— To appear in *Proc. International Parallel Processing Symposium*, 1996 —

Self-Timed Resynchronization: A Post-Optimization for Static Multiprocessor Schedules

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

Abstract

In a shared-memory multiprocessor system, it is possible that certain synchronization operations are redundant — that is, their corresponding sequencing requirements are enforced completely by other synchronizations in the system — and can be eliminated without compromising correctness. This paper addresses the problem of adding new synchronization operations in a multiprocessor implementation in such a way that the number of original synchronizations that consequently become redundant significantly exceeds the number of new synchronizations. We refer to this approach to reducing synchronization overhead as resynchronization. In this paper we formally define the resynchronization problem, we show that optimal resynchronization is NP-hard, and we propose a family of heuristics for this problem. Finally we present a practical example where resynchronization is useful.

1. Motivation

Resynchronization is based on the concept that there can be redundancy in the synchronizations of a multiprocessor implementation. Shaffer showed that the amount of run-time overhead required for synchronization can be reduced significantly by detecting and eliminating redundant synchronizations; an efficient, optimal algorithm was also proposed for this purpose [18], and this algorithm was subsequently extended to handle iterative computations in [2].

The objective of resynchronization is to introduce new synchronizations in such a way that the number of original synchronizations that consequently become redundant is significantly greater that the number of new synchronizations. We formulate and study this problem in the context of self-timed execution of iterative synchronous dataflow (SDF) [12] programs. The resynchronization technique is therefore significantly different from the techniques presented in [2, 18], which focus on removing redundant synchronization points without considering the addition of new synchronizations. 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 studied and applied extensively, particularly in the context of signal processing software. Self-timed execution refers to a combined compile-time/run-time scheduling strategy in which processors synchronize with one another only based on interprocessor communication (IPC) requirements, and thus, synchronization of processors at the end of each loop iteration does not generally occur [13].

SDF 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 graphical DSP design environments, in which signal processing applications are represented as hierarchies of block diagrams. Examples of commercial tools based on SDF are the Signal Processing Worksystem (SPW) [17], and COSSAP [21]. Tools developed at various universities that use SDF and related models include Ptolemy [16], the

This research was partially funded as part of the Ptolemy project, which is supported by the Advanced Research Projects Agency and the U.S. Air Force (under the RASSP program, contract F33615-93-C-1317), the Semiconductor Research Corporation (project 94-DC-008), the National Science Foundation (MIP-9201605), the State of California MICRO program, and the following companies: Bellcore, Bell Northern Research, Dolby Laboratories, Hitachi, LG Electronics, Mentor Graphics, Mitsubishi, Motorola, NEC, Pacific Bell, Philips, and Rockwell.

S. S. Bhattacharyya is with the Semiconductor Research Laboratory, Hitachi America, Ltd., 201 East Tasman Drive, San Jose, CA 95134, USA, shuvra@halsrl.com, fax: (408)954-8907.

S. Sriram is with the DSP R&D Centre at Texas Instruments, MS446, 13510 North Central Expressway, Dallas, TX75265-5474, USA, sriram@hc.ti.com, fax: (214)995-6194.

E. A. Lee is with the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, CA 94720, USA, eal@eecs.berkeley.edu, fax: (510)642-2739.

Warp compiler [20], DESCARTES [21], GRAPE [11], and the Graph Compiler [23].

An important property of SDF graphs is that it is possible to determine efficient schedules for such graphs at compile time. A number of techniques have been proposed for statically scheduling SDF programs for efficient multiprocessor implementation. Parhi and Messerschmitt [15], and Chao and Sha [4] have developed systematic techniques for exploiting overlapped execution to generate schedules that have optimal throughput, assuming zero cost for IPC. Other work has focused on taking IPC costs into account during scheduling, such as that described in [1, 19, 20, 24]; these efforts have not attempted to exploit overlapped execution of graph iterations. Similarly, in [8], Govindarajan and Gao develop techniques to simultaneously maximize throughput, possibly using overlapped execution, and minimize buffer memory requirements under the assumption of zero IPC cost. Our work 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. *Delays* on SDF edges represent initial tokens, and specify dependencies between iterations of the actors in iterative execution. For example, if tokens produced by the kth invocation of actor A are consumed by the $(k+2)$ th invocation of actor B, then the edge (A, B) contains two delays. Tasks can be of arbitrary complexity. In DSP design environments, they typically range in complexity from basic operations such as addition or subtraction to signal processing subsystems such as FFT units and adaptive filters. 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 [12], 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 [13]. 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**) 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. Sender-receiver synchronization is also assumed to take place by setting and checking flags in shared memory (see [1] for details on the assumed synchronization protocols). Thus, effective resynchronization results in a significantly reduced rate of accesses to shared memory for the purpose of synchronization.

Resynchronization has been studied earlier in the context of hardware synthesis [7]. However in this work, the scheduling model and implementation model are significantly different from the structure of self-timed multiprocessor implementations, and as a consequence, the analysis techniques and algorithmic solutions do not apply to our context, and vice-versa [3].

2. Analysis of self-timed execution

A *strongly connected component* (*SCC*) of a directed graph is a maximal subgraph in which there is a path from each vertex to every other vertex. A *feedforward edge* is an edge that is not contained in an SCC. The source and sink actors of an SDF edge e are denoted $src(e)$ and $snk(e)$, and the delay on e is denoted $delay(e)$. An edge e is a *self loop edge* if $src(e) = snk(e)$. An SCC C is a *source SCC* if there does not exist any edge e such that $snk(e)$ is in C and $src(e)$ is not in C. Similarly, C is a *sink SCC* if there does not exist an edge e such that $src(e)$ is in C and $snk(e)$ is not in C.

For each task v in a given 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 DFG 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 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**

$$start(v_i, k) \geq end(v_j, k - delay((v_j, v_i))) , \qquad \textbf{(EQ 1)}$$

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

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 in certain "valid" ways (defined shortly) by adding and deleting edges. At the end of our optimizations, the synchronization graph may look very different from the IPC graph: it is of the form $(V, (E_{ipc} - F + F'))$, where F is the set of original synchronization edges that were deleted, and F' is the set of new synchronization edges that are were introduced. At this point the IPC edges in G_{ipc} represent buffer activity, and must be implemented as buffers in shared memory, whereas the synchronization edges represent synchronization constraints, and are implemented 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 so as to ensure sender-receiver synchronization. On the other hand, if there is an IPC edge between two actors in the IPC graph, but there is no synchronization edge between the two, then no synchronization needs to be done before accessing the shared buffer. If there is a synchronization edge between two actors but no IPC edge, then no shared buffer is allocated between the two actors; only the corresponding synchronization protocol is invoked.

If the execution time of each actor v is a fixed constant $t^*(v)$ for all invocations of v, and the time required for IPC is ignored (assumed to be zero), then as a consequence of Reiter's analysis in [26], the throughput (number of DFG iterations per unit time) of a synchronization graph G is given by $1/\lambda_{max}$, where

$$\lambda_{max} = \max_{\text{cycle } C \text{ in } G} \left\{ \frac{\sum_{v \in C} t^*(v)}{\Delta(C)} \right\}, \qquad \textbf{(EQ 2)}$$

where $\Delta(C)$ is the sum of the delays of all edges that are traversed by the cycle C [1].

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 (2) to obtain the *estimated iteration period* $\tilde{\lambda}_{max}$. In the transformations that we present in this paper, we ensure that we do not alter the **estimated throughput** $1/\tilde{\lambda}_{max}$. Thus, our objective in this paper 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.

All transformations that we perform on the synchronization graph must respect the synchronization constraints implied by G_{ipc}. If we ensure this, then we only need to implement the synchronization edges of the optimized synchronization graph. If $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$ are synchronization graphs with the same vertex-set and the same set of intraprocessor edges (edges that are not synchronization edges), we say that G_1 **preserves** G_2 if for all $e \in E_2$ such that $e \notin E_1$, we have $\rho_{G_1}(src(e), snk(e)) \le delay(e)$, where $\rho_G(x, y) \equiv \infty$ if there is no path from x to y in the synchronization graph G, and if there is a path from x to y, then $\rho_G(x, y)$ is the minimum over all paths p directed from x to y of the sum of the edge delays on p. The $\rho_G(x, y)$ values can be computed efficiently using Djkstra's all pairs shortest path algorithm in $O(|V||E|)$ time. The following theorem, which is developed in [1], underlies the validity of resynchronization.

Theorem 1: The synchronization constraints (as specified by (1)) 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, because the path $((C, E), (E, D), (D, F))$ implicitly enforces the synchronization constraint specified by the edge (C, F). In [1], it is shown that if all redundant edges in a synchronization graph are removed, then the resulting graph preserves the original synchronization graph, and an efficient algorithm is given for determining the redundant synchronization edges of a synchronization graph. This algorithm is an extension to iterative dataflow programs of an earlier algorithm developed by Shaffer [18].

We conclude this section with a number of definitions. An **execution source** of a synchronization graph is any actor that either has no input edges, or has nonzero delay on each input edge. Given a synchronization graph G, a synchronization edge (x_1, x_2) in G, and an ordered pair of actors (y_1, y_2) in G, we say that (y_1, y_2) **subsumes** (x_1, x_2) in G if

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

Thus, (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.

3. Resynchronization in self-timed systems

Definition 1 formalizes our concept of resynchronization. This considers resynchronization only “across” feedforward edges. We impose this restriction so that the serialization imposed by resynchronization does not degrade the estimated throughput [3].

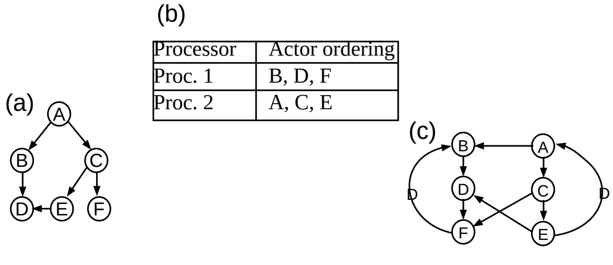
Definition 1: Suppose $G = (V, E)$ is a synchronization graph and F is the set of feedforward edges in G. A **resynchronization** of G is a set $R \equiv \{e_1', e_2', \ldots, e_m'\}$ of edges that are not necessarily contained in E, but whose source and sink vertices are in V, such that $e_1', e_2', \ldots, e_m'$ are feedforward edges in the DFG $G^* \equiv (V, (E - F) + R)$, and G^* preserves G. Each member of R that is not in E is a **resynchronization edge**, G^* is called the **resynchronized graph** associated with R, and this graph is denoted by $R(G)$.

For example $R = \{(E, B)\}$ is a resynchronization of the synchronization graph shown in Figure 1(c), since (E, B) subsumes all the synchronization edges $\{(A, B), (C, F), (E, D)\}$.

The **resynchronization problem** is the problem of finding a resynchronization that has minimal cardinality. In [1], we prove that the resynchronization problem is NP-hard. 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* [14], 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.



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

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.

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 edge p from a vertex in C_1 to a vertex of C_2, we associate the subset of F defined by

$$\chi(p) \equiv \{e \in F \mid (p\ subsumes\ e)\} .$$

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. The sets $\chi(p)$ can be found using Theorem 1, and applying the shortest paths based technique of [2] to find redundant edges in a worst case complexity of $O(|V|^3)$. Clearly then, $\{e_1', e_2', \ldots, e_n'\}$ is a resynchronization if and only if each $(e \in F)$ is contained in at least one $\chi((src(e_i'), snk(e_i')))$ — that is, if and only if $\{\chi((src(e_i'), snk(e_i'))) \mid 1 \le i \le n\}$ covers F. Thus, solving the pairwise resynchronization problem for G is equivalent to finding a minimal cover for F given the family of subsets $\{\chi(x, y) \mid (x \in C_1, y \in C_2)\}$.

Figure 2 helps to illustrate this intuition and our method for converting an instance of the set covering problem to an instance of pairwise resynchronization. Suppose that we are given the set $X = \{x_1, x_2, x_3, x_4\}$, and the family of subsets $T = \{t_1, t_2, t_3\}$, where $t_1 = \{x_1, x_3\}$, $t_2 = \{x_1, x_2\}$, and $t_3 = \{x_2, x_4\}$. To construct an instance of the pairwise resynchronization problem, we first create two vertices and an edge directed between these vertices *for each* member of X; we label each of the edges created in this step with the corresponding member of X. Then for each $t \in T$, we create two vertices $vsrc(t)$ and $vsnk(t)$. Next, for each relation $x_i \in t_j$ (there are six such relations in this example), we create two zero-delay edges — one directed from the 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 $(vsrc(t_j), vsnk(t_j))$ preserve exactly those edges that correspond to members of t_j; in other words, after this construction, $\chi((vsrc(t_j), vsnk(t_j))) = t_j$, for each j. 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 2 shows the synchronization graph that results from this construction process. Here, it is assumed that each vertex corresponds to a separate processor; the associated unit delay, self loop edges are not shown to avoid excessive clutter. Observe that the graph contains two SCCs — $(\{src(x_i)\} \cup \{vsrc(t_i)\})$ and $(\{snk(x_i)\} \cup \{vsnk(t_i)\})$ — and that the set of feedforward edges is the set of edges that correspond to members of X. Now, recall that a major correspondence between the given instance of set covering and the instance of pairwise resynchronization defined by Figure 2(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 2(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 straightforward, albeit tedious, to verify that the resynchronization illustrated in Figure 2(b), $\{(vsrc(t_1), vsnk(t_1)), (vsrc(t_3), vsnk(t_3))\}$, is a minimal resynchronization of Figure 2(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.

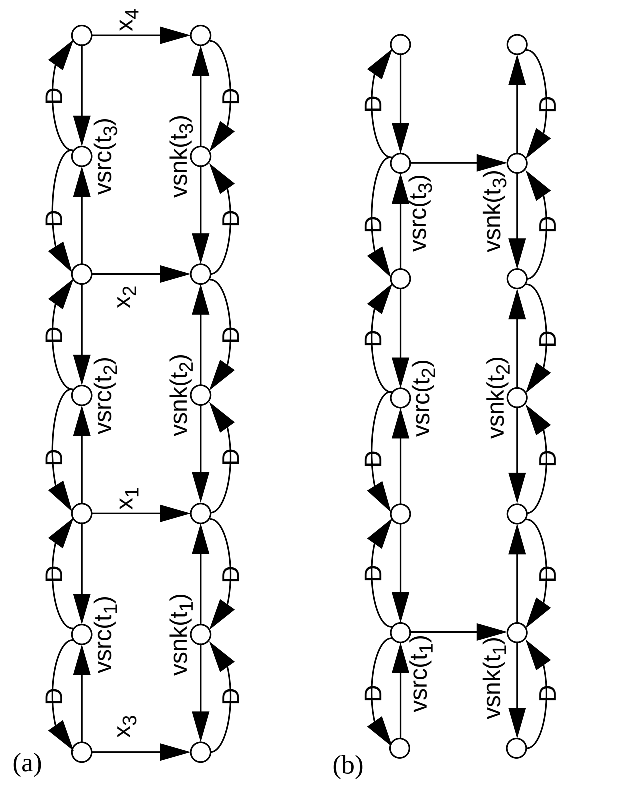


FIGURE 2. (a). An instance of the pairwise resynchronization problem that is derived from an instance of the set covering problem (b). The DFG that results from a solution to this instance of resynchronization.

This example illustrates how an instance of pairwise resynchronization can be constructed (in polynomial time) from an instance of set covering, and how a solution to this instance of pairwise resynchronization can easily be converted into a solution of the set covering instance. Our proof of the NP-hardness of pairwise resynchronization, presented in [1], is a generalization of the example in Figure 2. We summarize with the following theorem. A formal proof is given in [1].

Theorem 2: The pairwise resynchronization problem is NP-hard, and thus, the resynchronization problem is NP-hard.

This correspondence to set covering also yields a framework for adapting any heuristic for set covering into a heuristic for the resynchronization problem. In this framework, each pair (C_1, C_2) of distinct SCCs is resynchronized separately by computing the family of subsets $T = \{\varphi(u, v) | u \in C_1, v \in C_2\}$, where $\varphi(u, v)$ is the set of synchronizations that can be eliminated if we implement a zero delay synchronization edge directed from u to v. Each of the sets $\varphi(u, v)$ can easily be obtained by adding edge (u, v) to the synchronization graph, and then applying the redundant edge removal procedure described in [1]. The complexity of this procedure is $O(|V|^3)$. If we denote the set of original synchronization edges directed from members of C_1 to C_2 by Z, then T and Z form an instance of set covering. This instance is attacked with the given set covering heuristic to yield a set Z' of new synchronization edges directed from vertices in C_1 to vertices in C_2. Then, in the overall synchronization graph, the members of Z are replaced with the members of Z'. This procedure is repeated for every distinct pair of SCCs C_1 and C_2 such that there is at least one edge directed from a member of C_1 to a member of C_2. A detailed specification of this approach will be given in section 5.

In addition to establishing the intractability of resynchronization, and identifying a natural heuristic solution, we have identified a class of synchronization graphs for which optimal resynchronizations can be computed using an efficient polynomial-time algorithm. Due to lack of space, we cannot elaborate on this approach here; instead, we refer the reader to [3] for details.

4. Example

In this section we present an example of resynchronization as applied to a 7 band QMF (Quadrature Mirror Filter) Filter Bank application. Such an application consists of a set of analysis filters used to decompose a signal, and a set of synthesis filters used to reconstruct the decomposed signal (Fig. 3). See [22] for detailed description about filter banks.

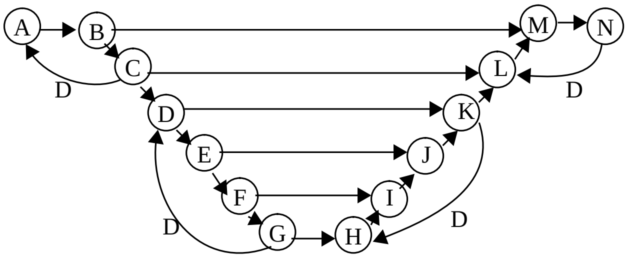


FIGURE 3. Initial synch. graph for a 7 band QMF filter bank; A, B, C are assigned to one processor, D, E, F, G, to the second, H, I, J, K to the third, and L, M, N to the fourth.

If we apply the resynchronization heuristic outlined in the previous section, we get the resynchronized graph in Fig. 4. Note that instead of the 9 feedforward synchronization edges in Fig. 3, we now have only 4 such edges (dashed edges in Fig. 4).

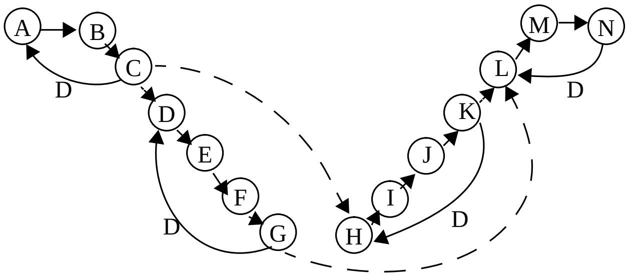


FIGURE 4. The resynchronized graph, $R(G_s)$, corresponding to Fig. 3.

5. Conclusions

This paper has outlined our research on resynchronization for shared-memory multiprocessor implementations of iterative dataflow programs. We have established that the optimal resynchronization problem is NP-hard through a correspondence to set covering. Furthermore, we have shown that a family of heuristics emerges naturally from the correspondence to set covering. The psudocode for an algorithm corresponding to such a family of heuristics is shown in Fig. 5. We have illustrated a practical example where such heuristics may be applied to reduce synchronization costs.

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

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

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

  $\chi(u, v) = \{e \in F | \rho_G(v, snk(e)) = \rho_G(v, snk(e)) = 0\}$

  End For
End For
  $T = \{\chi(u, v) | (u \text{ is in } G_1 \textbf{ and } v \text{ is in } G_2)\}$

$\Xi = Cover(F, T)$
Return $\{d_0(u, v) | \chi(u, v) \in \Xi\}$
```

FIGURE 5. An algorithm for resynchronization that is derived from an arbitrary algorithm Cover for the set covering problem

Several useful directions for future work emerge from our study. These include investigating whether efficient techniques can be developed that consider resynchronization opportunities within strongly connected components, rather than just accross feedforward edges. There may also be considerable room for improvement over our proposed family of heuristics, which is a straightforward adaptation of existing set covering algorithms. In particular, it may be interesting to search for properties of practical synchronization graphs that could be exploited in addition to the correspondence with set covering. The extension of Sarkar's concept of counting semaphores [25] to self-timed, iterative execution, and the incorporation of extended counting semaphores within our resynchronization framework are also interesting directions for further work.

References[1]

[1] S. S. Bhattacharyya, S. Sriram, and E. A. Lee, *Optimizing Synchronization in Multiprocessor Implementations of Iterative Dataflow Programs*, Electronics Research Laboratory, University of California at Berkeley, January, 1995.

[2] S. S. Bhattacharyya, S. Sriram, and E. A. Lee, "Minimizing Synchronization Overhead in Statically Scheduled Multiprocessor Systems," *Proc. Intl. Conf. on Application Specific Array Processors*, July, 1995.

[3] S. S. Bhattacharyya, S. Sriram, and E. A. Lee, *Resynchronization for Embedded Multiprocessors*, Electronics Research Laboratory, University of California at Berkeley, September, 1995.

[4] L. F. Chao and E. Sha, "Unfolding and Retiming Data-Flow DSP Programs for RISC Multiprocessor Scheduling," *Proc. Intl. Conf. on Acoustics, Speech, and Signal Processing*, April 1992.

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

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

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

[8] R. Govindarajan, G. R. Gao, and P. Desai, "Minimizing Memory Requirements in Rate-Optimal Schedules," *Proc. Intl. Conf. on Application Specific Array Processors*, August, 1994.

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

[10] D. S. Johnson, "Approximation Algorithms for Combinatorial Problems," *Journal of Computer and System Sciences*, Vol. 9, 1974.

[11] 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.

[12] E. A. Lee and D. G. Messerschmitt, "Synchronous Dataflow", *Proceedings of the IEEE*, September, 1987.

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

[14] L. Lovasz, "On the Ratio of Optimal Integral and Fractional Covers," *Discrete Mathematics*, Vol. 13, 1975.

[15] K. Parhi and D. G. Messerschmitt, "Static Rate-optimal Scheduling of Iterative Data-flow Programs via Optimum Unfolding," *IEEE Transactions on Computers*, February 1991.

[16] J. Pino, S. Ha, E. A. Lee, and J. T. Buck, "Software Synthesis for DSP Using Ptolemy," *Journal of VLSI Signal Processing*, January, 1995.

[17] D. B. Powell, E. A. Lee, and W. C. Newman, "Direct Synthesis of Optimized DSP Assembly Code from Signal Flow Block Diagrams," *Proc. Intl. Conf. on Acoustics, Speech, and Signal Processing*, March, 1992.

[18] P. L. Shaffer, "Minimization of Interprocessor Synchronization in Multiprocessors with Shared and Private Memory," *Proc. Intl. Conf. on Parallel Processing*, 1989.

[19] 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, 1993.

[20] H. Printz, "Compilation of Narrowband Spectral Detection Systems for Linear MIMD Machines," *Proc. Intl. Conf. on Application Specific Array Processors*, August, 1992.

[21] S. Ritz, M. Pankert, and H. Meyr, "High Level Software Synthesis for Signal Processing Systems," *Proc. Intl. Conf. on Application Specific Array Processors*, August, 1992.

[22] P. P. Vaidyanathan, Multirate Systems and Filter Banks, Prentice Hall, 1993.

[23] M. Veiga, J. Parera, and J. Santos, "Programming DSP Systems on Multiprocessor Architectures," *Proc. Intl. Conf. on Acoustics, Speech, and Signal Processing*, April 1990.

[24] G. C. Sih and E. A. Lee, "Scheduling to Account for Interprocessor Communication Within Interconnection-Constrained Processor Networks," *Proc. Intl. Conf. on Parallel Processing*, 1990.

[25] V. Sarkar, "Synchronization Using Counting Semaphores," *Proc. Intl. Symp. on Supercomputing*, 1988.

[26] R. Reiter, Scheduling Parallel Computations, *Journal of the ACM*, October 1968.

[1] References 1, 2, and 3 are available by anonymous ftp from ptolemy.eecs.berkeley.edu in the directory pub/ptolemy/papers/synchOpt.