Presented at the Twenty-Eighth Annual Conference on Signals, Systems, and Computers, Asilomar, Nov 1994

Statically Scheduling Communication Resources in Multiprocessor DSP Architectures

S. Sriram, and Edward A. Lee†
{sriram,eal}@eecs.berkeley.edu
University of California at Berkeley

ABSTRACT

In statically scheduled multiprocessors inter-processor communication resources can be scheduled by determining, at compile time, the order in which processors require access to shared resources and enforcing this order at run time. We show how to choose an access order such that, under certain assumptions, imposing that order incurs no performance penalty.

1.0 Introduction

In this paper we address multiprocessor implementation of applications that are specified as Synchronous Data Flow (SDF) graphs. We recall that in a dataflow representation, an algorithm is represented as a graph where nodes (actors) are individual computations and directed arcs between them represent flow of data (tokens). SDF refers to a subclass of dataflow graphs where the actors lack data dependency in their firing patterns. I.e. the number of tokens produced and consumed in each of the output and input arcs of each actor is constant and fixed at compile time. In this paper we assume that the application is a homogeneous SDF graph, i.e. a graph in which the actors always produce and consume exactly one token. A general SDF graph can always be converted into a homogeneous graph [1]. SDF has been shown to be a useful model for representing a large class of DSP algorithms. The dataflow graph (DFG) corresponding to an application may be extracted directly from a block diagram specification (e.g. in Ptolemy [2]) or from an applicative language like Silage (as done for example in Hyper [3]). The dataflow graphs of interest for the purpose of representing DSP algorithms are run in a nonterminating fashion; tokens flow from source actors to sink actors continually.

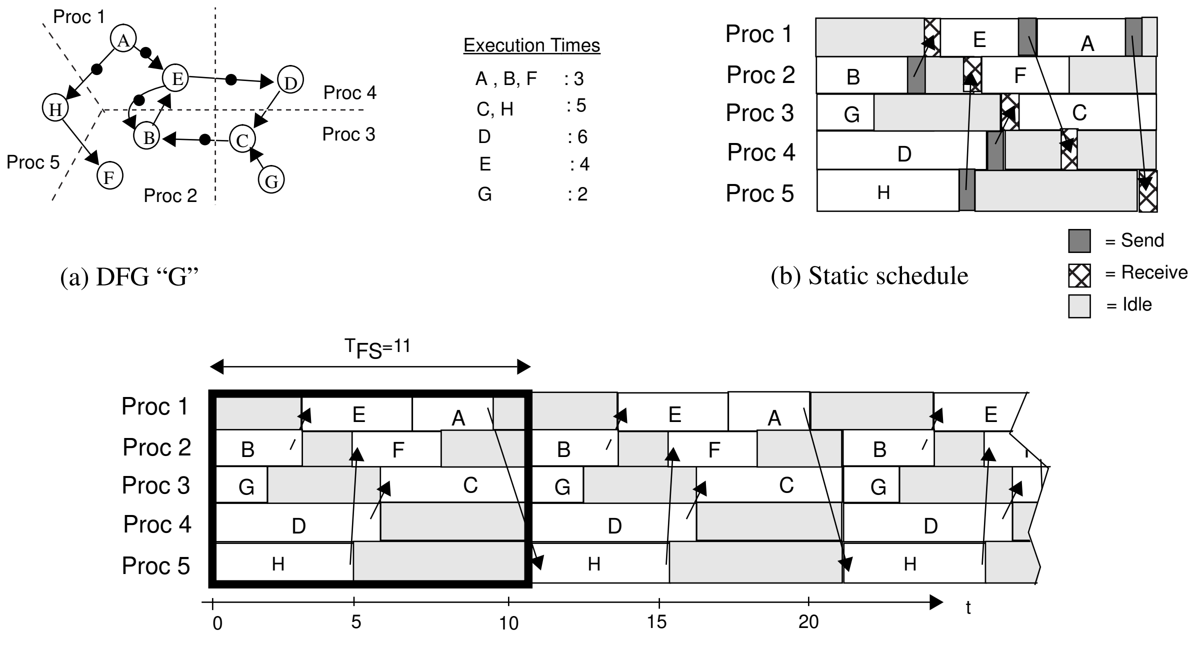
Multiprocessor implementation of an algorithm specified as a DFG involves scheduling computations in the algorithm. By "scheduling" we collectively refer to the task of assigning actors in the DFG to processors, ordering execution of these actors on each processor, and determining when each actor fires (begins execution) such that all data precedence constraints are met. Each of these three tasks may be performed either at run time (a dynamic strategy) or at compile time (static strategy). In [4] and [5] the authors propose a scheduling taxonomy based on which of these tasks are performed at compile time and which at run time; in this paper we will use the same terminology that was introduced there. To reduce run time computation costs it is advantageous to perform as many of the three scheduling tasks as possible at compile time. Which of these can be effectively performed at compile time depends on the information available about the execution time of each actor.

The performance metric that is of interest for evaluating schedules is the average iteration period T: the average time it takes for all the actors in the graph to be executed once. Equivalently, we could use the throughput T^{-1} (i.e. the number of iterations of the graph executed per unit time) as a performance metric. Thus an optimal schedule is one that minimizes T.

In this paper we focus on scheduling strategies that assign actors to processors and determine the order of execution of actors on processors, both at compile time, because these strategies appear to be most useful for a significant class of DSP algorithms. We will look at three such scheduling strategies: fully-static, self-timed, and ordered transactions. In the fully-static scenario, the exact firing time of each actor is also determined at compile time; in the self-timed strategy processors determine when to fire an actor by synchronizing with other processors at run time, whereas in the ordered transactions approach a total order on all inter-processor communications is determined at compile time and enforced at run time.

Out of the three strategies, run time overhead is the smallest for the fully-static case and is the most for the self-timed case; the ordered transactions strategy lies in between. The fully-static strategy works only if actor execution time estimates are accurate and data-independent. The ordered transactions strategy is tolerant of variations in execution times of actors, and the self-timed schedule is even more so. If we ignore the run time communication and synchronization overhead and assume that the execution time estimates are accurate, then in general the self-timed strategy yields the minimum possible iteration period among the three scheduling strategies if we keep processor assignment and actor ordering fixed. The main result of this paper is that it is possible to choose a transaction ordering such that the transaction ordered strategy performs as well as the self-timed strategy. Thus we show how to find the best possible transaction ordering for the

† The authors were supported by SRC (DC-008-014), ARPA and US Air Force (under the RASSP program), and the NSF (MIP9201605).



(a) DFG "G"

(b) Static schedule

(c) Fully-static execution

Figure 1. Fully-static schedule on five processors

given processor assignment, actor ordering, and timing estimates.

2.0 Notation

The DFG (assumed to be a homogeneous SDF graph in this paper) is represented as a weighted digraph $G(V, E, t, w)$ where the vertices $v \in V$ represent the actors, directed edges $v_i \to v_j \in E$ represent the data dependencies in G, the function $t(v)$ assigns a positive integer execution time to each actor v (the actual execution time can be interpreted as $t(v)$ cycles of a base clock), and the function $d(v_i, v_j)$ (or $d(i, j)$ for short) assigns a positive integer number of initial tokens to each edge $v_i \to v_j \in E$ of G. We represent initial tokens on arcs by bullets on the edges of the DFG (see Fig. 1(a)). Recall that the semantics of a DFG is that an actor v can begin executing its function when it has tokens on all its input arcs, and it produces one token on each of its output arcs $t(v)$ time units after it begins execution.

3.0 Fully-static schedule

A fully-static scheduling strategy is one in which all the three scheduling operations — assigning actors to processors, ordering of actors on processors, and determining when each actor fires — are performed at compile time [5]. Although determining an optimal fully-static schedule is NP-hard, several heuristics have been proposed for this problem. Some of these heuristics generate a non-overlapped blocked schedule [6], whereas others generate overlapped schedules using a modified list scheduling technique [7][8]. Thus, if P processors are available, these heuristics determine the processor assignment $\sigma_p(v) \to [1, 2, \ldots, P]$ for each actor v, and specify when the kth invocation of each actor starts: $s(v, k) \to Z^+$ (positive integer). Because the firing times are enforced by a finite state controller in practice, a fully-static schedule is also constrained to be periodic, i.e. $s(v, k) = \sigma_t(v) + kT_{FS}$; $\sigma_t(v)$ is the starting time of the first execution of actor v and T_{FS} (the subscript FS implies fully-static) is the schedule period. Thus a fully-static schedule specifies the triple $\{\sigma_p(v), \sigma_t(v), T_{FS}\}$. Clearly, the throughput for such a schedule is $1/T_{FS}$. An example of a fully-static execution of a DFG is shown in the Gantt chart in Fig. 1: Fig 1(c) is one possible fully-static schedule on five processors for the graph G in Fig. 1(a). Note that inter-processor communication primitives (*send* and *receive* actors) need to be inserted when data cross processor boundaries. The fully-static schedule specifies exactly when these communications occur. If we ignore communication costs, i.e. assume *send*s and *receive*s take zero time, then T_{FS} for this example is 11 units.Also, the sizes of buffers between processors can be inferred from the schedule, and hence these buffers can be statically allocated.

In some cases it is advantageous to *unfold* a graph by a certain unfolding factor, say u, and schedule u iterations of the graph together in order to exploit inter-iteration parallelism more effectively [1][6]. The unfolded graph contains u copies of each actor of the original graph. In that case σ_p and σ_t are defined for all the nodes of the *unfolded* graph (i.e. σ_p and σ_t are defined for the first u invocations of each actor) and T_{FS} is the iteration

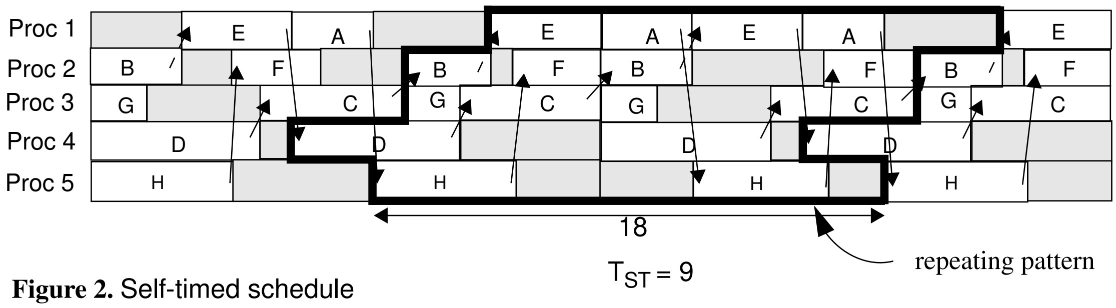


Figure 2. Self-timed schedule

period for the unfolded graph. Then, the average iteration period for the original algorithm is T_{FS}/u. For the remainder of this paper, we will assume we are dealing with the unfolded graph and we will refer only to the iteration period and throughput of the unfolded graph, with the understanding that these quantities can be scaled by the unfolding factor to obtain the corresponding quantities for the original graph.

Fully-static scheduling requires accurate estimates of execution times of actors and requires that actors have constant, data-independent execution times because the timings specified by the fully-static schedule guarantee correct sender-receiver synchronization only when the execution time estimates are accurate and constant. One way to get around this problem is to use guaranteed worst case execution time estimates. Such worst case estimates are often used in scheduling hardware in high-level VLSI synthesis. For programmable processors however, determining good upper bounds on execution times is not always possible, especially when the object code is compiled from a high level language or when processors employ pipelining and other instruction-level parallelism techniques. The strategy described next is more robust to changes in execution times of actors, but it achieves this flexibility at a greater run time cost.

4.0 Self-timed schedules

Consider now the *self-timed* scheduling strategy of [4]. In this strategy we retain the processor assignment σ_p from the fully-static schedule, we also retain the ordering of actors on each processor as specified by σ_t, but we discard the precise timing information specified in the fully-static schedule. Each processor is assigned a sequential list of actors, some of which are *send* and *receive* actors, which it executes in an infinite loop. When a processor executes a communication actor, it synchronizes with the processor(s) it communicates with. Thus exactly when a processor executes each actor depends on when, at run time, all input data for that actor is available, unlike the fully-static case where no such run time check is needed. Conceptually, the processor sending data writes data into a FIFO (first-in-first-out) buffer, and blocks when that buffer is full. The receiver on the other hand blocks when the buffer it reads from is empty. Such buffers may be implemented using shared memory, or by using hardware FIFOs between processors. It is possible to optimize (minimize) buffer sizes such that the throughput is not constrained by the fact that buffer sizes are bounded [10]; however, since we are mainly interested in determining the best performance achievable by a self-timed strategy, we will not be concerned with buffer optimization in this paper. Instead, we will assume that the buffers are large enough so that their finite sizes do not affect the throughput of the system of processors.

A self-timed strategy is robust with respect to changes in execution times of actors, because sender-receiver synchronization is performed at run time. However, such run time synchronization also implies a higher inter-processor communication cost compared to the fully-static strategy because semaphore checks need to be performed to ensure sender-receiver sychronization and shared resources need to be arbitrated at run time.

Another feature of the self-timed strategy is that it allows successive iterations of the DFG to overlap in a natural manner. Fig. 2 shows how the self-timed schedule corresponding to the fully-static schedule in Fig. 1 evolves. Note that the self-timed schedule in Fig. 2 eventually settles to a periodic pattern consisting of two iterations of the DFG. Thus the average execution time for one iteration of the DFG, T_{ST}, is 9 units. Clearly, if we neglect inter-processor communication costs, the overlapping of successive iterations ensures that $T_{ST} \leq T_{FS}$.

In a self-timed strategy, $s(v, k)$ (time when actor v starts firing for the kth time) is determined by how the schedule evolves at run time. To model the evolution of a self-timed schedule, we construct another DFG G' by adding arcs (edges) to the original DFG G to reflect the processor assignment and ordering of actors on each processor. More precisely, G' is obtained from G by linking actors assigned to each processor into a cycle that has a single initial token. Thus the fact that actors assigned to the same processor run sequentially is reflected in G'. Fig. 3 represents the graph G' for the example of Fig. 1(a),(b). Note, for instance, how nodes B and F are linked into a cycle in G' to reflect the fact that they are both assigned to the same processor (Proc 2). Also, note that the initial token is placed on the input arc of B; this reflects the fact that the kth firing of B always precedes the kth firing of F on Proc 2. Again, we have ignored explicit communication actors and their associated execution times here, but these can be included in the model in a

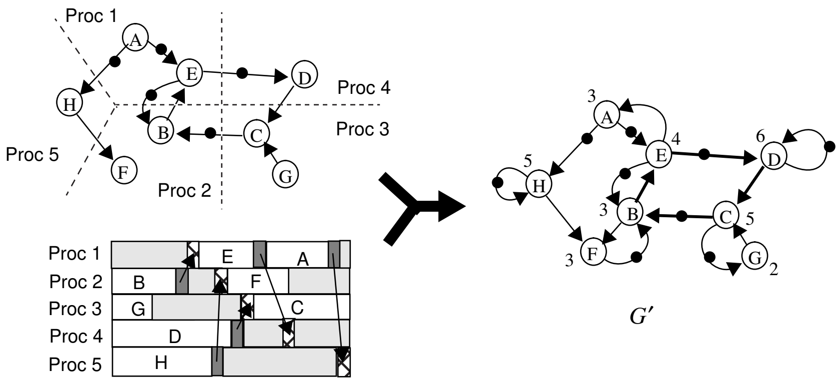


Figure 3. Construction of G' from G and from the static schedule

straightforward manner. The average iteration period for the self-timed schedule is given by:

$$T_{ST} = \max_{\text{cycle C in G}'} \left\{ \frac{\sum_{v \in C} t(v)}{d(C)} \right\}$$ where $d(C)$ is

the number of initial tokens on the cycle C.[1]

For example, the value of T_{ST} obtained from G' in Fig. 4 is 9 units (corresponding to the cycle B⟶E⟶D⟶C⟶B, which has total weight of 18 and has two initial tokens on it). Thus the average period for the self-timed schedule of Fig. 2 is 9. Note that T_{ST} is a rational number, but it is not necessarily an integer.

5.0 Ordered transactions

In the ordered transactions strategy we discard the firing time information in a fully-static schedule, but retain the order of execution of nodes on each processor and the order in which processors communicate with one another [5]. This strategy is more constrained than self-timed scheduling because at run time we impose a pre-determined order on the inter-processor communication pattern. However, as in the self-timed scenario, this ordering strategy is tolerant of variations in execution times of actors. Fig. 4 shows an example of how such an order could be derived from a given static schedule.

The main advantage of ordering inter-processor transactions is that it allows us to restrict access to communication resources statically, based on the communication pattern determined at compile time. Since communication resources are typically shared between processors, run time contention for these resources is eliminated by statically scheduling them. This can potentially result in efficient inter-processor communication mechanism at low hardware cost. We have demonstrated the ordered transactions concept by building a prototype four processor DSP board, called the Ordered Memory Access (OMA) architecture, that uses shared memory and a single shared bus for inter-processor communication. The order in which processors access shared memory is determined at compile time, and a controller on the board enforces this pre-determined access order at run time, thus eliminating the need for bus arbitration or semaphore synchronization at run time. This results in efficient inter-processor communication at relatively low hardware cost. The OMA multiprocessor is described in detail in [12].

The ordered transactions strategy falls in between fully-static and self-timed strategies in that, like the self-timed strategy, it is tolerant of variations in execution times and, like the fully-static strategy, has low communication and synchronization costs. However, the ordered transaction strategy is not as flexible as self-timed, because the order in which processors access shared resources is forced at run time to exactly match the order determined at compile time.

The OT strategy lies in between fully-static and self-

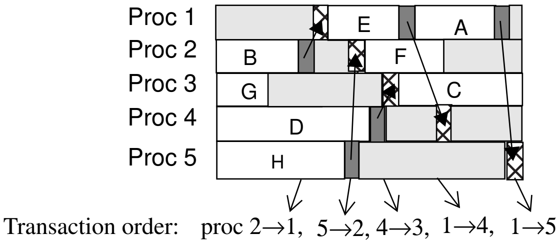


Figure 4. One possible transaction order

timed in terms of the average throughput also. In general $T_{FS} \geq T_{OT} \geq T_{ST}$. For example if we enforce the transaction ordering specified in Fig. 4, then the schedule evolves as shown in Fig. 5, and T_{OT} is 10 units, which is larger than T_{ST} (9 units) but is smaller than T_{FS} (11 units).

6.0 Optimal ordering

To summarize the previous sections, run time costs increase when we move from a fully-static to a self-timed strategy, but the self-timed strategy is robust with respect to changes in execution times of actors, and the throughput achieved is not significantly affected as long as the variations in execution times are small. If the execution times vary significantly, then to obtain reliably good performance we necessarily have to use a more dynamic strategy such as static assignment or fully dynamic scheduling [4]. It is possible to quantify the effects of variations in actor execution times on the average throughput achieved by a self-timed schedule; such an analysis is however beyond the scope of this paper. Instead we will simply assume that

1. $d(C) > 0$ for every cycle C in G' if the schedule S is deadlock free.

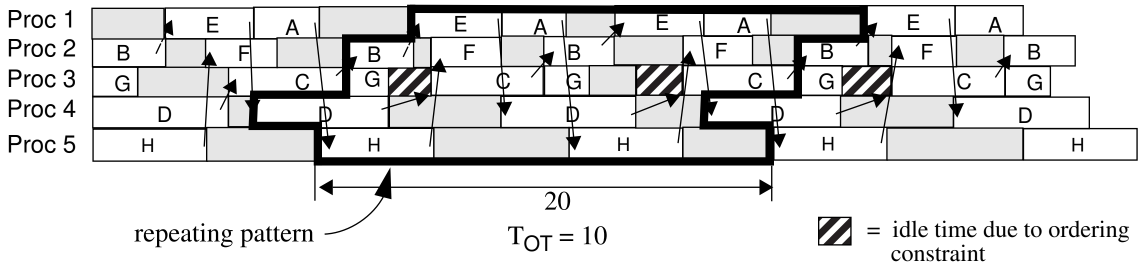


Figure 5. Schedule evolution when the transaction ordering of Fig. 4 is enforced

the variations in execution times are small enough so that a self-timed or an ordered transaction strategy is viable.

Suppose we ignore the above two effects, i.e. we assume zero communication costs and assume the execution times are accurate, and focus on the throughput achieved by the three types of scheduling strategies. At first it seems that for a given processor assignment and ordering of actors on processors, the self-timed approach will perform better than the fully-static or transaction ordered approaches simply because of the manner in which it allows successive iterations to overlap. However, the following result — the main result in this paper — tells us that it is always possible to modify any given fully-static schedule so that it performs nearly as well as it's self-timed counterpart. Stated more precisely:

Claim 1: Given a fully-static schedule $S \equiv \{\sigma_p(v), \sigma_t(v), T_{FS}\}$, let T_{ST} be the average iteration period for the corresponding self-timed schedule (as mentioned before, $T_{FS} \ge T_{ST}$). Then, if $T_{FS} > T_{ST}$ there exists a valid fully-static schedule S' that has the same processor assignment as S, the same order of execution of actors on each processor, but an iteration period of $\lceil T_{ST} \rceil$ (smallest integer greater than T_{ST}). I.e. $S' \equiv \{\sigma_p(v), \sigma'_t(v), \lceil T_{ST} \rceil\}$ where, if actors v_i, v_j are on the same processor (i.e. $\sigma_p(v_i) = \sigma_p(v_j)$) then $\sigma_t(v_i) > \sigma_t(v_j) \Rightarrow \sigma'_t(v_i) > \sigma'_t(v_j)$.

S' is obtained by solving the following set of linear inequalities for σ'_t:

$\sigma'_t(v_i) - \sigma'_t(v_j) \ge t(v_j) - \lceil T_{ST} \rceil \times d(j, i) \;\; \forall \;\; v_j \to v_i$ in G'.

Proof: Let S' have a period equal to T. Then, under the schedule S', the kth starting time of actor v_i is given by:

$$s(v_i, k) = \sigma'_t(v_i) + kT$$

precedence constraints imply:

$$s(v_i, k) \ge s(v_j, k - d(j, i)) + t(v_j) \quad \text{for each edge } v_j \to v_i \text{ in } G'$$

Substituting EQ 1 in EQ 2:

$$\sigma'_t(v_i) - \sigma'_t(v_j) \ge t(v_j) - d(j, i) \times T \;\; \forall \; v_j \to v_i \text{ in } G'$$

Note that the construction of G' ensures that processor assignment constraints are automatically met: if $\sigma_p(v_i) = \sigma_p(v_j)$ and v_i is to be executed immediately after v_j then there is an edge $v_j \to v_i$ in G'. EQ 3 represents a system of $|E'|$ (the number of edges in G') inequalities in $|V|$ unknowns (the quantities $\sigma'_t(v_i)$).

These inequalities fall into a particular class of linear programming problems that can be solved in polynomial time ($O(|E'||V|)$) using the Bellman-Ford shortest-path algorithm [13][14].

Feasible solutions for σ'_t exist for ([14]):

$$T \ge T_{ST} = \max_{\text{cycle C in G}'} \left\{ \frac{\sum_{v \in C} t(v)}{d(C)} \right\}$$

If we set $T = \lceil T_{ST} \rceil$, then the right hand sides of the system of inequalities in (EQ3) are integers, and the Bellman-Ford algorithm yields integer solutions for $\sigma'_t(v)$.

Thus $S' \equiv \{\sigma_p(v), \sigma'_t(v), \lceil T_{ST} \rceil\}$ is a valid fully-static schedule. ❑

Remark: Claim 1 essentially states that a fully static schedule can be modified by skewing the relative starting times of processors so that the resulting schedule has iteration period at most 1 unit larger than that of the corresponding self-timed schedule. It is possible to unfold the graph and generate a fully-static schedule with average period exactly T_{ST}, but the resulting increase in code size may not be worth the benefit of (at most) one time unit decrease in the iteration period.

For example the static schedule S corresponding to Fig. 1 has $T_{FS} = 11 > T_{ST} = 9$ units. Using the procedure outlined in Claim 1, we can skew the starting times of processors in the schedule S to obtain a schedule S', as shown in Fig. 6, that has a period equal to 9 units. Note that the processor assignment and actor ordering in the schedule of Fig. 6 is identical to that of the schedule in Fig. 1.

Claim 1 may not seem useful at first sight: why not obtain a fully-static schedule that has a period $\lceil T_{ST} \rceil$ to begin with, thus eliminating the post-processing step suggested in Claim 1? Recall from Section 3.0 that a fully static schedule is usually obtained using heuristic techniques that are either based on blocked non-overlapped scheduling (using critical path based heuristics) [6] or are based on overlapped scheduling that employs list scheduling heuristics [7][8]. None of these techniques guarantee that the generated fully-static schedule will have an iteration period within one unit of the period achieved if the same schedule were run in a self-timed manner. Thus for a schedule generated using any of these techniques, we

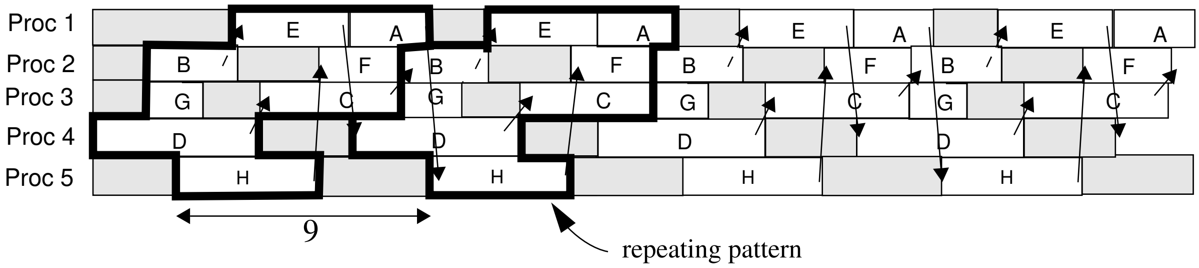


Figure 6. Modified schedule S'

might be able to obtain a gain in performance, essentially for free, by performing the post-processing step suggested in Claim 1. What we have proposed can therefore be added as the final step in existing schedulers. Of course, an exhaustive search procedure like the one proposed in [11] will certainly find the schedule S' directly.

For an ordered transaction strategy, we use the transaction ordering suggested by the modified schedule S' rather than the transaction order from S used in Fig. 5. Thus imposing the transaction order $2 \longrightarrow 1$, $4 \longrightarrow 3$, $5 \longrightarrow 2$, $1 \longrightarrow 4$, $3 \longrightarrow 2$, and $1 \longrightarrow 5$ as in Fig. 6 results in T_{OT} of 9 units instead of 10 that one gets if the transaction order of Fig. 4 is used. Under the transaction order specified by S', $T_{ST} \le T_{OT} \le \lceil T_{ST} \rceil$; thus this order ensures that the average period is within one unit of the unconstrained self-timed strategy. Again, unfolding may be required to obtain a transaction ordered schedule that has period exactly equal to T_{ST}, but the extra cost of a larger controller (to enforce the transaction ordering) outweighs the small gain of at most one unit reduction in the iteration period. Thus for all practical purposes the transaction order specified by S' is the *optimal* order. The "optimality" is of course only under the assumption that the specified execution times of actors are accurate, and under the constraint that the processor assignment and order of execution of actors is kept the same as the original fully static schedule. In other words the transaction order we determine is the best possible one for the available timing information, given the processor assignment and actor ordering.

If the generated fully-static schedule is to be run in a self-timed fashion, then of course there is no need for the post-processing step of Claim 1.

7.0 Conclusions

Determining the order of processor transactions at compile time and enforcing this order at run time leads to a low-cost inter-processor communication mechanism. In this paper we have shown how to determine the best possible transaction order under the given timing information. The procedure, instead of simply extracting the transaction order from the given fully-static schedule, first modifies the fully-static schedule by skewing the starting times of processors. The resulting fully-static schedule has a period within one time unit of the average period obtained if the same schedule were run in a self-timed fashion. Using the transaction ordering specified by the modified schedule results in a transaction ordered schedule with an average period that is at most one unit larger than that of the self-timed strategy. Thus enforcing this particular order on the transactions has almost no penalty over the unconstrained self-timed strategy, under the given actor execution time information.

8.0 References

[1] E. A. Lee, "A Coupled Hardware and Software Architecture for Programmable DSPs," Ph. D. Thesis, Department of EECS, University of California Berkeley, May 1986.

[2] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems," *International Journal of Computer Simulation*, January 1994.

[3] J. M. Rabaey, C. Chu, P. Hoang, and M. Potkonjak, "Fast Prototyping of Datapath Intensive Architectures," *IEEE Design and Test of Computers*, June 1991.

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

[5] E. A. Lee, J. C. Bier, "Architectures for Statically Scheduled Dataflow," *Journal of Parallel and Distributed Computing*, December 1990.

[6] G. C. Sih, "Multiprocessor Scheduling to account for Interprocessor Communication," Ph. D. Thesis, Department of EECS, University of California Berkeley, April 1991.

[7] M. Lam, "A Systolic Array Optimizing Compiler," Ph D. Thesis, Carnegie Mellon University, May 1987.

[8] S. M. H. de Groot, S. Gerez, and O. Herrmann, "Range-Chart-Guided Iterative Data-Flow Graph Scheduling," *IEEE Transactions on Circuits and Systems*, May 1992.

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

[10] S. Y. Kung, P. S. Lewis, and S. C. Lo, "Performance Analysis and Optimization of VLSI Dataflow Arrays," Journal of Parallel and Distributed Computing, Vol. 4, 1987.

[11] D. A. Schwartz, "Synchronous Multiprocessor Realizations of Shift-invariant Flow Graphs," Ph. D. Thesis, Georgia Institute of Technology, June 1985.

[12] S. Sriram, E. A. Lee, "Design and Implementation of an Ordered Memory Access Architecture," *Proceedings of the International Conference on Acoustics Speech and Signal Processing*, April 1993.

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

[14] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, "Introduction to Algorithms," The MIT Press and the McGraw Hill Book Company, Sixth printing, pp. 542-543, 1992.