1 and C_2 are distinct SCCs in (V, E), we say that C_1 is a **predecessor SCC** of C_2 if there is an edge directed from some vertex in C_1 to some vertex in C_2; C_1 is a **successor SCC** of C_2 if C_2 is a predecessor SCC of C_1. An SCC is a **source SCC** if it has no predecessor SCC; an SCC is a **sink SCC** if it has no successor SCC; and an SCC is an **internal SCC** if it is neither a source SCC nor a sink SCC. An edge is a **feedforward** edge if it is not contained in an SCC, or equivalently, if it is not contained in a cycle; an edge that is contained in at least one cycle is called a **feedback** edge.

We denote the number of elements in a finite set S by $|S|$.

4. Synchronization model

In this section, we review the model that we use for analyzing synchronization in self-timed multiprocessor systems. The model was originally developed in [37] to study the execution and interprocessor communication patterns of actors under self-timed evolution, and in [6], the model was augmented for the analysis of synchronization overhead.

A DFG 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* 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**

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

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

4.1 The synchronization graph

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.*

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 are implemented as buffers in shared memory, whereas the synchronization edges represent synchronization constraints, and are implemented by updating and testing flags in shared memory. 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 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.

4.2 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 allo-

cated 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, which can be computed efficiently from the synchronization graph topology [5]. In this 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 [6].

An important parameter in an implementation of FFS or FBS is the **back-off time** T_b. If a receiving processor finds that the corresponding IPC buffer is full, then the processor releases the shared memory bus, and waits T_b time units before requesting the bus again to re-check the shared memory synchronization variable. Similarly, a sending processor waits T_b time units between successive accesses of the same synchronization variable. The back-off time can be selected experimentally by simulating the execution of the given synchronization graph (with the available execution time estimates) over a wide range of candidate back-off times, and selecting the back-off time that yields the highest simulated throughput.

4.3 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 anal-

ysis in [32], the throughput (number of DFG iterations per unit time) of a synchronization graph G is given by $1/(\lambda_{max}(G))$, where

$$\lambda_{max}(G) \equiv \max_{\text{cycle } C \text{ in } G} \left\{ \frac{\sum_{v \in C} t^*(v)}{Delay(C)} \right\}. \tag{3}$$

The quotient in (3) is called the **cycle mean** of the cycle C, and the entire quantity on the RHS of (3) is called the **maximum cycle mean** of G. A cycle in G whose cycle mean is equal to the maximum cycle mean of G is called a **critical cycle** of G. 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 an estimate of the maximum cycle mean. The reciprocal of this estimate of the maximum cycle mean is called 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.

4.4 Preservation of synchronization graphs

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 $\rho_{G_1}(src(e), snk(e)) \leq delay(e)$.

The following theorem, which is developed in [6], underlies the validity of our synchronization optimizations.

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.

5. Related work

Shaffer has developed an algorithm that removes redundant synchronizations in the self-timed execution of a non-iterative DFG [35]. 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.

Filo, Ku and De Micheli have studied synchronization rearrangement in the context of minimizing the controller area for hardware synthesis of synchronization digital circuitry [13, 14], and significant differences in the underlying analytical models prevent these techniques from applying to our context. In the graphical hardware model of [14], called the *constraint graph* model, each vertex corresponds to a separate hardware device and edges have arbitrary weights that specify sequencing constraints. When the source vertex has bounded execution time, a positive weight $w(e)$ (*forward constraint*) imposes the constraint

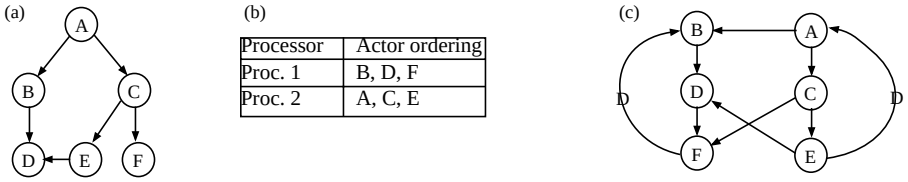


Figure 1. Part (c) shows the IPC graph that corresponds to the DFG of part (a) and the processor assignment / actor ordering of part (b). A "D" on top of an edge represents a unit delay.

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

while a negative weight (*backward constraint*) implies

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

If the source vertex has unbounded execution time, the forward and backward constraints are relative to the *completion* time of the source vertex. In contrast, in our synchronization graph model, multiple actors can reside on the same processing element (implying zero synchronization cost between them), and the timing constraints always correspond to the case where $w(e)$ is positive and equal to the execution time of $src(e)$.

The implementation models, and associated implementation cost functions are also significantly different. A constraint graph is implemented using a scheduling technique called *relative scheduling* [19], which can roughly be viewed as intermediate between self-timed and fully-static scheduling. In relative scheduling, the constraint graph vertices that have unbounded execution time, called *anchors,* are used as reference points against which all other vertices are scheduled: for each vertex v, an offset f_i is specified for each anchor a_i that affects the activation of v, and v is scheduled to occur once f_i clock cycles have elapsed from the completion of a_i, for each i.

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

As a result of the significant differences in both the scheduling models and the implementation models, the techniques developed for resynchronizing constraint graphs do not extend in any straightforward manner to the resynchronization of synchronization graphs for self-timed multiprocessor implementation, and the solutions that we have developed for synchronization graphs are significantly different in structure from those reported in [14]. For example, the fundamental relationships that we establish between set covering and our use of resynchronization have

not emerged in the context of constraint graphs.

6. Resynchronization

We refer to the process of adding one or more new synchronization edges and removing the redundant edges that result as *resynchronization* (defined more precisely below). Figure 2(a) illustrates how this concept can be used to reduce the total number of synchronizations in a multiprocessor implementation. Here, the dashed edges represent synchronization edges. Observe that if we insert the new synchronization edge $d_0(C, H)$, then two of the original synchronization edges — (B, G) and (E, J) — become redundant. Since redundant synchronization edges can be removed from the synchronization graph to yield an equivalent synchronization graph, we see that the net effect of adding the synchronization edge $d_0(C, H)$ is to reduce the number of synchronization edges that need to be implemented by 1. In Figure 2(b), we show the synchronization graph that results from inserting the *resynchronization edge* $d_0(C, H)$ into Figure 2(a), and then removing the redundant synchronization edges that result.

Definition 1 gives a formal definition of resynchronization that we will use throughout the remainder of this paper. This considers resynchronization only "across" feedforward edges. Resynchronization that includes inserting edges into SCCs, is also possible; however, in general,

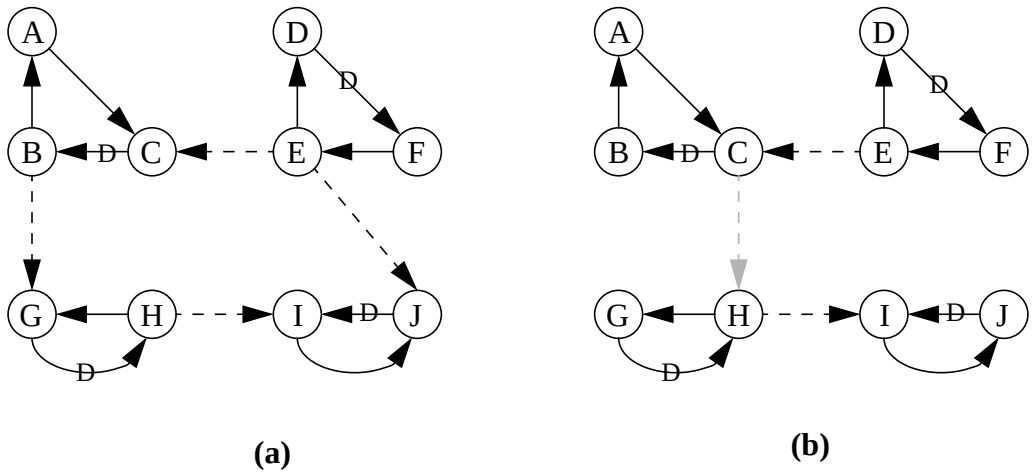


Figure 2. An example of resynchronization.

such resynchronization may increase the estimated throughput (see Theorem 2 at the end of Section 7). Thus, for our objectives, it must be verified that each new synchronization edge introduced in an SCC does not decrease the estimated throughput. To avoid this complication, which requires a check of significant complexity ($O(|V||E|\log_2(|V|))$, where (V, E) is the modified synchronization graph — this is using the Bellman Ford algorithm described in [21]) *for each* candidate resynchronization edge, we focus only on "feedforward" resynchronization in this paper. Future research will address combining the insights developed here for feedforward resynchronization with efficient techniques to estimate the impact that a given *feedback* resynchronization edge has on the estimated throughput.

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

If we let G denote the graph in Figure 2, then the set of feedforward edges is $F = \{(B, G), (E, J), (E, C), (H, I)\}$; $R = \{d_0(C, H), (E, C), (H, I)\}$ is a resynchronization of G; Figure 2(b) shows the DFG $G^* = (V, ((E - F) + R))$; and from Figure 2(b), it is easily verified that F, R, and G^* satisfy conditions (a) and (b) of Definition 1.

7. Properties of resynchronization

In this section, we introduce a number of useful properties of resynchronization that we will apply throughout the developments of this paper.

Lemma 1: Suppose that G and G' are synchronization graphs such that G' preserves G, and

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

Thus, if a synchronization graph G' preserves another synchronization graph G and p is a path in G from actor x to actor y, then there is at least one path p' in G' such that 1) the path p' is directed from x to y; 2) the cumulative delay on p' does not exceed the cumulative delay on p; and 3) every actor that is traversed by p is also traversed by p' (although p' may traverse one or more actors that are not traversed by p).

For example in Figure 2(a), if we let $x = B$, $y = I$, and $p = ((B, G), (G, H), (H, I))$, then the path $p' = ((B, A), (A, C), (C, H), (H, G), (G, H), (H, I))$ in Figure 2(b) confirms Lemma 1 for this example. Here $tr(p) = \{B, G, H, I\}$ and $tr(p') = \{A, B, C, G, H, I\}$.

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

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

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

The following lemma states that if a resynchronization contains a resynchronization edge e such that there is a delay-free path in the original synchronization graph from the source of e to the sink of e, then e must be redundant in the resychronized graph.

Lemma 2: Suppose that G is a synchronization graph; R is a resynchronization of G; and (x, y) is a resynchronization edge such that $\rho_G(x, y) = 0$. Then (x, y) is redundant in $\Psi(R, G)$.

Thus, a minimal resynchronization (fewest number of elements) has the property that

$\rho_G(x', y') > 0$ for each resynchronization edge (x', y').

Proof: Let p denote a minimum-delay path from x to y in G. Since (x, y) is a resynchronization edge, (x, y) is not contained in G, and thus, p traverses at least three actors. From Lemma 1, it follows that there is a path p' in $\Psi(R, G)$ from x to y such that $Delay(p') = 0$, and p' traverses at least three actors. Thus, $Delay(p') \leq delay((x, y))$ and $p' \neq (x, y)$, and we conclude that (x, y) is redundant in $\Psi(R, G)$. *QED.*

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

Theorem 2: If G is a synchronization graph, and G' is a synchronization graph that preserves G, then $\lambda_{max}(G') \geq \lambda_{max}(G)$.

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

$$\sum_{v \text{ is traversed by } C'} t(v) \geq \sum_{v \text{ is traversed by } C} t(v), \tag{4}$$

and this observation together with a) implies that the cycle mean of C' is greater than or equal to the cycle mean of C. Since C is a critical cycle in G, it follows that $\lambda_{max}(G') \geq \lambda_{max}(G)$. *QED.*

Thus, in any saving in synchronization cost obtained by rearranging synchronization edges may come at the expense of a decrease in estimated throughput. As implied by Definition 1, we avoid this complication by restricting our attention to feedforward synchronization edges. Clearly, resynchronization that rearranges only feedforward synchronization edges cannot decrease the estimated throughput since no new cycles are introduced and no existing cycles are altered. Thus, with the form of resynchronization that we address in this paper, any decrease in synchronization cost that we obtain is not diminished by a degradation of the estimated throughput.

8. Relationship to set covering

We refer to the problem of finding a resynchronization with the fewest number of elements as the **resynchronization problem**. In [6], we formally show that the resynchronization problem is NP-hard; in this section, we explain the intuition behind this result. To establish the NP-hardness of the resynchronization problem, we examine a special case that occurs when there are exactly two SCCs, which we call the **pairwise resynchronization problem**, and we derive a polynomial-time reduction from the classic *set covering problem* [12], a well-known NP-hard problem, to the pairwise resynchronization problem. In the set covering problem, one is given a finite set X and a family T of subsets of X, and asked to find a minimal (fewest number of members) subfamily $T_s \subseteq T$ such that $\bigcup_{t \in T_s} t = X$. A subfamily of T is said to *cover* X if each member of X is contained in some member of the subfamily. Thus, the set covering problem is the problem of finding a minimal cover.

Definition 2: Given a synchronization graph G, let (x_1, x_2) be a synchronization edge in G, and let (y_1, y_2) be an ordered pair of actors in G. We say that (y_1, y_2) **subsumes** (x_1, x_2) in G if $\rho(x_1, y_1) + \rho(y_2, x_2) \leq delay((x_1, x_2))$.

Thus, every synchronization edge subsumes itself, and intuitively, if (x_1, x_2) is a synchronization edge, then (y_1, y_2) subsumes (x_1, x_2) if and only if a zero-delay synchronization edge directed from y_1 to y_2 makes (x_1, x_2) redundant.

The following fact is easily verified from Definitions 1 and 2.

Fact 1: Suppose that G is a synchronization graph that contains exactly two SCCs, F is the set of feedforward edges in G, and F' is a resynchronization of G. Then for each $e \in F$, there exists $e' \in F'$ such that $(src(e'), snk(e'))$ subsumes e in G.

An intuitive correspondence between the pairwise resynchronization problem and the set covering problem can be derived from Fact 1. Suppose that G is a synchronization graph with exactly two SCCs C_1 and C_2 such that each feedforward edge is directed from a member of C_1

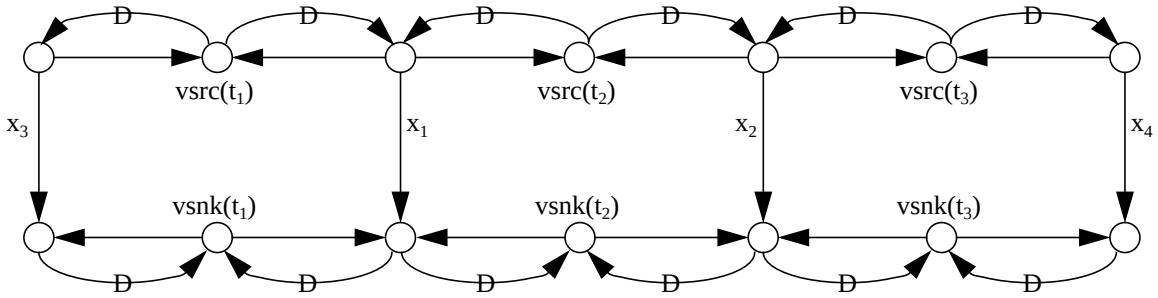
to a member of C_2. We start by viewing the set F of feedforward edges in G as the finite set that we wish to cover, and with each member p of $\{(x, y)|(x \in C_1, y \in C_2)\}$, we associate the subset of F defined by $\chi(p) \equiv \{e \in F|(p \text{ subsumes } e)\}$. Thus, $\chi(p)$ is the set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

Figure 3 helps to illustrate this intuition. Suppose that we are given the set $X = \{x_1, x_2, x_3, x_4\}$, and the family of subsets $T = \{t_1, t_2, t_3\}$, where $t_1 = \{x_1, x_3\}$, $t_2 = \{x_1, x_2\}$, and $t_3 = \{x_2, x_4\}$. To construct an instance of the pairwise resynchronization problem, we first create two vertices and an edge directed between these vertices *for each* member of X; we label each of the edges created in this step with the corresponding member of X. Then for each $t \in T$, we create two vertices $vsrc(t)$ and $vsnk(t)$. Next, for each relation $x_i \in t_j$ (there are six such relations in this example), we create two delayless edges — one directed from the source of the edge corresponding to x_i to $vsrc(t_j)$, and another directed from $vsnk(t_j)$ to the sink of the edge corresponding to x_i. This last step has the effect of making each pair $(vsrc(t_i), vsnk(t_i))$ subsume exactly those edges that correspond to members of t_i; in other words, after this construction, $\chi((vsrc(t_i), vsnk(t_i))) = t_i$, for each i. Finally, for each edge created in the previous step, we create a corresponding feedback edge oriented in the opposite direction, and having a unit delay.

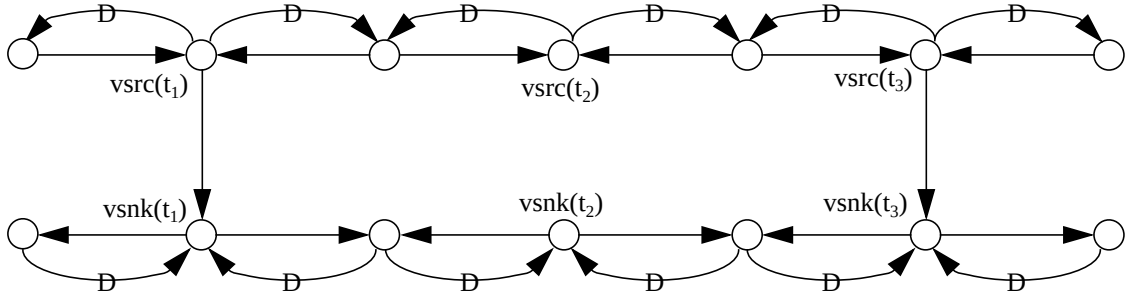
Figure 3(a) shows the synchronization graph that results from this construction process. Here, it is assumed that each vertex corresponds to a separate processor; the associated unit delay, self loop edges are not shown to avoid excessive clutter. Observe that the graph contains two SCCs — $(\{src(x_i)\} \cup \{vsrc(t_i)\})$ and $(\{snk(x_i)\} \cup \{vsnk(t_i)\})$ — and that the set of feedforward edges is the set of edges that correspond to members of X. Now, recall that a major corre-

spondence between the given instance of set covering and the instance of pairwise resynchronization defined by Figure 3(a) is that $\chi((vsrc(t_i), vsnk(t_i))) = t_i$, for each i. Thus, if we can find a minimal resynchronization of Figure 3(a) such that each edge in this resynchronization is directed from some $vsrc(t_k)$ to the corresponding $vsnk(t_k)$, then the associated t_k's form a minimum cover of X. For example, it is easy, albeit tedious, to verify that the resynchronization illustrated in Figure 3(b), $\{d_0(vsrc(t_1), vsnk(t_1)), d_0(vsrc(t_3), vsnk(t_3))\}$, is a minimal resynchronization of Figure 3(a), and from this, we can conclude that $\{t_1, t_3\}$ is a minimal cover for X. From inspection of the given sets X and T, it is easily verified that this conclusion is correct.

This example illustrates how an instance of pairwise resynchronization can be constructed



(a)



(b)

Figure 3. (a) An instance of the pairwise resynchronization problem that is derived from an instance of the set covering problem; (b) the DFG that results from a solution to this instance of pairwise resynchronization.

(in polynomial time) from an instance of set covering, and how a solution to this instance of pairwise resynchronization can easily be converted into a solution of the set covering instance. Our formal proof of the NP-hardness of pairwise resynchronization, presented in [6], is a generalization of the example in Figure 3.

9. Heuristic solutions

9.1 Applying set covering techniques to pairs of SCCs

A heuristic framework for the pairwise resynchronization problem emerges naturally from the relationship that we have established between set covering and pairwise resynchronization in Section 8. Given an arbitrary algorithm *COVER* that solves the set covering problem, and given an instance of pairwise resynchronization that consists of two SCCs C_1 and C_2, and a set S of feedforward synchronization edges directed from members of C_1 to members of C_2, this heuristic framework first computes the subset

$$\chi((u, v)) = \{e \in S | (\rho_G(src(e), u) = 0) \textbf{ and } (\rho_G(v, snk(e)) = 0)\}$$

for each ordered pair of actors (u, v) that is contained in the set

$$T \equiv \{(u', v') | (u' \text{ is in } C_1 \textbf{ and } v' \text{ is in } C_2)\},$$

and then applies the algorithm *COVER* to the instance of set covering defined by the set S together with the family of subsets $\{\chi((u', v')) | ((u', v') \in T)\}$. If Ξ denotes the solution returned by *COVER*, then a resynchronization for the given instance of pairwise resynchronization can be derived by $\{d_0(u, v) | \chi((u, v)) \in \Xi\}$. This resynchronization is the solution returned by the heuristic framework.

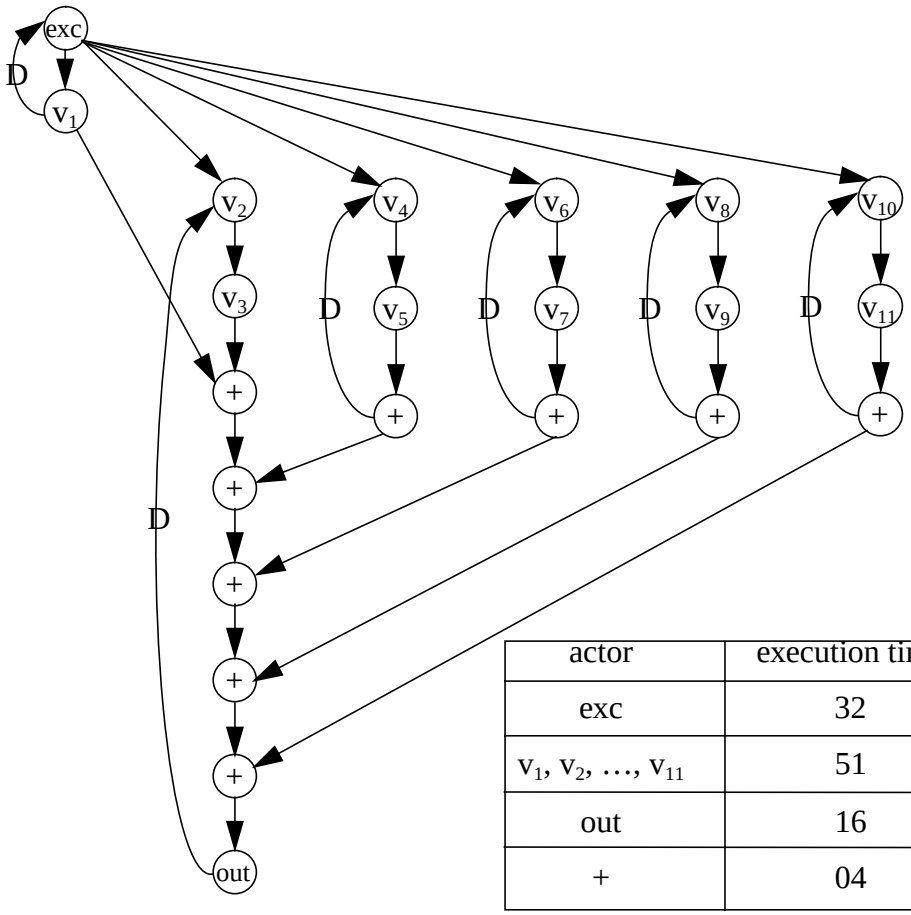
From the correspondence between set covering and pairwise resynchronization that is outlined in Section 8, it follows that the quality of a resynchronization obtained by our heuristic framework is determined entirely by the quality of the solution computed by the set covering algorithm that is employed; that is, if the solution computed by *COVER* is *X*% worse (*X*% more subfamilies) than an optimal set covering solution, then the resulting resynchronization will be *X*% worse (*X*% more synchronization edges) than an optimal resynchronization of the given

instance of pairwise resynchronization.

The application of our heuristic framework for pairwise resynchronization to each pair of SCCs, in some arbitrary order, in a general synchronization graph yields a heuristic framework for the general resynchronization problem (a pseudocode specification of this approach can be found in [7]). However, a major limitation of this extension to general synchronization graphs arises from its inability to consider resynchronization opportunities that involve paths that traverse more than two SCCs, and paths that contain more than one feedforward synchronization edge.

Thus, in general, the quality of the solutions obtained by this approach will be worse than the quality of the solutions that are derived by the particular set covering heuristic that is employed, and roughly, this discrepancy can be expected to increase as the number of SCCs increases relative to the number of synchronization edges in the original synchronization graph.

For example, Figure 4 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, the only pair of distinct SCCs that have more than one synchronization edge between them is the pair consisting of the SCC containing $\{exc, v_1\}$ and the SCC containing v_2, v_3, five addition actors, and the actor labeled *out*. Thus, the best result that can be derived from the heuristic extension for general synchronization graphs described above is a resynchronization that optimally rearranges the synchronization edges between these two SCCs in isolation, and leaves all other synchronization edges unchanged. Such a resynchronization is illustrated in Figure 5. This synchronization graph has a total of nine synchronization edges, which is only one less than the number of synchronization edges in the original graph. In contrast, we will show in the following subsection that with a more flexible approach to resynchronization, the total synchronization cost of this example can be reduced to only five synchronization edges.



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

Figure 4. The synchronization graph that results from a six processor schedule of a music synthesizer based on the Karplus-Strong technique.

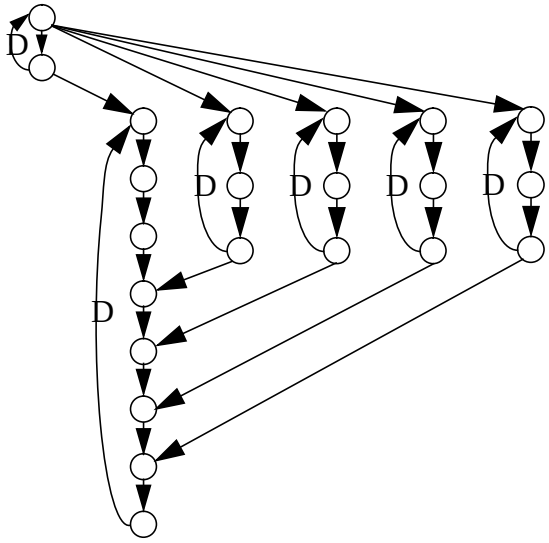


Figure 5. The synchronization graph that results from applying the heuristic framework based on pairwise resynchronization to the example of Figure 4.

9.2 A more flexible approach

In this subsection, we present a more global approach to resynchronization, called Algorithm Global-resynchronize, which overcomes the major limitation of the pairwise approach discussed in Section 9.1. Algorithm Global-resynchronize is based on the simple greedy approximation algorithm for set covering that repeatedly selects a subset that covers the largest number of *remaining elements*, where a remaining element is an element that is not contained in any of the subsets that have already been selected. In [18, 25] it is shown that this set covering technique is guaranteed to compute a solution whose cardinality is no greater than $(\ln(|X|) + 1)$ times that of the optimal solution, where X is the set that is to be covered.

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

$$T \equiv \{\chi(v_1, v_2) \big| ((v_1, v_2 \in V) \textbf{ and } (\rho_G(v_2, v_1) = \infty))\}, \tag{5}$$

where $G = (V, E)$ is the input synchronization graph. The constraint $\rho_G(v_2, v_1) = \infty$ in (5) ensures that inserting the resynchronization edge (v_1, v_2) does not introduce a cycle, and thus that it does not introduce deadlock or reduce the estimated throughput.

Algorithm *Global-resynchronize* determines the family of subsets specified by (5), chooses a member of this family that has maximum cardinality, inserts the corresponding delayless resynchronization edge, removes all synchronization edges that it subsumes, and updates the values $\rho_G(x, y)$ for the new synchronization graph that results. This entire process is then repeated on the new synchronization graph, and it continues until it arrives at a synchronization graph for which the computation defined by (5) produces the empty set — that is, the algorithm terminates when no more resynchronization edges can be added. Figure 6 gives a pseudocode specification of this algorithm (with some straightforward modifications to improve the running time).

Clearly, each time a delayless resynchronization edge is added to a synchronization graph,

the number of ordered vertex pairs (x, y) that satisfy $\rho_G(x, y) = 0$ is increased by at least one. Thus, the number of iterations of the **while** loop in Figure 6 is bounded above by $|V|^2$. The complexity of one iteration of the **while** loop is dominated by the computation in the pair of nested **for** loops. The computation of one iteration of the inner **for** loop is dominated by the time required to compute $\chi(x, y)$ for a specific actor pair (x, y). Assuming $\rho_G(x', y')$ is available for all

```
function Global-resynchronize
input: a synchronization graph G = (V, E)
output: an alternative 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 ∈ V
        for y ∈ V
            if (ρ_G(y, x) = ∞)
                χ* = χ((x, y))
                if (|χ*| > M)
                    M = |χ*|
                    best = (x, y)
                end if
            end if
        end for
    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 6. A heuristic for resynchronization.

$x', y' \in V$, the time to compute $\chi(x, y)$ is $O(s_c)$, where s_c is the number of feedforward synchronization edges in the current synchronization graph. Since the number of feedforward synchronization edges never increases from one iteration of the **while** loop to the next, it follows that the time-complexity of the overall algorithm is $O(s|V|^4)$, where s is the number of feedforward synchronization edges in the input synchronization graph. In practice, however, the number of resynchronization steps (**while** loop iterations) is usually much lower than $|V|^2$ since the constraints on the introduction of cycles severely limit the number of resynchronization steps. Thus, our $O(s|V|^4)$ bound can be viewed as a very conservative estimate.

9.3 Example

Figure 7 shows the optimized synchronization graph that is obtained when Algorithm *Global-resynchronize* is applied to the example of the Figure 4.

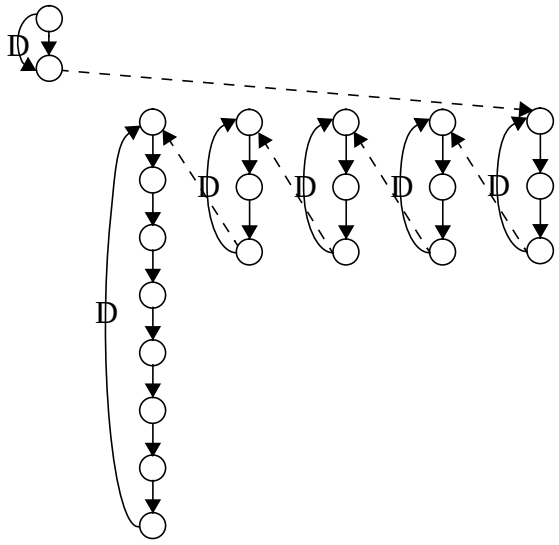


Figure 7. The optimized synchronization graph that is obtained when Algorithm *Global-resynchronize* is applied to the example of the Figure 4.

Observe that the total number of synchronization edges has been reduced from 10 to 5. The total number of "resynchronization steps" (number of while-loop iterations) required by the heuristic to complete this resynchronization is 7.

Table 1 shows the relative throughput improvement delivered by the optimized synchroni-

zation graph of Figure 7 over the original synchronization graph as the shared memory access time varies from 1 to 10 processor clock cycles. The assumed synchronization protocol is FFS, and the back-off time for each simulation is obtained by the experimental procedure mentioned in Section 4.2. The second and fourth columns show the *average iteration period* for the original synchronization graph and the resynchronized graph, respectively. The average iteration period, which is the reciprocal of the average throughput, is the average number of time units required to execute an iteration of the synchronization graph. From the sixth column, we see that the resynchronized graph consistently attains a throughput improvement of 22% to 26%. This improvement includes the effect of reduced overhead form maintaining synchronization variables and reduced contention for shared memory. The third and fifth columns of Table 1 show the average number of shared memory accesses per iteration of the synchronization graph. Here we see that the resynchronized solution consistently obtains at least a 30% improvement over the original synchronization graph. Since accesses to shared memory typically require significant amounts of

Memory access time	Original graph		Resynchronized graph		Percentage decrease in iter. period
	Iter. period	Memory accesses/pd	Iter. period	Memory accesses/pd	
1	250	67	195	43	22%
2	292	66	216	43	26%
3	335	64	249	43	26%
4	368	63	273	40	26%
5	408	63	318	43	22%
6	459	63	350	43	24%
7	496	63	385	43	22%
8	540	63	420	43	22%
9	584	63	455	43	22%
10	655	65	496	43	24%

Table 1. Performance comparison between the resynchronized solution and the original synchronization graph for the example of Figure 4.

energy, particularly for a multiprocessor system that is not integrated on a single chip, this reduction in the average rate of shared memory accesses is especially useful when low power consumption is an important implementation issue.

10. Efficient, optimal resynchronization for a class of synchronization graphs

In this section, we show that although optimal resynchronization is intractable for general synchronization graphs, a broad class of synchronization graphs exists for which optimal resynchronizations can be computed using an efficient polynominal-time algorithm.

10.1 Chainable synchronization graph SCCs

Definition 3: Suppose that C is an SCC in a synchronization graph G, and x is an actor in C. Then x is an **input hub** of C if for each feedforward synchronization edge e in G whose sink actor is in C, we have $\rho_C(x, snk(e)) = 0$. Similarly, x is an **output hub** of C if for each feedforward synchronization edge e in G whose source actor is in C, we have $\rho_C(src(e), x) = 0$. We say that C is **linkable** if there exist actors x, y in C such that x is an input hub, y is an output hub, and $\rho_C(x, y) = 0$. A synchronization graph is **chainable** if each SCC is linkable.

For example, consider the SCC in Figure 8(a), and assume that the dashed edges represent the synchronization edges that connect this SCC with other SCCs. This SCC has exactly one input hub, actor A, and exactly one output hub, actor F, and since $\rho(A, F) = 0$, it follows that the

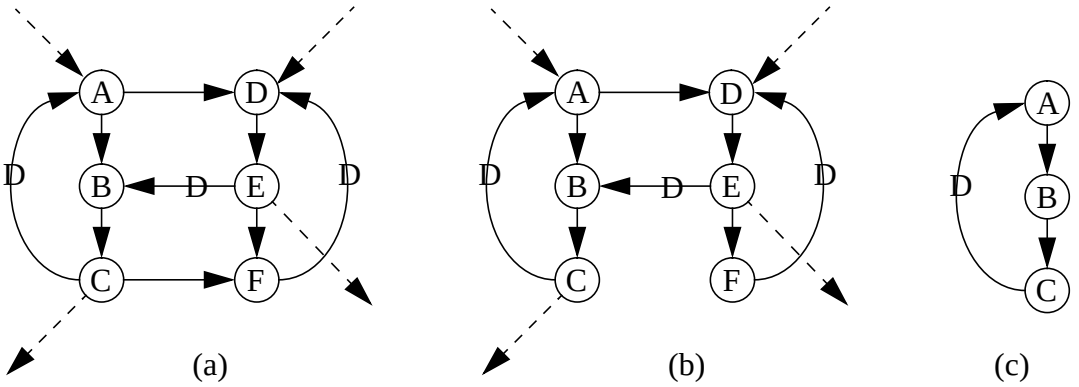


Figure 8. An illustration of input and output hubs for synchronization graph SCCs.

SCC is linkable. However, if we remove the edge (C, F), then the resulting graph (shown in Figure 8(b)) is not linkable since it does not have an output hub. A class of linkable SCCs that occur commonly in practical synchronization graphs are those SCCs that corresponds to only one processor, such as the SCC shown in Figure 8(c). In such cases, the first actor executed on the processor is always an input hub and the last actor executed is always an output hub.

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

An important property of linkable synchronization graphs is that if C_1 and C_2 are distinct linkable SCCs, then all synchronization edges directed from C_1 to C_2 are subsumed by the single ordered pair (l_1, l_2), where l_1 denotes the selected output hub of C_1 and l_2 denotes the selected input hub of C_2. Furthermore, if there exists a path between two SCCs C_1', C_2' of the form $((o_1, i_2), (o_2, i_3), \ldots, (o_{n-1}, i_n))$, where o_1 is the selected output hub of C_1', i_n is the selected input hub of C_2', and there exist distinct SCCs $Z_1, Z_2, \ldots, Z_{n-2} \notin \{C_1', C_2'\}$ such that for $k = 2, 3, \ldots, (n-1)$, i_k, o_k are respectively the selected input hub and the selected output hub of Z_{k-1}, then all synchronization edges between C_1' and C_2' are redundant.

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

This chaining technique can be viewed as a form of pipelining, where each SCC in the

output synchronization graph corresponds to a pipeline stage. Pipelining has been used extensively to increase throughput via improved parallelism ("temporal parallelism") in multiprocessor DSP implementations (see for example, [2, 16, 27]). However, in our application of pipelining, the load of each processor is unchanged, and the estimated throughput is not affected (since no new cyclic paths are introduced), and thus, the benefit to the *overall* throughput of our chaining technique arises chiefly from the optimal reduction of synchronization overhead.

The time-complexity of our optimal algorithm for resychronizing chainable synchronization graphs is $O(v^2)$, where v is the number of synchronization graph actors.

10.2 Comparison to the Global-Resynchronize heuristic

It is easily verified that the original synchronization graph for the music synthesis example of Section 9.2, shown in Figure 4, is chainable. Thus, the chaining technique presented in Section 10.1 is guaranteed to produce an optimal resynchronization for this example, and since no feedback synchronization edges are present, the number of synchronization edges in the resynchronized solution is guaranteed to be equal to one less than the number of SCCs in the original synchronization graph; that is, the optimized synchronization graph contains $6 - 1 = 5$ synchronization edges. From Figure 7, we see that this is precisely the number of synchronization edges

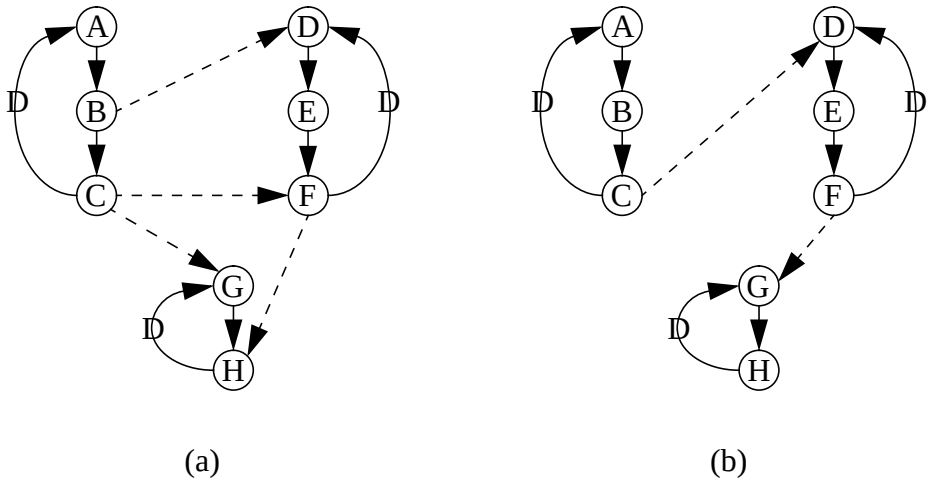


Figure 9. An illustration of an algorithm for optimal resynchronization of chainable synchronization graphs. The dashed edges are synchronization edges.

in the synchronization graph that results from our implementation of Algorithm Global-resynchronize.

However, Algorithm Global-resynchronize does not always produce optimal results for chainable synchronization graphs. For example, consider the synchronization graph shown in Figure 10(a), which corresponds to an eight-processor schedule in which each of the following subsets of actors are assigned to a separate processor — $\{I\}$, $\{J\}$, $\{G, K\}$, $\{C, H\}$, $\{D\}$, $\{E, L\}$, $\{A, F\}$, and $\{B\}$. The dashed edges are synchronization edges, and the remaining edges connect actors that are assigned to the same processor. The total number of synchronization edges is 14. Now it is easily verified that actor K is both an input hub and an output hub for the SCC $\{C, G, H, J, K\}$, and similarly, actor L is both an input and output hub for the SCC $\{A, D, E, F, L\}$. Thus, we see that the overall synchronization graph is chainable. It is easily verified that the chaining technique developed in Section 10.1 uniquely yields the optimal resynchronization illustrated in Figure 10(b), which contains only 11 synchronization edges.

In contrast, the quality of the resynchronization obtained for Figure 10(a) by Algorithm Global-resynchronize depends on the order in which the actors are traversed by each of the two nested **for** loops in Figure 6. For example, if both loops traverse the actors in alphabetical order, then Global-resynchronize obtains the sub-optimal solution shown in Figure 10(c), which contains 12 synchronization edges.

However, actor traversal orders exist for which Global-resynchronize achieves optimal resynchronizations of Figure 10(a). One such ordering is $K, D, C, B, E, F, G, H, I, J, L, A$; if both **for** loops traverse the actors in this order, then Global-resynchronize yields the same resynchronized graph that is computed uniquely by the chaining technique of Section 10.1 (Figure 10(b)). It is an open question whether or not given an arbitrary chainable synchronization graph, actor traversal orders always exist with which Global-resynchronize arrives at optimal resynchronizations. Furthermore, even if such traversal orders are always guaranteed to exist, it is doubtful that they can, in general, be computed efficiently.

In addition to guaranteed optimality, another important advantage of the chaining tech-

nique for chainable synchronization graphs is its relatively low time-complexity ($O(v^2)$ versus $O(sv^4)$ for Global-resynchronize), where v is the number of synchronization graph actors, and s is the number of feedforward synchronization edges. The primary disadvantage is, of course, its restricted applicability. A useful direction for further investigation is the integration of the chaining technique with algorithm Global-resynchronize for general (not necessarily chainable) synchronization graphs.

10.3 A generalization of the chaining technique

The chaining technique developed in Section 10.1 can be generalized to optimally resynchronize a somewhat broader class of synchronization graphs. This class consists of all synchronization graphs for which each source SCC has an output hub (but not necessarily an input hub), each sink SCC has an input hub (but not necessarily an output hub), and each internal SCC is linkable. In this case, the internal SCCs are pipelined as in the previous algorithm, and then for each source SCC, a synchronization edge is inserted from one of its output hubs to the selected input hub of the first SCC in the pipeline of internal SCCs, and for each sink SCC, a synchronization edge is inserted to one of its input hubs from the selected output hub of the last SCC in the pipeline of internal SCCs. If there are no internal SCCs, then the sink SCCs are pipelined by selecting one input hub from each SCC, and joining these input hubs with a chain of synchronization edges. Then a synchronization edge is inserted from an output hub of each source SCC to an input hub of the first SCC in the chain of sink SCCs.

11. Conclusions

This paper develops a post-optimization called resynchronization for self-timed, embedded multiprocessor implementations. The goal of resynchronization is to introduce new synchronizations in such a way that the number of additional synchronizations that become redundant exceeds the number of new synchronizations that are added, and thus the net synchronization cost is reduced.

We show that optimal resynchronization is intractable by deriving a reduction from the

classic set covering problem. However, we define a broad class of systems for which optimal resynchronization can be performed in polynomial time. We also present a heuristic algorithm for resynchronization of general systems that emerges naturally from the correspondence to set covering. We illustrate the performance of our implementation of this heuristic on a multiprocessor

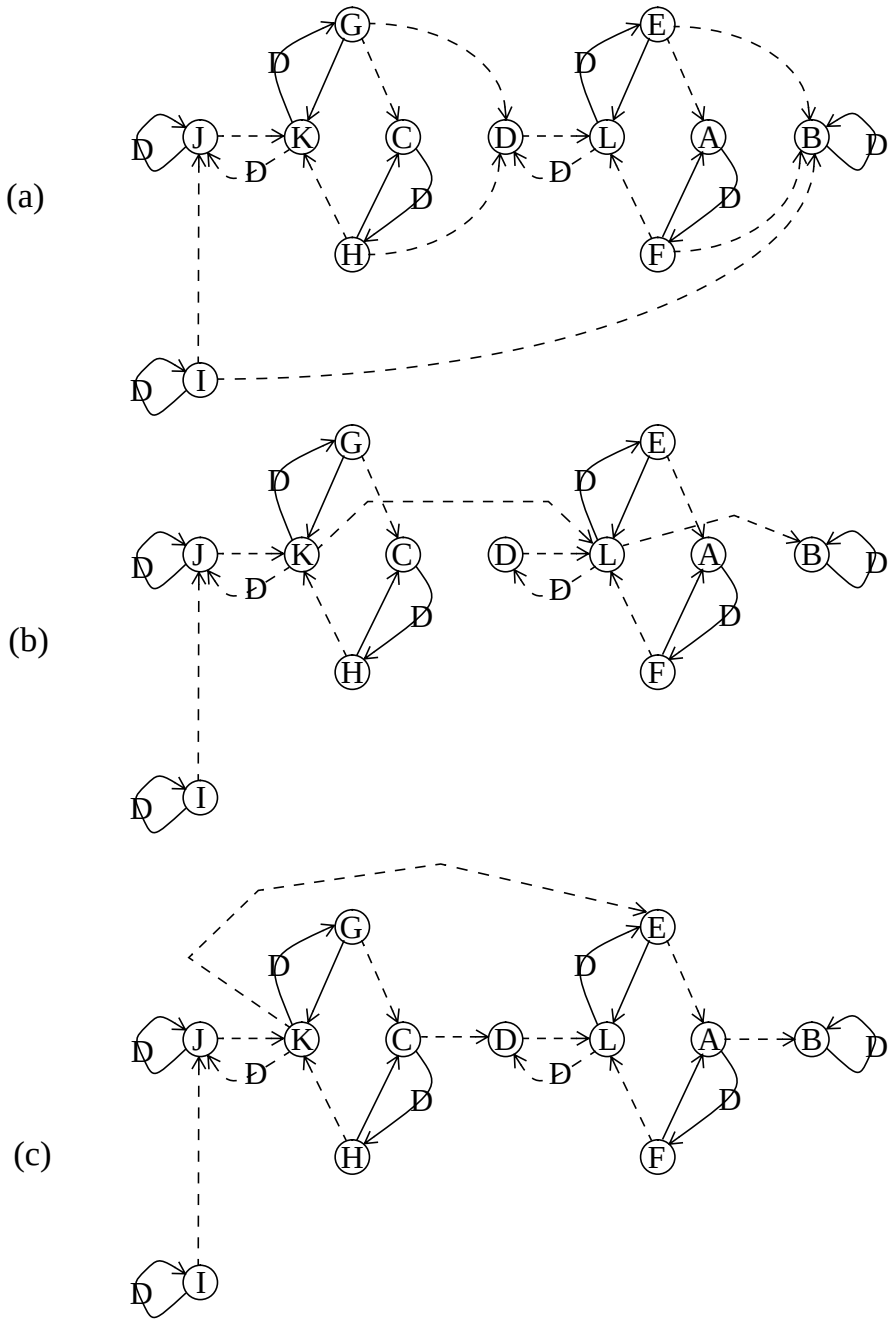


Figure 10. A chainable synchronization graph for which Global-resynchronize fails to produce an optimal solution.

schedule for a music synthesis system. The results demonstrate that the heuristic can efficiently reduce synchronization overhead and improve throughput.

Several useful directions for further work emerge from our study. These include investigating whether efficient techniques can be developed that consider resynchronization opportunities within strongly connected components, rather than just across feedforward edges. There may also be considerable room for improvement over our proposed heuristic, which is a straightforward adaptation of an existing set covering algorithm. In particular, it would be useful to explore ways to best integrate the proposed heuristic for general synchronization graphs with the optimal chaining method for a restricted class of graphs, and it may be interesting to search for properties of practical synchronization graphs that could be exploited in addition to the correspondence with set covering. The extension of Sakar's concept of counting semaphores [34] to self-timed, iterative execution, and the incorporation of extended counting semaphores within our resynchronization framework are also interesting directions for further work.

12. 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.

13. 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.

λ_{max}: The maximum cycle mean of a DFG.

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

$\langle(p_1, p_2, \ldots, p_k)\rangle$: The *concatenation* of the paths $p_1, p_2, \ldots, p_k$.

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

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

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

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.

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

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

$\Psi(R, G)$: 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.

14. 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, *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.

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

[8] S. S. Bhattacharyya, S. Sriram, and E. A. Lee, *Resynchronization of Multiprocessor Schedules — Part 2: Latency-constrained Resynchronization*, Memorandum UCB/ERL M96/56, Electronics Research Laboratory, University of California, Berkeley, October, 1996.

[9] 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.*

[10] 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*, Vol. 4, April, 1994.

[11] 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.

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

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

[14] 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.

[15] 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.

[16] 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.

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

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

[19] D.C. Ku, G. De Micheli, "Relative Scheduling Under Timing Constraints: Algorithms for High-level Synthesis of Digital Circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol.11, No.6, pp. 696-718, June, 1992.

[20] 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.

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

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

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

[24] 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.

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

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

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

[28] 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.

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

[30] 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.

[31] 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.

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

[33] 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.

[34] V. Sarkar, “Synchronization Using Counting Semaphores,” *Proceedings of the International Symposium on Supercomputing*, 1988.

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

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

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

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