Resynchronization for Multiprocessor DSP Systems

Shuvra S. Bhattacharyya, *Member, IEEE*, Sundararajan Sriram, *Member, IEEE*, and Edward A. Lee, *Fellow, IEEE*

***Abstract*—This paper introduces a technique, called resynchronization, for reducing synchronization overhead in multiprocessor implementations of digital signal processing (DSP) systems. The technique applies to arbitrary collections of dedicated, programmable or configurable processors, such as combinations of programmable DSP's, ASICS, and FPGA subsystems. Thus, it is particularly well-suited to the evolving trend toward heterogeneous single-chip multiprocessors in DSP systems. Resynchronization exploits the well-known observation [43] 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 original 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 for iterative dataflow specifications of DSP applications. An iterative dataflow specification consists of a dataflow representation of the body of a loop that is to be iterated indefinitely; dataflow programming in this form has been employed extensively in the DSP domain.

***Index Terms*—Embedded multiprocessors, iterative dataflow graphs, latency, multiprocessor scheduling, pipelining, real-time signal processing, self-timed systems, set covering, shared memory, VLSI signal processing.**

I. Introduction

THIS paper is concerned with implementation of iterative, dataflow-dominated algorithms on embedded multiprocessor systems. In the DSP domain, such multiprocessors typically consist of one or more CPU's and one or more application-specific hardware components. Such embedded multiprocessor systems are becoming increasingly common today in applications ranging from digital audio/video equipment to portable devices such as cellular phones and PDA's. A digital cellular phone, for example, typically consists of a micro-controller, a DSP, and custom ASIC circuitry. With increasing levels of integration, it is now feasible to integrate such heterogeneous systems entirely on a single chip. The design task of such multiprocessor systems-on-a-chip is complex, and the complexity will only increase in the future.

A critical issue in the design of embedded multiprocessors is managing communication and synchronization between the heterogeneous processing elements. In this paper, we focus on the problem of minimizing communication and synchronization overhead in embedded multiprocessors. We propose algorithms that automate the process of designing synchronization points in a shared-memory multiprocessor system with the objective of reducing synchronization overhead.

Specifically, we develop 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, and 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 more than the number of new synchronizations.

We study this problem in the context of self-timed execution of iterative *synchronous dataflow* (SDF) specifications, which are SDF representations of computations that are to be repeated indefinitely. In SDF, an application is represented as a directed graph in which vertices **(actors)** represent computational tasks of arbitrary complexity, edges specify data dependences, and the number of data values **(tokens)** produced and consumed by each actor is fixed.

Although the model is too restricted for many general-purpose applications, iterative SDF has proven to be a useful framework for representing a significant class of DSP algorithms, and it has been used as a foundation for numerous DSP design environments [10], [26], [40], [42]. A wide variety of techniques have been developed to schedule SDF specifications for efficient multiprocessor implementation (e.g., [1], [2], [11], [17], [18], [30], [36], [40], [44] and [47]). 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.

Each SDF edge has associated a nonnegative integer *delay*. SDF 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,

Manuscript received August 25, 1998; revised March 6, 2000. The work of S. S. Bhattacharyya was supported in part by the U.S. National Science Foundation (CAREER, MIP9734275). Part of this work was performed as part of the Ptolemy project, which is supported by the Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory, the State of California MICRO program, and the following companies: The Alta Group of Cadence Design Systems, Hewlett Packard, Hitachi, Hughes Space and Communications, NEC, Philips, and Rockwell. This paper was recommended by Associate Editor J. Götze.

S. S. Bhattacharyya is with the Department of Electrical and Computer Engineering, and the Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742 USA (ssb@eng.umd.edu).

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

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

Publisher Item Identifier S 1057-7122(00)09920-7.

1057–7122/00$10.00 © 2000 IEEE

since efficient techniques have been developed to convert general SDF graphs into homogeneous graphs [28], our techniques can easily be adapted to general SDF graphs.

We refer to a homogeneous SDF graph as a **dataflow graph (DFG)**. We represent a DFG by an ordered pair (V, E), where V is the set of 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 is e an **output edge** of $src(e)$, e is an **input edge** of $snk(e)$, and e is **delayless** if $delay(e) = 0$.

Our implementation model involves a *self-timed* scheduling strategy [29]. 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 they may exhibit occasional deviations from their compile-time estimates.

Interprocessor communication **(IPC)** between processors is assumed to take place through shared memory, which could be global memory between all processors, or could be distributed between pairs of processors (for example, hardware first-in-first-out (FIFO) queues or dual ported memory). Such simple communication mechanisms, as opposed to cross bars and elaborate inter-connection networks, are common in embedded systems, owing to their simplicity and low cost.

Synchronization is performed by setting and testing flags in shared memory. For example, in the *BBS* protocol [5] for a dataflow edge e, a write pointer $wr(e)$ for is maintained on the processor that executes $src(e)$, a read pointer $rd(e)$ is maintained on the processor that executes $snk(e)$; and a copy of $wr(e)$ is maintained in some shared memory location $sv(e)$. The pointers $rd(e)$ and $wr(e)$ are initialized to zero and $delay(e)$, respectively. Just after each execution of $src(e)$, the new data value produced onto e is written into the shared memory buffer for e at off-set $wr(e)$, $wr(e)$ is updated by the operation $(wr(e) \leftarrow (wr(e) + 1) \bmod B(e))$, where $B(e)$ is the buffer size associated with e, and $sv(e)$ is updated to contain the new value of $wr(e)$. Just before each execution of $snk(e)$, the value contained in $sv(e)$ is repeatedly examined (with interleaved periods of "backoff" from the shared bus) until it is found to be not equal to $rd(e)$; then the data value residing at offset $rd(e)$ of the shared memory buffer for e is read and $rd(e)$ is updated by the operation $(rd(e) \leftarrow (rd(e) + 1) \bmod B(e))$.

Similarly, 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. Synchronization of such interfaces 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 [20]. 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 Sections IV and V, we address the problem of resynchronization under the assumption that a relatively large increase in latency is acceptable. Such a scenario arises when the computations occur in a feedforward manner, e.g., audio/video decoding for playback from media such as DVD (Digital Video Disk), and also for a wide variety of simulation applications. Sections VI–VIII, on the other hand, examine the relationship between resynchronization and latency, and address the problem of optimal resynchronization when only a limited increase in latency is tolerable. Such latency constraints are present in interactive applications such as video conferencing and telephony.

II. Background

A **path** in a directed graph (V, E) is a finite sequence $(e_1, e_2, \cdots, e_n)$, where each e_i is in E, and $snk(e_i) = src(e_{i+1})$, for $i = 1, 2, \cdots, (n-1)$. We say that the path $p = (e_1, e_2, \cdots, e_n)$ **contains** each e_i and each contiguous subsequence of $(e_1, e_2, \cdots, e_n)$; p is **directed from** $src(e_1)$ to $snk(e_n)$ and each member of $\{src(e_1), src(e_2), \cdots, src(e_n), snk(e_n)\}$ 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.

Given a path $p = (e_1, e_2, \cdots, e_n)$, the **path delay** of p, denoted $\mathrm{Delay}(p)$, is given by

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

Since the delays on all DFG edges are restricted to be nonnegative, 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 **minimum-delay path** between x and y. Given a DFG G, and vertices x, y in G, we define $\rho_G(x, y)$ to be equal to ∞ if there is no path from x to y, and equal to the path delay of a minimum-delay path from x to y if there exist one or more paths from x to y. If G is understood, then we may drop the subscript, and simply write "ρ" in place of "ρ_G."

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

III. 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 [45] to study the execution patterns of actors under self-timed evolution, and in [5], 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 nonnegative 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. Fig. 1(c) shows the IPC graph that corresponds to the application DFG of Fig. 1(a, b) and the processor assignment / actor ordering of Fig. 1(a, b).

Each edge e in G_{ipc} represents the **synchronization constraint**

$$start(snk(e), k) \geq end(src(e), k - \text{delay}(e)) \qquad (2)$$

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

A. 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. Details on the operation of synchronization protocols for self-timed dataflow schedules can be found in [5].

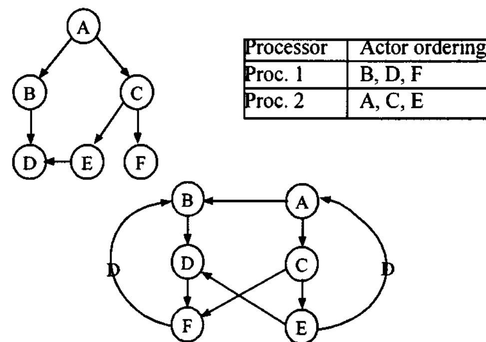

Fig. 1. Part (c) shows the IPC graph that corresponds to the DFG of part (a) and the processor assignment / actor ordering of part (b). A D on top of an edge represents a unit delay.

B. 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 [41], the throughput (number of DFG iterations per unit time) of a synchronization graph G is given by $1/(\lambda_{\max}(G))$, where

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

If the maximum in (3) is infinite, there exists at least one delay free cycle in G, which means that the schedule modeled by the synchronization graph is deadlocked. In the remainder of this paper, we are concerned only with synchronization graphs that result from schedules that are not deadlocked. Thus, we assume the absence of delay-free cycles. In practice, this assumption is not a problem since delay-free cycles can be detected efficiently [22].

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 whose cycle mean is equal to the maximum cycle mean is called a **critical cycle**. 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.

C. Preservation and Subsumption in 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 $\rho_{G_1}(src(e), snk(e)) \leq \text{delay}(e)$.

Theorem 1 [5]: 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. The synchronization graph G is **reduced** if G contains no redundant synchronization edges. For example, in Fig. 1(c), the synchronization edge (C, F) is redundant due to the path $((C, E), (E, D), (D, F))$.

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 $\rho(x_1, y_1) + \rho(y_2, x_2) \leq \text{delay}((x_1, x_2))$. 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.

Given an ordered pair (v_1, v_2) of actors, the set of synchronization edges that are subsumed by (v_1, v_2) is denoted $\chi((v_1, v_2))$.

IV. 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). Fig. 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 Fig. 2(b), we show the synchronization graph that results from inserting the *resynchronization edge* $d_0(C, H)$ into Fig. 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 SCC's is also possible; however, in general, such resynchronization may increase the estimated throughput (see Theorem 2). Thus, for our objectives, it must be verified that each new synchronization edge introduced in an SCC does not decrease

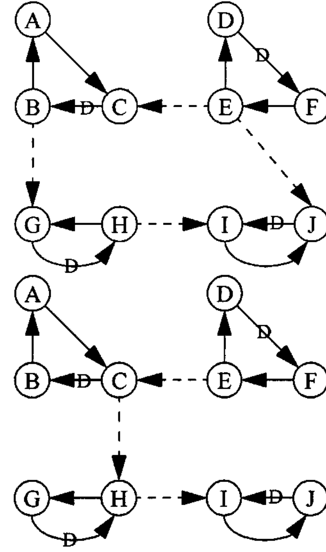

Fig. 2. An example of resynchronization.

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 [27]) *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.

Opportunities for feedforward resynchronization are particularly abundant in the dedicated hardware implementation of dataflow graphs. If each actor is mapped to a separate piece of hardware, as in the VLSI dataflow arrays of Kung *et al.* [25], then for any application graph that is acyclic, every communication channel between two units will have an associated feedforward synchronization edge. Feedforward synchronization edges also arise naturally in multiprocessor software implementations as well. A software example of a music synthesis application is presented in detail in Section VIII.

Definition 1: Suppose that $G = (V, E)$ is a synchronization graph, and $F \equiv \{e_1, e_2, \cdots, e_n\}$ is the set of all feedforward edges in G. A **resynchronization** of G is a finite set $R \equiv \{e'_1, e'_2, \cdots, e'_m\}$ of edges that are not necessarily contained in E, but whose source and sink vertices are in V, such that a) $e'_1, e'_2, \cdots, e'_m$ are feedforward edges in the DFG $G^* \equiv (V, ((E - F) + R))$; and b) G^* preserves G—that is, $\rho_{G^*}(src(e_i), snk(e_i)) \leq \text{delay}(e_i)$ for all $i \in \{1, 2, \cdots, n\}$. 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 Fig. 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; Fig. 2(b) shows the DFG $G^* = (V, ((E - F) + R))$; and from Fig. 2(b), it is easily verified that F, R, and G^* satisfy conditions (a) and (b) of Definition 1.

Lemma 1 [7]: 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 $\text{Delay}(p') \leq \text{Delay}(p)$, 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).

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) $\text{Delay}(C') \leq \text{Delay}(C)$ 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

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

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 $\lambda_{\max}(G') \geq \lambda_{\max}(G)$. Q.E.D.

Thus, 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.

We refer to the problem of finding a resynchronization with the fewest number of elements as the **maximum-throughput resynchronization problem**, or simply, the **resynchronization problem**. In [7], we show that the resynchronization problem is NP-hard by deriving a reduction from the classic set *covering problem* [13], which is a well-known NP-hard problem.

V. 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.

Definition 2: 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 feed-forward 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.

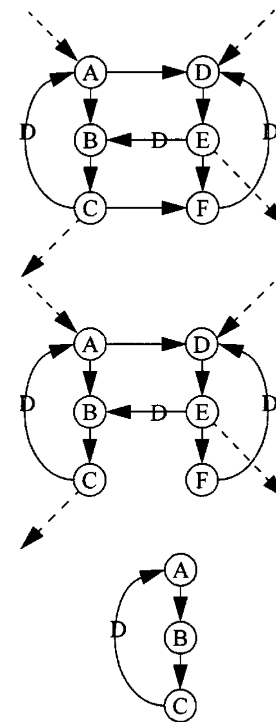

Fig. 3. An illustration of input and output hubs for synchronization graph SCC's.

For example, consider the SCC in Fig. 3(a), and assume that the dashed edges represent the synchronization edges that connect this SCC with other SCC's. This SCC has exactly one input hub, actor A, and exactly one output hub, actor F, and since $\rho(A, F) = 0$, it follows that the SCC is linkable. However, if we remove the edge (C, F), then the resulting graph [shown in Fig. 3(b)] is not linkable since it does not have an output hub. A class of linkable SCC's that occur commonly in practical synchronization graphs are those SCC's that correspond to single-processor subsystems, such as the SCC shown in Fig. 3(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 make 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 SCC's, 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 SCC's C_1', C_2' of the form $((o_1, i_2), (o_2, i_3), \cdots, (o_{n-1}, i_n))$ 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 SCC's $Z_1, Z_2, \cdots, Z_{n-2} \notin \{C_1', C_2'\}$ such that for $k = 2, 3, \cdots, (n-1)$, 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 SCC's, 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 $i = 1, 2, \cdots, (n-1)$, where n is the total number of SCC's, and then removing all of the redundant synchronization edges that result.

This chaining technique can be viewed as a generalized 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 in multiprocessor DSP implementations (see for example, [2], [18] and [35]). 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 chaining technique defined above 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 SCC's 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 SCC's, 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 SCC's. If there are no internal SCC's, then the sink SCC's 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 SCC's.

VI. Resynchronization and Latency

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. In this and the following two sections, we address the problem of computing an optimal resynchronization among all resynchronizations that do not increase the latency beyond a prespecified upper bound $L_{\max}$. This enables us to realize some of the benefits of reduced synchronization overhead due to resynchronization, while maintaining the required latency constraint.

Definition 3: Suppose G_0 is an application DFG, G is a synchronization graph that results from a multiprocessor schedule for G_0, x is an execution source (an actor that has no input edges or has nonzero delay on all input edges) in G, and y is an actor in G other than x. We define the **latency** from x to y by $L_G(x, y) \equiv \text{end}(y, 1 + \rho_{G_0}(x, y))$. 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, and thus, the latency corresponds to the critical path in the dataflow implementation to the first output invocation that is influenced by the input. This interpretation of the latency as the critical path is widely used in VLSI signal processing [24], [32].

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

$$\max(\{v_1, v_2, \cdots, v_n\}) + t(z) \tag{5}$$

where $\{\{v_1, v_2, \cdots, v_n\}\}$ is the set of token values consumed during the actor execution. In such a simulation, the ith token value produced by an actor z gives the completion time of the ith invocation of z in the ASAP execution of G. Thus, the latency can be determined as the value of the $(1 + \rho_{G_0}(x, y))$th output token produced by y. With careful implementation of the functional simulator described above, the latency can be determined in $O(d \times \max(\{|V|, s\}))$ time, where $d = 1 + \rho_{G_0}(x, y)$, and s denotes the number of synchronization edges in G. The simulation approach described above is similar to approaches described in [46].

For a broad class of synchronization graphs, latency can be analyzed even more efficiently during resynchronization. This is the class of synchronization graphs in which the first invocation of the latency output is influenced by the first invocation of the latency input. Equivalently, it is the class of graphs that contain at least one delayless path in the corresponding application DFG directed from the latency input to the latency output. For this class of synchronization graphs, we can directly apply well-known longest-path based techniques for computing latency.

Definition 4: 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

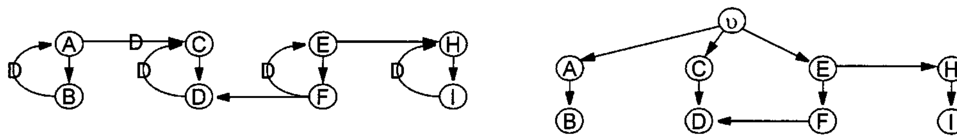

Fig. 4. An example used to illustrate the construction of $fi(G)$. The graph on the right is $fi(G)$ if G is the left-side graph.

graph, which we call the **first-iteration graph** of G, denoted $fi(G)$, is constructed by removing all edges from G that have nonzero-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. Fig. 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 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

$$T_{fi(G)}(x, y) = \max\left(\sum_{p \text{ traverses } z} t(z) \middle| (p \text{ is a path from } x \text{ to } y \text{ in} fi(G)) \right). \quad (6)$$

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 [13].

The following theorem gives an efficient means for computing the latency L_G for transparent synchronization graphs. A straightforward proof based on induction can be found in [8].

Theorem 3: Suppose that G is a synchronization graph that is transparent with respect to latency input x and latency output y. Then $L_G(x, y) = T_{fi(G)}(\upsilon, y)$.

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

Definition 5: An instance of the **latency-constrained resynchronization problem** consists of a synchronization graph G with latency input x and latency output y, and a *latency constraint* $L_{\max} \geq L_G(x, y)$. A solution to such an instance is a resynchronization R such that 1) $L_{\Psi(R,G)}(x, y) \leq L_{\max}$, and 2) no resynchronization of G that results in a latency less than or equal to $L_{\max}$ has smaller cardinality than R.

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

We have established that the latency-constrained resynchronization problem is NP-hard even for the very restricted subclass of synchronization graphs in which every synchronization graph is transparent, each SCC corresponds to a single actor, and all synchronization edges have zero delay [8]. As with the maximum-throughput resynchronization problem, the intractability of this special case of latency-constrained resynchronization can be established by a reduction from set covering.

VII. Two-Processor Systems

The problem of latency-constrained synchronization for the case where there are only two processors in the system (the **2LCR** problem) is an interesting special case. Although the general LCR problem is NP-hard, the 2LCR problem can be solved in polynomial time. This reveals a pattern of complexity that is somewhat analogous to the classic, nonpreemptive multiprocessor scheduling problem with deterministic execution times [19].

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

An example of an instance of 2LCR is shown in Fig. 5(a). Here, $p = q = 8$ and we assume that $t(z) = 1$ for each actor z, and $L_{\max} = 10$.

In this discussion, we assume that $\tilde{G}$ is transparent and that delay $(s_i) = 0$ for all s_i. We refer to the subproblem that results from these restrictions as **delayless 2LCR**. In this section, we illustrate how delayless 2LCR can be solved in time quadratic in the number of vertices in the synchronization graph. We have

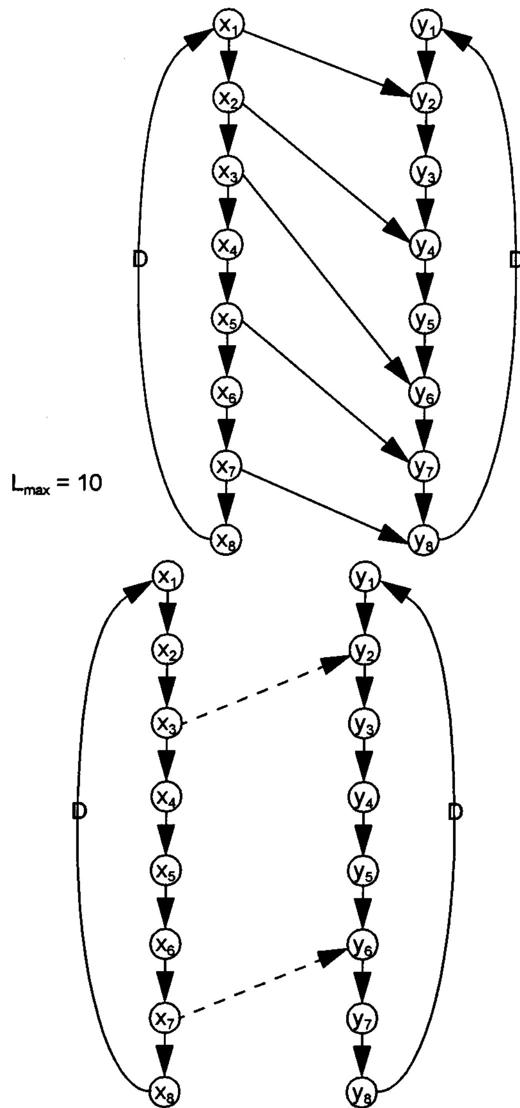

Fig. 5. An instance of two-processor latency-constrained resynchronization. In this example, the execution times of all actors are identically equal to unity.

extended this approach to solve the general (not necessarily delayless) 2LCR problem in cubic time; we refer the reader to [8] for details on this extension and for formal proofs of the optimality of our techniques for delayless 2LCR and general 2LCR.

The delayless 2LCR problem can be reduced to a special case of set covering called **interval covering**, in which we are given an ordering $w_1, w_2, \cdots, w_N$ of the members X of (the set that must be covered) such that the collection of subsets consists entirely of subsets of the form $\{w_a, w_{a+1}, \cdots, w_b\}$, $1 \leq a \leq b \leq N$. Thus, while general set covering involves covering a set from a collection of subsets, interval covering amounts to covering an interval from a collection of subintervals. Interval covering can be solved in $O(|X||T|)$ time using a straightforward approach [8].

Our algorithm for the 2LCR problem is based on the following result.

Theorem 4 [8]: If R is a resynchronization of $\tilde{G}$, then

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

where

$$t_{\text{pred}}(x_i) \equiv \sum_{j \leq i} t(x_j) \quad \text{for} \quad i = 1, 2, \cdots, p,$$

and

$$t_{\text{succ}}(y_i) \equiv \sum_{j \geq i} t(y_j) \quad \text{for} \quad i = 1, 2, \cdots, q.$$

The set X in the interval covering instance that we derive from $\tilde{G}$ is the set $\{s_1, s_2, \cdots, s_n\}$ of synchronization edges in $\tilde{G}$. To derive the interval covering instance, we start by ordering the synchronization edges according to the order in which the source actors execute on the source processor. This ordering, denoted $(s_1', s_2', \cdots, s_n')$, is specified by

$$(x_a = src(s_i'), x_b = src(s_j'), a < b) \Rightarrow (i < j). \tag{7}$$

Next, we define X_0 to be the set of the source processor actors x_i that satisfy $t_{\text{pred}}(x_i) + t(y_q) \leq L_{\max}$, and for each i such that $x_i \in X_0$, we define an ordered pair of actors (a "resynchronization candidate") by

$$v_i \equiv (x_i, y_j), \quad \text{where} \quad j = \min(\{k | (t_{\text{pred}}(x_i) + t_{\text{succ}}(y_k) \leq L_{\max})\}). \tag{8}$$

Consider the example shown in Fig. 5(a) (recall that for this example, we assume that $t(z) = 1$ for each actor z, and $L_{\max} = 10$). Here, $X_0 = \{x_1, x_2, \cdots, x_8\}$, and from (8), we have

$$\begin{aligned} v_1 &= (x_1, y_1), \quad v_2 = (x_2, y_1), \quad v_3 = (x_3, y_2), \\ v_4 &= (x_4, y_3), \quad v_5 = (x_5, y_4), \quad v_6 = (x_6, y_5), \\ v_7 &= (x_7, y_6), \quad v_8 = (x_8, y_7). \end{aligned} \tag{9}$$

The set T of "interval" subsets of $\{s_1, s_2, \cdots, s_n\}$ to be covered is then computed as

$$T = \{\chi(v_i) | x_i \in X_0\}. \tag{10}$$

In [8] we show that the family of subsets defined by (10) together with the ordering specified by (7) always forms an instance of interval covering, and that given a solution (minimal cover) $\{\chi(v_{r_1}), \chi(v_{r_2}), \cdots, \chi(v_{r_z})\}$ to this instance of interval covering, $R = \{v_{r_1}, v_{r_2}, \cdots, v_{r_z}\}$ is an optimal latency-constrained resynchronization of $\tilde{G}$.

For Fig. 5(a), the ordering specified by (7) is

$$\begin{aligned} s_1' &= (x_1, y_2), \quad s_2' = (x_2, y_4), \quad s_3' = (x_3, y_6), \\ s_4' &= (x_5, y_7), \quad s_5' = (x_7, y_8) \end{aligned} \tag{11}$$

and thus from (9), we have

$$\begin{aligned} &\chi(v_1) = \{s_1'\}, \quad \chi(v_2) = \{s_1', s_2'\}, \quad \chi(v_3) = \{s_1', s_2', s_3'\}, \\ &\chi(v_4) = \{s_2', s_3'\}, \quad \chi(v_5) = \{s_2', s_3', s_4'\}, \quad \chi(v_6) = \{s_3', s_4'\}, \\ &\chi(v_7) = \{s_3', s_4', s_5'\}, \quad \chi(v_8) = \{s_4', s_5'\}. \end{aligned} \tag{12}$$

It is easily verified that $C = \{\chi(v_3), \chi(v_7)\}$ is a minimal cover for $\{s_1, s_2, \cdots, s_n\}$ from the family of subsets specified by (12). Thus, we are guaranteed that the resynchronization $R = \{v_3, v_7\}$ is an optimal latency-constrained resynchronization of Fig. 5(a). The synchronization graph that results from this resynchronization is shown in Fig. 5(b).

```
function Resynchronize
input: a synchronization graph G = (V, E) , and a latency constraint L_max ∈ ({0, 1, ...} ∪ {∞}).
output: an alternative reduced synchronization graph that preserves G.

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

Fig. 6. A heuristic for latency-constrained resynchronization.

VIII. A Heuristic for General Synchronization Graphs

In this section, we present a general heuristic for resynchronization called Algorithm *Resynchronize* that exploits the correspondence to set covering described in Sections IV and VI. Algorithm *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 [21] and [31] 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 of elements to be covered to be the set of feedforward synchronization edges, and choosing the family of subsets to be

$$T \equiv \{\chi(v_1, v_2) | ((v_1, v_2) \notin E) \quad \textbf{and} \quad (\rho_G(v_2, v_1) = \infty) \quad \textbf{and} \quad (L'(v_1, v_2) \le L_{\max}))\} \tag{13}$$

where

$L_{\max}$ is the maximum tolerable latency,

$G = (V, E)$ is the input synchronization graph, and

$L'(v_1, v_2)$ 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.

The constraint $\rho_G(v_2, v_1) = \infty$ in (13) ensures that inserting the edge (v_1, v_2) does not introduce a cycle, and thus, that it neither introduces deadlock nor reduces the estimated throughput. If $L_{\max} = \infty$, then the algorithm effectively attempts to compute an efficient maximum-throughput resynchronization of G; otherwise, the algorithm computes a latency-constrained resynchronization whose latency is no greater than $L_{\max}$.

Algorithm *Resynchronize* assumes that the input synchronization graph is reduced (e.g., from the redundant synchronization removal technique of [5]). The algorithm determines the family of subsets specified by (13), 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 (13) produces the empty set. Fig. 6 gives a pseudocode specification of this algorithm.

```
for x, y ∈ (V ∪ {υ})            /* update T_fi(G) */
    T_new(x, y) = max({ T_fi(G)(x, y), T_fi(G)(x, src(best)) + T_fi(G)(snk(best), y) })
end for
T_fi(G) = T_new
```

Fig. 7. Pseudocode to update $T_{fi(G)}$ for use in the customization of Algorithm Resynchronize to transparent synchronization graphs.

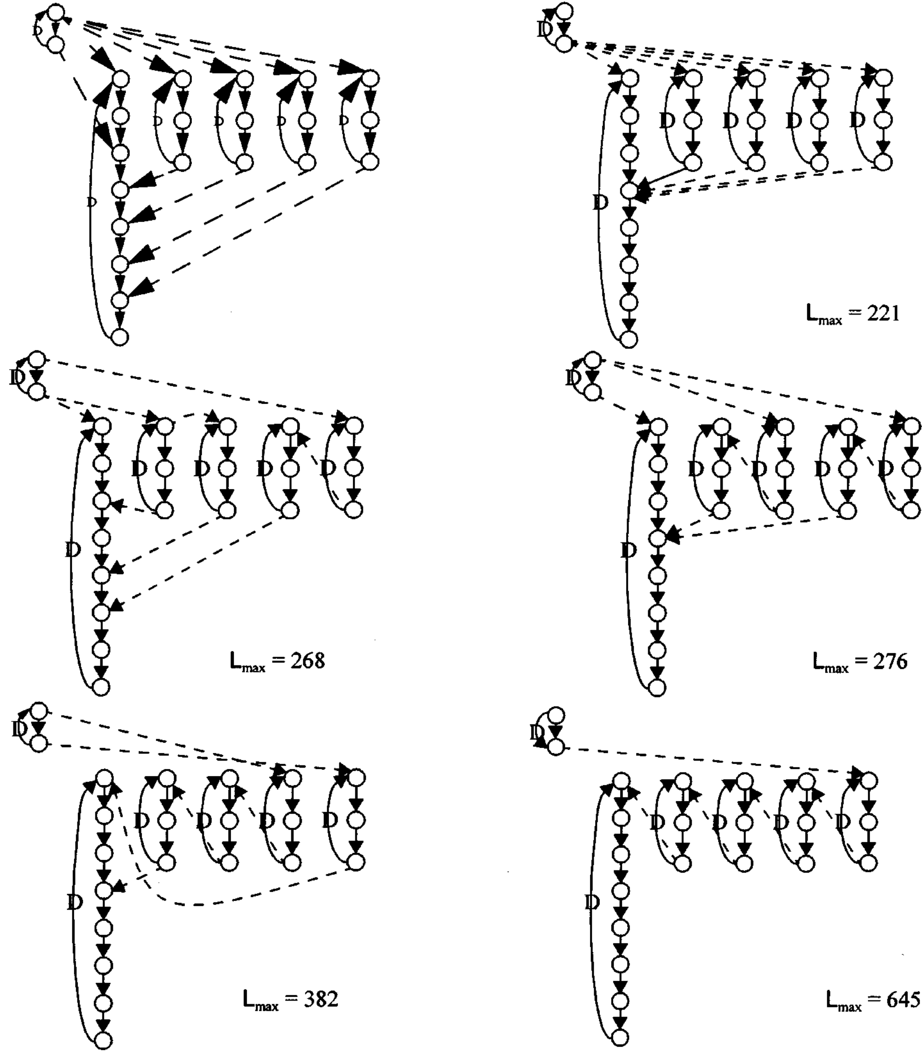

Fig. 8. Synchronization graphs computed by Algorithm *Resynchronize* on a music synthesis example for different values of $L_{\max}$.

A. Latency Computation and Algorithm Complexity

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

$$L'(v_1, v_2) = \max(\{(T_{fi(G)}(\upsilon, v_1) + T_{fi(G)}(v_2, o_L)), L_G\}) \tag{14}$$

where

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

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

$$T_{\text{new}}(x, y) = \max(\{T_{fi(G)}(x, y), T_{fi(G)}(x, src(\text{best})) + T_{fi(G)}(snk(\text{best}), y)\}) \tag{15}$$

where T_{new} denotes the maximum cumulative execution time between actors in the first iteration graph after the insertion of the edge *best* in G. The computations in (15) can be performed by inserting the simple **for** loop shown in Fig. 7 at the end of the **else** block in Algorithm *Resynchronize*. Thus, as with the computation of ρ_G, the Bellman–Ford algorithm need only be invoked once, at the beginning of Algorithm *Resynchronize*, to initialize $T_{fi(G)}(x, y)$. This loop can be inserted immediately before or after the **for** loop that updates ρ_G.

TABLE I
PERFORMANCE RESULTS FOR THE RESYNCHRONIZATIONS OF FIG. 8. THE FIRST COLUMN GIVES THE MEMORY ACCESS TIME; "IP" STANDS FOR "AVERAGE ITERATION PERIOD"; AND "A/P" STANDS FOR "MEMORY ACCESSES PER GRAPH ITERATION"

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

When the algorithm is customized to transparent synchronization graphs in the manner described above, the time-complexity of Algorithm *Resynchronize* is $O(s_{ff}n^4)$, where n is the number of actors in the input synchronization graph G, and s_{ff} is the number of feedforward synchronization edges [8]. For general (not necessarily transparent) synchronization graphs, we can use the functional simulation approach described in Section VI to determine $L'(x, y)$. This yields a running time of $O(ds_{ff}n^4 \max(\{n, s\}))$ for Algorithm *Resynchronize* on general synchronization graphs [8], where s is the number of synchronization edges in G, and $d = 1 + \rho_{G_0}(x, y)$.

B. Example

Fig. 8(a) 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. There are ten synchronization edges shown, and none of these is redundant. Fig. 8(b)–(f) show how the number and placement of synchronization edges in the result computed by Algorithm *Resynchronize* change 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 large 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.

Table I shows how the average iteration period (the reciprocal of the average throughput) varies with different memory access times for the various resynchronizations of Fig. 8. Here, the columns labeled A-F respectively represent the resynchronizations depicted in Fig. 8(a)–(f). Thus, as we go from column "A" to column "F," the number of synchronization edges in the resynchronized solution decreases monotonically. However, as seen in Table I, the average iteration period need not exactly follow this trend. For example, even though synchronization graph A has one synchronization edge more than graph B, the iteration period curve for graph B lies slightly above that of A. This is because the simulations shown in the figure model a shared bus, and take bus contention into account. Thus, even though graph B has one less synchronization edge than graph A, it entails higher bus contention, and hence results in a higher average iteration period. A similar anomaly is seen between graph C and graph D. However, we observe such anomalies only within highly localized neighborhoods in which the number of synchronization edges differs by only one. Overall, in a global sense, the figure shows a clear trend of decreasing iteration period with loosening of the latency constraint, and reduction of the number of synchronization edges. Table I also shows a similarly pronounced trend toward reduction in the average rate of shared memory accesses as the number of synchronization edges is reduced. Since shared memory accesses typically consume significant amounts of energy, such reduction in the rate of shared memory accesses is useful in low power applications.

IX. RELATED WORK

Shaffer has developed an algorithm that removes redundant synchronizations in the self-timed execution of a noniterative DFG [43]. 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, *et al.* have studied synchronization rearrangement in the context of minimizing the controller area for hardware synthesis of synchronous digital circuitry [14], [15]. However, due to significant differences in both the scheduling models and the implementation models involved, the techniques developed in [14] and [15] do not extend in any straightforward manner to the resynchronization of synchronization graphs for self-timed multiprocessor implementation, and are significantly different in structure from the methods developed in this paper [7].

Tradeoffs between latency and throughput have been studied by Potkonjac and Srivastava in the context of transformations for dedicated implementation of linear computations [39]. Because this work is based on synchronous implementations, it does not address the synchronization issues and opportunities that we encounter in our self-timed dataflow context.

Preliminary versions of the material in this paper have been summarized in [4] and [6].

X. Summary

The goal of resynchronization is to introduce new synchronizations in such a way that the number of original synchronizations that become redundant significantly exceeds the number of new synchronizations. To ensure that the serialization imposed by resynchronization does not degrade the throughput, the new synchronizations are restricted to lie outside of all cycles. We have shown that even in the absence of latency constraints (*maximum-throughput resynchronization*), optimal resynchronization is intractable. However, we have defined a broad class of systems for which optimal, maximum-throughput resynchronization can be performed in polynomial time.

We have also addressed the problem of latency-constrained resynchronization. Given an upper limit on the allowable latency, the objective of latency-constrained resynchronization is to derive a minimal resynchronization that does not violate this limit. We have established that optimal latency-constrained resynchronization is NP-hard even for a very restricted class of applications; and we have derived an efficient algorithm that computes optimal latency-constrained resynchronizations for two-processor systems.

Additionally, we have presented an effective heuristic framework for maximum-throughput and latency-constrained resynchronization of general systems.

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," in *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 Trans. Signal Processing*, vol. 43, pp. 1468–1484, June 1995.

[3] A. Benveniste and G. Berry, "The synchronous approach to reactive and real-time systems," *Proc. IEEE*, vol. 79, pp. 1270–1282, 1991.

[4] S. S. Bhattacharyya, S. Sriram, and E. A. Lee, "Latency-constrained resynchronization for multiprocessor DSP implementation," in *Proc. Int. Conf. Appl. Specific Syst., Architectures Processors*, Aug. 1996.

[5] ——, "Optimizing synchronization in multiprocessor DSP systems," *IEEE Trans. Signal Processing*, vol. 45, June 1997.

[6] ——, "Self-timed resynchronization: A post-optimization for static multiprocessor schedules," in *Proc. Int. Parallel Processing Symp.*, 1996.

[7] ——, *Resynchronization for Multiprocessor DSP Implementation—Part 1: Maximum-Throughput Resynchronization*. College Park: Digital Signal Processing Lab., Univ. Maryland, July 1998.

[8] ——, *Resynchronization for Multiprocessor DSP Implementation—Part 2: Latency-Constrained Resynchronization*. College Park: Digital Signal Processing Laboratory, Univ. Maryland, July 1998.

[9] S. Borkar *et al.*, "iWarp: An integrated solution to high-speed parallel computing," in *Proc. Supercomputing 1988 Conf.*, Orlando, FL, 1988.

[10] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems," *Int. J. 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," *J. VLSI Signal Processing*, pp. 207–223, 1995.

[12] E. G. Coffman Jr., *Computer and Job Shop Scheduling Theory*. New York: Wiley, 1976.

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

[14] D. Filo, D. C. Ku, C. N. Coelho Jr., and G. De Micheli, "Interface optimization for concurrent systems under timing constraints," *IEEE Trans. Very Large Scale Integration*, vol. 1, Sept. 1993.

[15] D. Filo, D. C. Ku, and G. De Micheli, "Optimizing the control-unit through the resynchronization of operations," *INTEGRATION, VLSI J.*, vol. 13, pp. 231–258, 1992.

[16] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*: W. H. Freeman , 1979.

[17] R. Govindarajan, G. R. Gao, and P. Desai, "Minimizing memory requirements in rate-optimal schedules," in *Proc. Int. Conf. Appl. Specific Array Processors*, San Francisco, CA, August 1994.

[18] P. Hoang, "Compiling real time digital signal processing applications onto multiprocessor systems," Electronics Research Lab., Univ. California , Berkeley, Memor. UCB/ERL M92/68, June 1992.

[19] T. C. Hu, "Parallel sequencing and assembly line problems," *Operations Res.*, vol. 9, 1961.

[20] J. A. Huisken *et al.*, "Synthesis of synchronous communication hardware in a multiprocessor architecture," *J. VLSI Signal Processing*, vol. 6, pp. 289–299, 1993.

[21] D. S. Johnson, "Approximation algorithms for combinatorial problems," *J. Computer Syst. Sciences*, vol. 9, pp. 256–278, 1974.

[22] R. Karp, "A note on the characterization of the minimum cycle mean in a digraph," *Discrete Math.*, vol. 23, 1978.

[23] D. C. Ku and G. De Micheli, "Relative scheduling under timing constraints: Algorithms for high-level synthesis of digital circuits," *IEEE Trans. Computer-Aided Design Integrated Circuits Syst.*, vol. 11, pp. 696–718, June 1992.

[24] S. Y. Kung, *VLSI Array Processors*: Prentice Hall, 1988.

[25] S. Y. Kung, P. S. Lewis, and S. C. Lo, "Performance analysis and optimization of VLSI data-flow arrays," *J. Parallel Distributed Computing*, pp. 592–618, 1987.

[26] R. Lauwereins, M. Engels, J. A. Peperstraete, E. Steegmans, and J. Van Ginderdeuren, "GRAPE: A case tool for digital signal parallel processing," *IEEE ASSP Mag.*, vol. 7, April 1990.

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

[28] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous dataflow programs for digital signal processing," *IEEE Trans. Computers*, Feb. 1987.

[29] E. A. Lee and S. Ha, "Scheduling strategies for multiprocessor real-time DSP," in *Proc. Globecom*, Nov. 1989.

[30] G. Liao, G. R. Gao, E. Altman, and V. K. Agarwal, "A Comparative Study of DSP Multiprocessor List Scheduling Heuristics," School of Computer Science, McGill University, Technical Rep., 1993.

[31] L. Lovasz, "On the ratio of optimal integral and fractional covers," *Discrete Math.*, vol. 13, pp. 383–390, 1975.

[32] V. Madisetti, *VLSI Digital Signal Processors*. New York: IEEE Press, 1995.

[33] D. M. Nicol, "Optimal partitioning of random programs across two processors," *IEEE Trans. Computers*, vol. 15, pp. 134–141, Feb. 1989.

[34] D. R. O'Hallaron, "The assign parallel program generator," School of Computer Science, Carnegie Mellon Univ., Memo. CMU-CS-91-141, May 1991.

[35] K. K. Parhi, "High-level algorithm and architecture transformations for DSP synthesis," *J. VLSI Signal Processing*, Jan. 1995.

[36] K. K. Parhi and D. G. Messerschmitt, "Static rate-optimal sheduling of iterative data-flow programs via optimum unfolding," *IEEE Trans. Computers*, vol. 40, Feb. 1991.
[37] J. L. Peterson, *Petri Net Theory and the Modeling of Systems*. Englewood Cliffs, NJ: Prentice-Hall Inc., 1981.
[38] J. Pino, S. Ha, E. A. Lee, and J. T. Buck, "Software synthesis for DSP using Ptolemy," *J. VLSI Signal Processing*, vol. 9, Jan. 1995.
[39] M. Potkonjac and M. B. Srivastava, "Behavioral synthesis of high performance, and low power application specific processors for linear computations," in *Proc. Int. Conf. Appl. Specific Array Processors*, 1994, pp. 45–56.
[40] H. Printz, "Automatic mapping of large signal processing systems to a parallel machine," Ph.D. dissertation, School of Computer Science, Carnegie Mellon Univ., May 1991.
[41] R. Reiter, "Scheduling parallel computations," *J. Association Computing Machinery*, Oct. 1968.
[42] S. Ritz, M. Pankert, and H. Meyr, "High level software synthesis for signal processing systems," in *Proc. Int. Conf. Application Specific Array Processors*, Berkeley, Aug. 1992.
[43] P. L. Shaffer, "Minimization of interprocessor synchronization in multiprocessors with shared and private memory," in *Int. Conf. Parallel Processing*, 1989.
[44] G. C. Sih and E. A. Lee, "Scheduling to account for interprocessor communication within interconnection-constrained processor networks," in *Int. Conf. Parallel Processing*, 1990.
[45] S. Sriram and E. A. Lee, "Statically scheduling communication resources in multiprocessor DSP architectures," in *Proc. Asilomar Conf. Signals, Syst. Computers*, Nov. 1994.
[46] J. Teich, L. Thiele, and E. A. Lee, "Modeling and simulation of heterogeneous real-time systems based on a deterministic discrete event model," in *Proc. Int. Symp. Syst. Synthesis*, 1995, pp. 156–161.
[47] V. Zivojnovic, H. Koerner, and H. Meyr, "Multiprocessor scheduling with a priori node assignment," in *VLSI Signal Processing VII*. New York: IEEE Press, 1994.

Shuvra S. Bhattacharyya (S'87–M'91) received the B.S. degree from the University of Wisconsin at Madison, and the Ph.D. degree from the University of California at Berkeley.

He is an Assistant Professor in the Department of Electrical and Computer Engineering, and the Institute for Advanced Computer Studies, at the University of Maryland, College Park. The coauthor of two books and the author or coauthor of more than 30 refereed technical articles, Dr. Bhattacharyya is a recipient of the NSF Career Award. His research interests center around architectures and computer-aided design for embedded systems. He has held industrial positions as a Researcher at Hitachi, and as a Compiler Developer at Kuck & Associates, and has consulted for industry in the areas of compiler techniques and multiprocessor architectures for embedded systems.

Sundararajan Sriram (S'92–M'95) received a Bachelor of Technology degree in electrical engineering from the Indian Institute of Technology, Kanpur, in 1989, and a Ph.D. in electrical engineering and computer sciences from the University of California at Berkeley in 1995.

In 1993, he spent a summer as an Intern with the VLSI Systems Department at Bell Laboratories in Holmdel, NJ. He is currently a Member, Group Technical Staff in the Wireless Communications Branch at the Digital Signal Processing Solutions R&D Center at Texas Instruments, Dallas, TX. His research interests are in design of algorithms and VLSI architectures for applications in digital signal processing and communications, an area in which he has over 12 publications and over 8 patent applications. He has also co-authored a book on embedded multiprocessors.

Dr. Sriram is a Member of the IEEE Communications and Signal Processing Societies, and is serving as an Associate Editor for the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS—II.

Edward A. Lee (S'80–S'83–M'86–SM'93–F'94) received the bachelors degree (B.S.) from Yale University (1979), the masters degree (S.M.) from MIT (1981), and the Ph.D. degree from U.C. Berkeley (1986).

His is a Professor in the Electrical Engineering and Computer Science Department at U.C. Berkeley. His research interests center on design, modeling, and simulation of embedded, real-time computational systems. He is director of the Ptolemy project at U.C. Berkeley. He is co-author of four books and numerous papers. From 1979–1982 he was a Member of technical staff at Bell Telephone Laboratories in Holmdel, NJ, in the Advanced Data Communications Laboratory. He is a co-founder of BDTI, Inc., where he is currently a Senior Technical Advisor, and has consulted for a number of other companies. He was an NSF Presidential Young Investigator and won the 1997 Frederick Emmons Terman Award for Engineering Education.