Submitted to Intl.Conf. on Application-specific Systems, Architectures & Processors

LATENCY-CONSTRAINED RESYNCHRONIZATION FOR MULTIPROCESSOR DSP IMPLEMENTATION

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

February 1, 1996

ABSTRACT

Resynchronization is a post-optimization for static multiprocessor schedules in which extraneous synchronization operations are introduced in such a way that the number of original synchronizations that consequently become *redundant* significant exceeds the number of additional synchronizations. Redundant synchronizations are synchronization operations whose corresponding sequencing requirements are enforced completely by other synchronizations in the system. The amount of run-time overhead required for synchronization can be reduced significantly by eliminating redundant synchronizations [3, 19]. Thus, effective resynchronization reduces the net synchronization overhead in the implementation of a multiprocessor schedule, and thus improves the overall throughput.

However, since additional serialization is imposed by the new synchronizations, resynchronization can produce significant increase in latency. This paper addresses the problem of computing an optimal resynchronization (one that results in the lowest average rate at which synchronization operations have to be performed) among all resynchronizations that do not increase the latency beyond a prespecified upper bound L_{max}. Our study is based in the context of self-timed execution of iterative dataflow programs, which is an implementation model that has been applied extensively for digital signal processing systems.

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 was with the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley. He is now with the DSP R&D Centre, Texas Instruments.

E. A. Lee is with the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley.

1. Introduction

In a shared-memory multiprocessor system, it is possible that certain synchronization operations are *redundant*, which means that their sequencing requirements are enforced entirely by other synchronizations in the system. It has been demonstrated that the amount of run-time overhead required for synchronization can be reduced significantly by detecting and eliminating redundant synchronizations [3, 19].

The objective of resynchronization is to introduce new synchronizations in such a way that the number of original synchronizations that consequently become redundant is significantly greater that the number of new synchronizations. Thus, effective resynchronization improves the overall throughput of a multiprocessor implementation by decreasing the average rate at which synchronization operations are performed. Since additional serialization is imposed by the new synchronizations, resynchronization can produce significant increase in latency. This paper addresses the problem of computing an optimal resynchronization among all resynchronizations that do not increase the latency beyond a prespecified upper bound L_{max}. We address this problem in the context of self-timed execution of iterative synchronous dataflow (SDF) [13] programs. An iterative dataflow program consists of a dataflow representation of the body of a loop that is to be iterated infinitely. Iterative SDF programming is used extensively for the implementation of digital signal processing systems. Examples of commercial tools that use this model are the Signal Processing Worksystem (SPW) [18], and COSSAP [21], and examples of tools developed at various universities are Ptolemy [17], the Warp compiler [20], DESCARTES [21], and GRAPE [12].

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 [13], our techniques can easily be adapted to general SDF graphs. We refer to a homogeneous SDF graph as a **DFG**. The techniques developed in this paper can be used as a post-processing step to improve the performance of any static multiprocessor scheduling technique for iterative DFGs, such as those described in [1, 6, 9, 17, 20].

Our implementation model involves a *self-timed* scheduling strategy [14]. 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 [2] for details on the assumed synchronization protocols). Thus, resynchronization achieves its benefit by reducing the rate of accesses to shared memory for the purpose of synchronization.

2. Background on synchronization optimization

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 $src(e) = snk(e)$. We define $d_n(x, y)$ to represent an SDF edge whose source and sink vertices are x and y, respectively, and whose delay is n (assumed non-negative).

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

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

$$start(v_i, k) \geq end(v_j, k - delay((v_j, v_i))), \quad (1)$$

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.* An **execution source** of a synchronization graph is any actor that has nonzero delay on each input edge.

Initially, the synchronization graph is identical to G_{ipc}. However, resynchronization modifies the synchronization graph by adding and deleting synchronization edges. After resynchronization, the IPC edges in G_{ipc} represent buffer activity, and must be implemented as buffers in shared memory, whereas the synchronization edges represent synchronization constraints, and are implemented by updating and testing flags in shared memory. 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 [22], the throughput (number of DFG iterations per unit time) of a synchronization graph G is given by

$$\tau = \min_{\text{cycle } C \text{ in } G} \left\{ \frac{\Delta(C)}{\sum_{v \in C} t^*(v)} \right\}, \tag{2}$$

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

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 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.

Any transformation that we perform on the synchronization graph must respect the synchronization constraints implied by G_{ipc}. If we ensure this, then we only need to implement the synchronization edges of the optimized synchronization graph. If $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$ are synchronization graphs with the same vertex-set and the same set of intraprocessor edges (edges that are not synchronization edges), we say that G_1 **preserves** G_2 if for all $e \in E_2$ such that $e \notin E_1$, we have $\rho_{G_1}(src(e), snk(e)) \leq delay(e)$, 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 following theorem, which is developed in [2], 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 due to the path $((C, E), (E, D), (D, F))$. In [2], it is shown that if all redundant edges in a synchronization graph are removed, then the resulting graph preserves the original synchronization graph.

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

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

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

If $G = (V, E)$ is a synchronization graph and F is the set of feedforward edges in G,

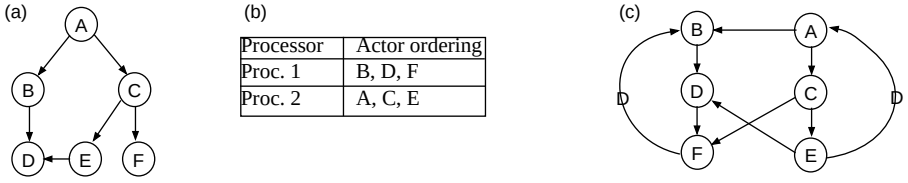


Figure 1. Part (c) shows the IPC graph that corresponds to the DFG of part (a) and the processor assignment / actor ordering of part (b). A "D" on top of an edge represents a unit delay.

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

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

Our concept of resynchronization considers the rearrangement of synchronizations only "across" feedforward edges. We impose this restriction so that the serialization imposed by resynchronization does not degrade the estimated throughput. Feedforward edges do not reduce the estimated throughput because they do not affect the value derived from (2).

3. Latency-constrained resynchronization

By the *latency* of a multiprocessor implementation, we mean the time required for the first invocation of a specific execution source, called the **latency input**, to influence a specific output actor, called the **latency output**. We denote the latency of a synchronization graph G by L_G.

As discussed earlier, resynchronization cannot decrease the estimated throughput since it manipulates only the feedforward edges of the synchronization graph. Frequently in real-time signal processing systems, latency is also an important issue. Since resynchronization introduces serialization of tasks in a DFG, it can in general produce a significant increase in latency. The **latency-constrained resynchronization (LCR) problem** is the problem of computing a minimal resynchronization among all resynchronizations that do not increase the latency beyond a pre-

specified upper bound L_{max}.

In our study of LCR, we restrict our attention to a class of synchronization graphs, called **transparent graphs**, for which the latency can be computed efficiently. This is the class of synchronization graphs in which the first invocation of the latency output is influenced by the first invocation of the latency input; equivalently, it is the class of graphs that have at least one delayless path in the corresponding application DFG directed from the latency input to the latency output [4]. Since the first invocation of any execution source starts execution at time 0, the latency of a transparent synchronization graph is given by $end(v_o, 1)$ (the completion time of the first invocation of v_o), where v_o is the associated latency output.

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

$$L_G'(x, y) = max(\{end(y, k + \rho_{G_0}(x, y)) - start(x, k) \mid (k = 1, 2, \ldots)\}), \quad (3)$$

where G_0 is the associated application DFG.

Currently, there are no known tight upper bounds on L_G' that can be computed efficiently from the synchronization graph for any useful subclass of graphs, and thus, we use the lower bound approximation L_G, which corresponds to the critical path, when attempting to analyze and optimize the input-output propagation delay of a self-timed system. The heuristic that we present in Section 7 for latency-constrained resynchronization can easily be adapted to handle arbitrary

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

If a synchronization graph G is transparent, then the latency can be computed efficiently using longest path calculations on an *acyclic* graph that is derived from the input synchronization graph G. This acyclic graph, denoted $\mathit{fi}(G)$, is constructed by removing all edges from G that have nonzero-delay, adding a vertex υ, setting $t(\upsilon) = 0$, and adding delayless edges from υ to each execution source of G.

In [4], it is shown that if G is a transparent synchronization graph with latency output y, then $L_G = T_{\mathit{fi}(G)}(\upsilon, y)$, where $T_{\mathit{fi}(G)}(a, b) \equiv -\infty$ if there is no path from a to b in $\mathit{fi}(G)$, and if there is a path from a to b, then $T_{\mathit{fi}(G)}(a, b)$ is defined to be the maximum cumulative execution time (sum of the execution times over all vertices in a given path) over all paths directed from a to b. Furthermore, the latency of the synchronization graph G' that results from inserting a new synchronization edge (v_1, v_2) (a resynchronization edge) into G can be computed from

$$L_{G'} = max(\{(T_{\mathit{fi}(G)}(\upsilon, v_1) + T_{\mathit{fi}(G)}(v_2, y)), L_G\}). \quad (4)$$

The values $T_{\mathit{fi}(G)}(a, b)$ for all pairs a, b 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 [7]. Such an efficient means for computing latency permits the development of systematic resynchronization techniques to trade off synchronization overhead and latency.

4. Related work

In [5], the problem of finding a resynchronization that has minimal cardinality (the *resynchronization problem*) is shown to be NP-hard, and an efficient family of heuristics is presented. Also, a class of synchronization graphs is identified for which optimal resynchronizations can be computed using an efficient polynomial-time algorithm. The developments of [5] assume that arbitrary increases in latency can be tolerated ("unbounded-latency resynchronization"); such a scenario may arise, for example, in simulation applications. In contrast this paper addresses the problem of resynchronization *under fixed latency constraints*.

In [19], Shaffer presents an algorithm that removes redundant synchronizations in the self-timed execution of a non-iterative DFG. This technique was subsequently extended to handle iterative execution and DFG edges that have delay [3]. 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.

In [3], an efficient algorithm, called *Convert-to-SC-graph*, is described for introducing new synchronization edges so that the synchronization graph becomes strongly connected, which allows all synchronization edges to be implemented with a more efficient synchronization protocol. It is shown that the net overhead required to implement the new edges that are added by *Convert-to-SC-graph* can be significantly less than the synchronization overhead that is eliminated by using the more efficient synchronization protocol. However, this technique may increase the latency.

Generally, resynchronization can be viewed as complementary to the *Convert-to-SC-graph* optimization: resynchronization is performed first, followed by *Convert-to-SC-graph*.

Under severe latency constraints, it may not be possible to accept the solution computed by *Convert-to-SC-graph*. In such a situation, *Convert-to-SC-graph* can be attempted on the original (before resynchronization) graph to see if it achieves a better result than resynchronization without *Convert-to-SC-graph*. However, for a significant class of synchronization graphs, the latency is not affected by *Convert-to-SC-graph*, and thus, for such systems resynchronization and *Convert-to-SC-graph* are fully complementary [4].

Resynchronization has been studied earlier in the context of hardware synthesis [8]. 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 [4].

5. Intractability

We have established that latency-constrained resynchronization is NP-hard even for the very restricted subclass of transparent synchronization graphs in which each SCC corresponds to a single actor, and all synchronization edges have zero delay. In this section, we outline the intuition behind this result; a detailed proof can be found in [4].

The intractability of latency-constrained resynchronization can be established by a reduction from the *set covering problem*, which is a well-known NP-hard problem [7]. 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.

To illustrate our reduction of set covering to latency-constrained resynchronization, we suppose that we are given the set $X = \{x_1, x_2, x_3, x_4\}$, and the family of subsets $T = \{t_1, t_2, t_3\}$, where $t_1 = \{x_1, x_3\}$, $t_2 = \{x_1, x_2\}$, and $t_3 = \{x_2, x_4\}$. Figure 2 illustrates the instance of latency-constrained resynchronization that we derive from the instance of set covering specified by (X, T). This is a synchronization graph in which each actor corresponds to a single processor and the self loop edges for each actor are not shown. The numbers beside the actors specify the actor execution times, and the latency constraint is $L_{max} = 103$. In the graph of Figure 2, which we denote by G, the edges labeled ex_1, ex_2, ex_3, ex_4 correspond respectively to the members x_1, x_2, x_3, x_4 of the set X in the set covering instance, and the vertex pairs (resynchronization candidates) $(v, st_1), (v, st_2), (v, st_3)$ correspond to the members of T. For each

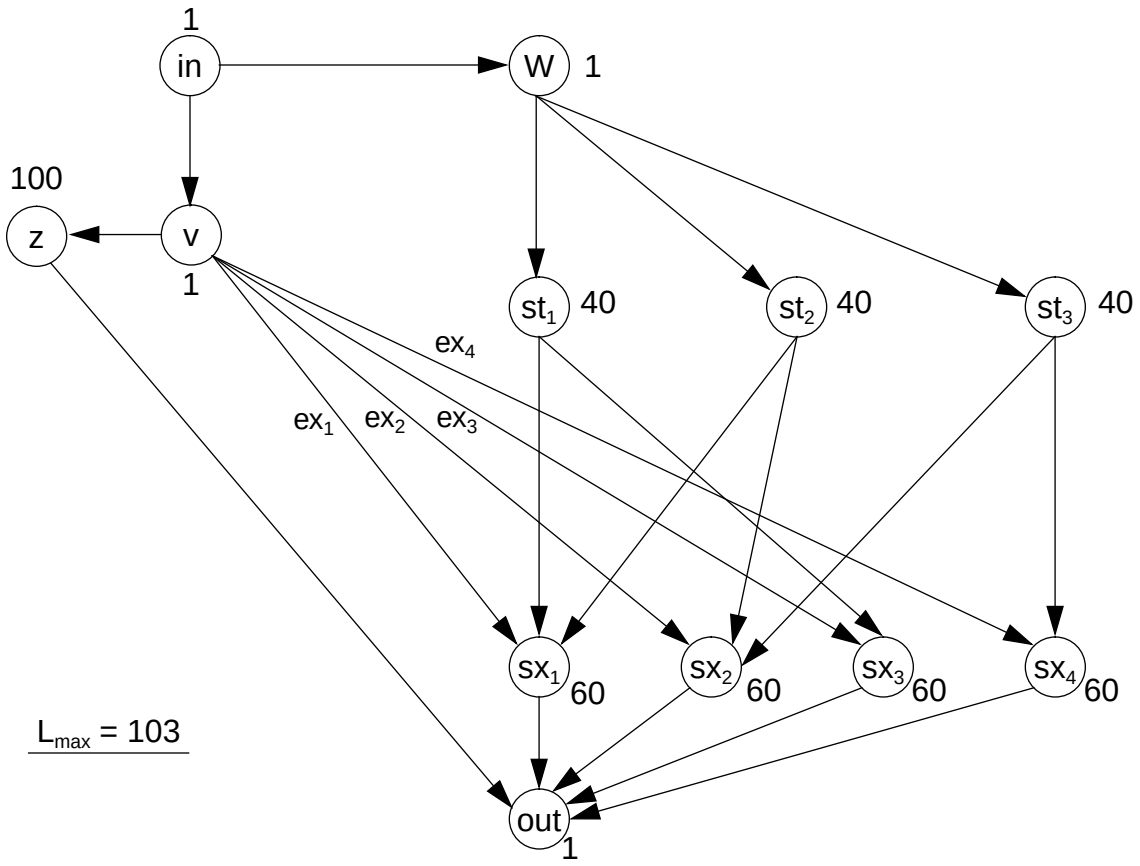


Figure 2. An instance of latency-constrained resynchronization that is derived from an instance of the set covering problem.

relation , an edge exists that is directed from to . The latency input and latency output are defined to be in and out respectively, and it is assumed that G is transparent.

Our general procedure for reducing an arbitrary instance (X', T') of set covering to an instance of latency-constrained resynchronization is in specified Figure 3. The time complexity of this transformation is $O(|X'||T'|)$.

Now let $(\tilde{X}, \tilde{T})$ denote an arbitrary instance of set covering, and let $\tilde{G}$ be the synchronization graph that results when the construction of Figure 3 is applied to $(\tilde{X}, \tilde{T})$. In [4] it is shown that from any optimal LCR $\tilde{R}$ for $\tilde{G}$, an optimal LCR for $\tilde{G}$ can be derived in polynomial time such that

for each resynchronization edge e in $\tilde{R}$, $src(e) = v$, and $snk(e) \in \Psi$, and (5)

- Instantiate actors v, w, z, in, out, with execution times 1, 1, 100, 1, and 1, respectively, and instantiate all of the edges in Figure 2 that are contained in the subgraph associated with these five actors.
- For each $t \in T'$, instantiate an actor labeled st that has execution time 40.
- For each $x \in X'$
  - Instantiate an actor labeled sx that has execution time 60.
  - Instantiate the edge $ex \equiv d_0(v, sx)$.
  - Instantiate the edge $d_0(sx, out)$.
- For each $t \in T'$
  - Instantiate the edge $d_0(w, st)$.
  - For each $x \in t$, instantiate the edge $d_0(st, sx)$.
- Set $L_{max} = 103$.

Figure 3. A procedure for constructing an instance I_{lr} of latency-constrained resynchronization from an instance I_{sc} of set covering such that a solution to I_{lr} yields a solution to I_{sc}.

$$\text{the set } \{t|((v, st) \in \tilde{R})\} \text{ is a minimum cover for } \tilde{X}, \tag{6}$$

where $\Psi = \{st|(t \in \tilde{T})\}$.[1]

Since the construction of Figure 3 can be performed in polynomial time, we have from (6) and the NP-hardness of set covering that the latency-constrained resynchronization problem is NP-hard.

The synchronization graph that results from an optimal resynchronization of Figure 2, with redundant synchronization edges removed is shown in Figure 2. Since the resynchronization candidates $(v, st_1), (v, st_3)$ were chosen to obtain the solution shown in Figure 2, this solution corresponds to the solution of (X, T) that consists of the subfamily $\{t_1, t_3\}$.

6. Two processor problem

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 (cubic in the number of nodes in the synchronization graph). This reveals a pattern of complexity that is analogous to the classic nonpreemptive processor scheduling problem with deterministic execution times, in which the problem is also intractable for general systems, but an efficient greedy algorithm suffices to yield optimal solutions for two-processor systems in which the execution times of all tasks are identical [10]. However, for latency-constrained resynchronization, the tractability for two-processor systems does not depend on any constraints on the task (actor) execution times.

1. The actors $\{st\}$ are those that were constructed in the second step of Figure 3.

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* $x_1, x_2, \ldots, x_p$, 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* $y_1, y_2, \ldots, y_q$, with associated execution times $\{t(y_i)\}$, such that each y_i is the ith actor scheduled on the sink processor; a set of irredundant synchronization edges $s_1, s_2, \ldots, s_n$ such that for each s_i, $src(s_i) \in \{x_1, x_2, \ldots, x_p\}$ and $snk(s_i) \in \{y_1, y_2, \ldots, y_q\}$; and a latency constraint L_{max}, which

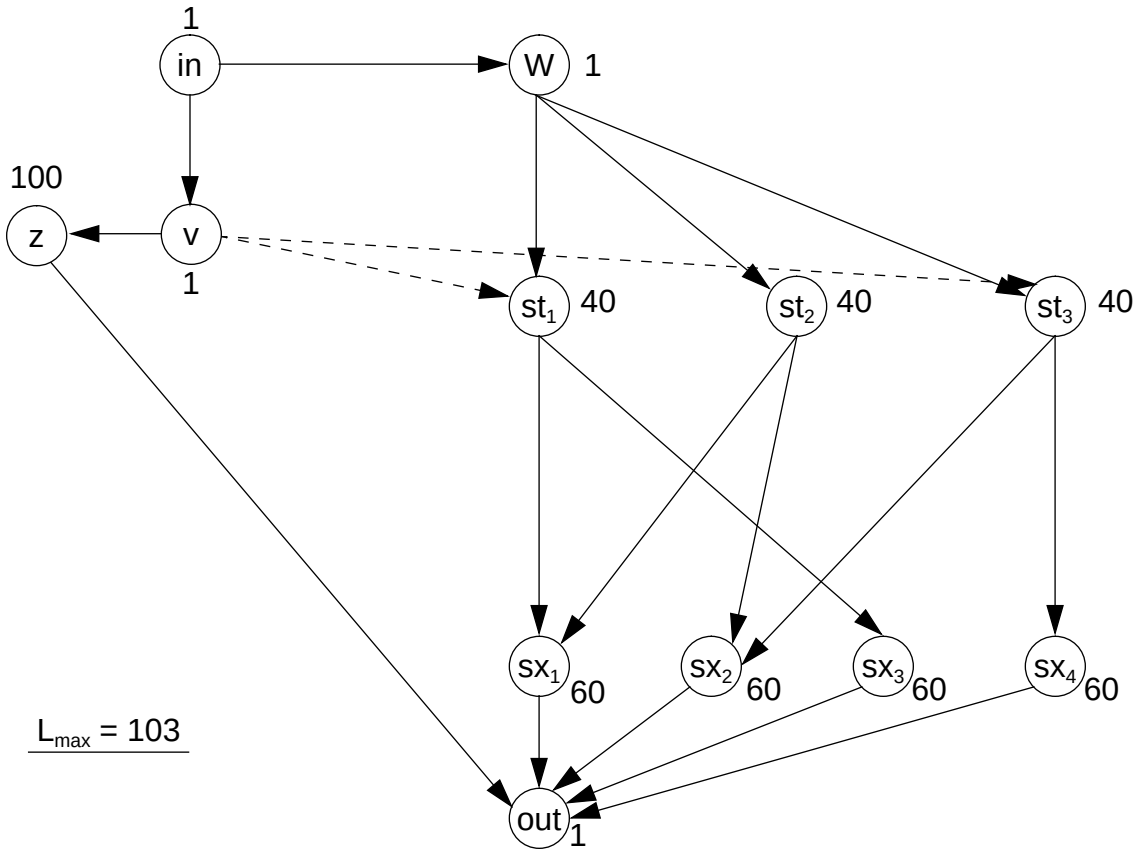


Figure 4. The synchronization graph that results from a solution to the instance of latency-constrained resynchronization shown in Figure 2.

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 Figure 5(a). Here, $p = q = 8$; and we assume that $t(z) = 1$ for each actor z, and $L_{max} = 10$.

As in the previous section, we assume that $\tilde{G}$ is transparent. In this discussion, we also assume that $delay(s_i) = 0$ for all s_i, and we refer to the subproblem that results from this restric-

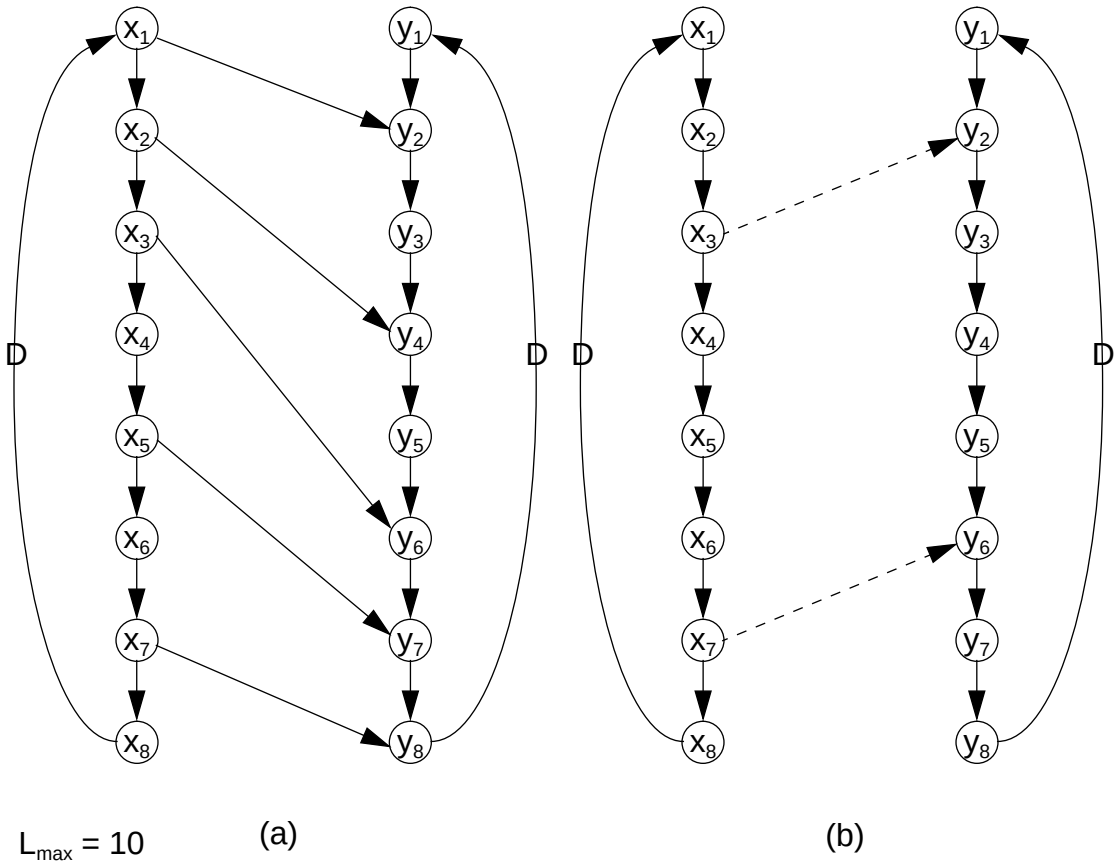


Figure 5. An instance of two-processor latency-constrained resynchronization. In this example, the execution times of all actors are identically equal to unity.

tion 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 extended this approach to solve the general (not necessarily delayless) 2LCR problem in cubic time; we refer the reader to [4] 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 $w_1, w_2, \ldots, w_N$ of the members of X (the set that must be covered) such that the collection of subsets T consists entirely of subsets of the form $\{w_a, w_{a+1}, \ldots, w_b\}$, $1 \leq a \leq b \leq N$. Thus, while general set covering involves covering a set from a collection of subsets, interval covering amounts to covering an interval from a collection of subintervals. The interval covering problem can be solved in $O(|X||T|)$ time using a straightforward approach [4].

Our algorithm for the 2LCR problem is based on the following lemma, which is established in [4].

Lemma 1: If R is a resynchronization of $\tilde{G}$, then

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

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

The set X in the interval covering instance that we derive from $\tilde{G}$ is the set $\{s_1, s_2, \ldots, s_n\}$ 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 $(s_1', s_2', \ldots, s_n')$, is specified by

$$(x_a = src(s_i'), x_b = src(s_j'), a < b) \Rightarrow (i < j)\,.^{1} \tag{7}$$

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

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

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

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

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

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

In [4] 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) $\{\chi(v_{r_1}), \chi(v_{r_2}), \ldots, \chi(v_{r_z})\}$ to this instance of interval covering, $R = \{v_{r_1}, v_{r_2}, \ldots, v_{r_z}\}$ is an optimal latency-constrained resynchronization of $\tilde{G}$.

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

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

and thus from (9), we have

1. Note that the source actors of the members of $\{s_1, s_2, \ldots, s_n\}$ are all distinct. This follows from the assumption that the members of $\{s_1, s_2, \ldots, s_n\}$ are not redundant.

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

It is easily verified that $C = \{\chi(v_3), \chi(v_7)\}$ is a minimal cover for $\{s_1, s_2, \ldots, s_n\}$ 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 Figure 5(a). The synchronization graph that results from this resynchronization is shown in Figure 5(b).

7. A heuristic for LCR

We have developed a heuristic for LCR in general, transparent synchronization graphs. Our heuristic is based on an 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 [11, 15].

To adapt this set covering technique to LCR, we construct an instance of set covering by choosing the set of elements to be covered to be the set of synchronization edges, and by choosing the family of subsets to be the collection of subsets of the form $\chi((v, w))$, where (v, w) is an ordered pair of vertices such that there is no path from w to v, and adding the resynchronization edge (v, w) does not increase the latency beyond L_{max}.

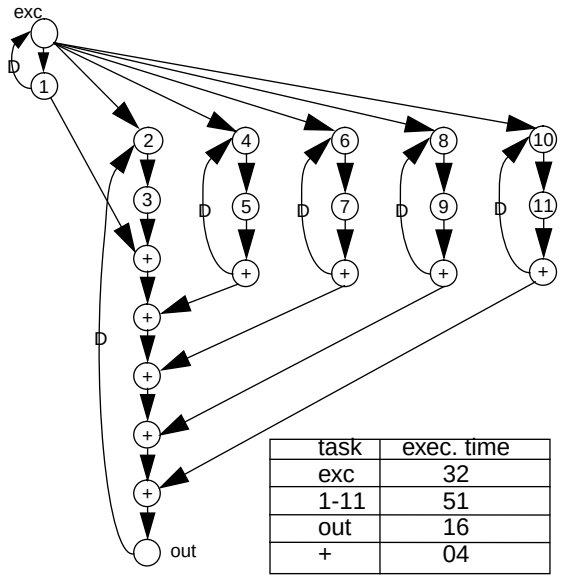
From this family of subsets, our heuristic chooses a member that has maximum cardinality, inserts the corresponding resynchronization edge, removes all synchronization edges that become redundant, and updates the values $T_{fi(G)}(x, y)$ and $\rho_G(x, y)$, for all x, y, for the new synchronization graph. This process is repeated until no more resynchronization edges can be

added without increasing the latency beyond L_{max}.

A complete pseudocode specification of the approach is shown in Figure 6. The running time is $O(v^4)$, where v is the number of graph vertices.

Figure 7 shows the synchronization graph that results from a six-processor schedule of a synthesizer for plucked-string musical instruments in 11 voices. Here *exc* represents the excitation input, each actor labeled with a positive integer i represents the i th voice, and the vertices labeled with "+" signs represent adders. The latency input and output are, respectively, *exc* and *out*, and the latency is 170.

The table on the right side of Figure 7 shows how the synchronization cost in the result computed by our heuristic changes as the latency constraint varies. If just over 50 units of latency can be tolerated beyond the original latency of 170, then the heuristic is able to eliminate a single



task	exec. time
exc	32
1-11	51
out	16
+	04

L_{max}	# synch edges
$170 \le L_{max} \le 220$	10
$221 \le L_{max} \le 267$	9
$268 \le L_{max} \le 275$	8
$276 \le L_{max} \le 381$	7
$382 \le L_{max} \le 644$	6
$645 \le L_{max} \le \infty$	5

Figure 7. A music synthesis example.

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

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

Figure 6. A heuristic for latency-constrained resynchronization.

synchronization edge. No further improvement can be obtained unless roughly another 50 units are allowed, at which point the synchronization cost 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, then the heuristic is able to reduce the synchronization cost to 6. No further improvement is obtained over the range 383-644. When $L_{max} \geq 645$, a synchronization cost of 5 is achieved.

8. Conclusions

This paper has addressed the problem of latency-constrained resynchronization for self-timed implementation of iterative dataflow programs. Given an upper bound L_{max} on the allowable latency, the objective of latency-constrained resynchronization is to insert extraneous synchronization operations in such a way that a) the number of original synchronizations that consequently become redundant significant exceeds the number of new synchronizations, and b) the serialization imposed by the new synchronizations does not increase the latency beyond L_{max}. To ensure that the serialization imposed by resynchronization does not degrade the throughput, the new synchronizations are restricted to lie outside of all cycles in the final synchronization graph. We have established that latency-constrained resynchronization is NP-hard; we have derived an optimal, polynomial-time algorithm for two-processor systems; and we have developed and implemented a heuristic for general *n*-processor systems. Through a practical example, we have illustrated the ability of this heuristic to systematically trade off between synchronization overhead and latency.

References[1]

[1] S. Banerjee, D. Picker, D. Fellman, and P. M. Chau, “Improved Scheduling of Signal Flow Graphs onto Multiprocessor Systems Through an Accurate Network Modeling Technique,” *VLSI Signal Processing VII*, IEEE Press, 1994.

[2] S. 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.

[3] 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.

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

[5] S. S. Bhattacharyya, S. Sriram, and E. A. Lee, “Self-Timed Resynchronization: a Post-optimization for Static Multiprocessor Schedules,” to appear in *Proc. Intl. Parallel Processing Symposium*, April, 1996.

[6] 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.

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

1. References 2, 3, and 4 are available by anonymous ftp from ptolemy.eecs.berkeley.edu in the directory pub/ptolemy/papers/synchOpt.

[8] 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.

[9] 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.

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

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

[12] 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.

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

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

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

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

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

[18] 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.

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

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