— Journal of VLSI Signal Processing, Vol. 15, No. 3, pp. 207-220, March1997 —

Determining the Order of Processor Transactions in Statically Scheduled Multiprocessors

S. Sriram, Edward A. Lee
Department of Electrical Engineering and Computer Sciences
Cory Hall, University of California, Berkeley CA94720
Phone: (510)642-0395, (510)642-0455
Email: {sriram,eal}@eecs.Berkeley.EDU

ABSTRACT

This paper addresses embedded multiprocessor implementation of iterative, real-time applications, such as digital signal and image processing, that are specified as dataflow graphs. Scheduling dataflow graphs on multiple processors involves assigning tasks to processors (processor assignment), ordering the execution of tasks within each processor (task ordering), and determining when each task must commence execution. We consider three scheduling strategies: fully-static, self-timed and ordered transactions, all of which perform the assignment and ordering steps at compile time. Run time costs are small for the fully-static strategy; however it is not robust with respect to changes or uncertainty in task execution times. The self-timed approach is tolerant of variations in task execution times, but pays the penalty of high run time costs, because processors need to explicitly synchronize whenever they communicate. The ordered transactions approach lies between the fully-static and self-timed strategies; in this approach the order in which processors communicate is determined at compile time and enforced at run time. The ordered transactions strategy retains some of the flexibility of self-timed schedules and at the same time has lower run time costs than the self-timed approach.

In this paper we determine an order of processor transactions that is nearly optimal given information about task execution times at compile time, and for a given processor assignment and task ordering. The criterion for optimality is the average throughput achieved by the schedule. Our main result is that it is possible to choose a transaction order such that the resulting ordered transactions schedule incurs no performance penalty compared to the more flexible self-timed strategy, even when the higher run time costs implied by the self-timed strategy are ignored.

1 Introduction

In this paper we address embedded multiprocessor implementation of iterative, real-time applications, such as digital signal and image processing (DSP and IP), that are specified as coarse grained Synchronous Data Flow (SDF) graphs [1]. Such applications have been found to be particularly amenable to compile-time (static) scheduling techniques mainly because their control flow is predictable at compile time. In addition, the excessive hardware and processing overhead inherent in dynamic load balancing techniques limits the utility of dynamic strategies in embedded DSP scenarios, especially given the essentially data-independent control flow of most DSP algorithms.

Dataflow programming — apart from being an intuitive specification mechanism for signal processing algorithms — has the obvious advantage that it exposes parallelism in the program. There is a significant body of work on techniques for compile-time scheduling of dataflow graphs on multiple processors (a few examples are [2, 3, 5, 6, 7]). In this paper we assume that we are already given a multiprocessor schedule (which could be determined using the aforementioned techniques, or even manually) along with the SDF description of an application. The main contribution of this paper is to determine the pattern of inter-processor communication (IPC) in a multiprocessor executing the given SDF application according to the specified static schedule with the intent of using this information to reduce run time IPC costs. Specifically, we determine the relative order (in time) in which the inter-processor communications specified by the multiprocessor schedule occur or, equivalently, the order in which access to shared communication resources (for example a shared bus in shared memory machines, or a shared communication link in an interconnection network) is required by the processors. The order that we determine has the property that if we force processors to access communication resources according to this order (as opposed to a more flexible access scheme that places no restriction on the accesses), no penalty in performance is incurred under certain assumptions on timing that we state and make. Determining such an order during compile time and enforcing it at run time eliminates run time arbitration and synchronization costs. This concept has been demonstrated in [10, 11], where the authors discuss the design and implementation of a shared memory multiprocessor system optimized for parallel implementation of embedded DSP applications specified as dataflow graphs. We discuss this multiprocessor further in Section 6.

Our communication or transaction ordering scheme applies to reducing IPC costs in embedded systems that make use of multiple programmable processors (e.g. programmable DSP chips), or a mix of programmable processors and custom VLSI parts (e.g. a dedicated FFT chip). We assume communication occurs through shared memory over a single shared bus, because the simplicity and low hardware cost of this approach makes it attractive, especially in embedded scenarios; however, the ideas we present here can be extended to multiprocessors that employ more elaborate interconnection networks for IPC. The applications we consider in this paper are real-time, where the final implementation of the algorithm must meet hard real-time deadlines to be functionally correct. Reducing IPC costs is particularly important in such applications, because there is usually a premium on processor cycles — for example, if we process audio at a sample rate of 44 KHz on a multiprocessor that consists of processors with a 60 nanosecond cycle time, we have an average of about 380 instructions per processor to complete all operations on each audio sample. IPC can potentially waste precious processor cycles, negating some of the benefits of using multiple processors.

The paper is organized as follows: in Section 2 we review dataflow semantics and the SDF model, we state precisely what we mean by scheduling, and we introduce three compile time scheduling approaches: fully-static, self-timed, and ordered transactions. Section 3 presents some background terminology and notation; Sections 4-6 describe the three compile time scheduling approaches in detail. Section 6 also introduces a graph theoretic inter-processor communication model (the IPC graph), which is then used in Section 7 where we develop an efficient algorithm to determine the access order for communication resources. Finally, Section 8 presents the summary and conclusions.

2 Scheduling Dataflow graphs

We recall that in a dataflow representation an algorithm is represented as a graph where vertices (**actors** in dataflow terminology) are individual computation tasks and directed arcs between them represent flow of data (**tokens**). An actor **fires** (begins execution) when it has sufficient tokens on its input arcs, and it produces tokens on its output arcs when it completes execution. SDF refers to a subclass of dataflow graphs where the actors lack data dependency in their firing patterns: 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. SDF and closely related models have been widely used for representing a significant class of digital signal processing (DSP) algorithms in CAD tools for system-level simulation, design, and prototyping [12, 13, 14, 15]. In this paper we assume that the application is a homogeneous SDF graph, i.e. a graph in which actors always produce and consume exactly one token; henceforth whenever we refer to a dataflow graph (DFG) we imply a homogeneous SDF graph. There is no loss of generality in the homogeneity assumption, because an arbitrary SDF graph can be efficiently transformed into a homogeneous graph [16]. The DFG corresponding to an application may be extracted directly from a block diagram specification (e.g. in Ptolemy [12]) or from an applicative language like Silage (as done for example in Hyper [17]). The dataflow graphs of interest for the purpose of representing DSP algorithms are run repetitively in a non-terminating fashion (i.e. are iterative in nature); tokens flow from source actors to sink actors continually. An **iteration** refers to one complete execution of all the actors in the DFG. These graphs are coarse grain; an atomic actor in the DFG may implement a filtering function for example.

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 (the **processor assignment** step), ordering execution of these actors on each processor (the **actor ordering** step), and determining exactly when each actor fires 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 [18] and [10] 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 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, especially in the context of algorithms that have hard real-time constraints. Which of these can be effectively performed at compile time depends on the information available about the execution time of each actor in the DFG.

The performance metric 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 perform processor assignment and actor ordering, both at compile time, because these strategies appear to be most useful for a significant class of real time DSP algorithms. Although assignment and ordering performed at run time would in general lead to a more flexible implementation (because a dynamic strategy can allow for run time variations in computation load and for operations that display data dependencies) the overhead involved in such a strategy is usually prohibitive and real-time performance guarantees are difficult to achieve. We therefore concentrate on three scheduling strategies that do the assignment and ordering steps at compile time: fully-static, self-timed, and ordered transactions. In the fully-static (FS) strategy, the exact firing time of each actor is also determined at compile time. Such a scheduling style is used in the design of systolic array architectures, for scheduling VLIW processors, and in high-level VLSI synthesis of applications that consist only of operations with guaranteed worst-case execution times. In the self-timed (ST) strategy we retain the processor assignment and actor ordering specified by the FS schedule, but we discard the precise firing times it specifies. Instead, processors determine when to fire an actor by synchronizing with other processors at run time. Such a scheme is employed in wavefront arrays, for example. The ordered transactions (OT) approach falls in between these two strategies; in this approach we determine, based on compile time analysis, an order in which the processors should communicate, and we enforce this pre-determined order at run time. We term this order the **transaction order**.

The FS strategy works only if actor execution time estimates are accurate and data-independent or if tight worst-case estimates are available, whereas the ST strategy is tolerant of variations in execution times of actors, and so is the ST schedule. The ST strategy is the least constrained scheme among all schedules that have the same processor assignment and actor ordering, because in this scheme processors communicate and fire actors as soon as data is available, and hence the only constraints on run time execution are due to data dependencies. Consequently, if we ignore IPC costs in a self-timed schedule, we obtain a lower bound on the iteration period achievable by any schedule for the given processor assignment and actor ordering. We discuss this issue further in Section 5.

The IPC costs are smallest for the FS strategy, comparable (but larger than FS) for the OT strategy, and in general significantly larger for the ST strategy. The FS and the OT strategies require the order of a few instruction cycles for IPC, whereas the ST strategy consumes tens of instruction cycles in typical implementations. In the following sections, we discuss how the higher run time overhead arises for the ST schedule when compared to the FS and OT approaches. In the analysis that follows, however, we ignore the IPC costs in all the three strategies. Doing so allows us to compare the FS and the OT strategies with the idealized self-timed strategy, and hence their iteration periods with the lowest attainable iteration period (the lower bound).

We assume we have reasonably good estimates of actor execution times available to us at compile time; however, these estimates need not be exact, and execution times of actors may even be data-dependent. Thus we allow actors that have different execution times from one iteration of the DFG to the next, as long as these variations are small or rare. This is typically the case when estimates are available for the task execution times, and actual execution times are close to the corresponding estimates with high frequency, but deviations from the estimates of (effectively) arbitrary magnitude can occasionally occur due to phenomena such as cache misses, interrupts, user inputs or error handling. Thus tight worst-case execution time bounds cannot generally be determined for such operations; however, reasonably good execution time estimates can in fact be obtained for these operations, so that static assignment and ordering techniques are viable. Empirical evidence indicates that such a model for actor execution times applies to most DSP applications that are represented as SDF graphs and are implemented on programmable processors [12, 13, 15]. For such applications the performance penalty due to lack of dynamic load balancing is overcome by the much smaller run time scheduling overhead involved when static assignment and ordering is employed.

In the analysis that follows, we develop a model to determine the iteration period of the three scheduling strategies: T_{FS}, T_{OT}, and T_{ST}; however, since we only have estimates for actor execution times, the values we obtain for T_{FS}, T_{OT}, and T_{ST} are only estimates. These estimates will be close to their actual values if the estimates of execution times of actor available to us at compile time are close to their respective run time values. Although the FS strategy, which requires tight worst-case bounds on actor execution times, cannot be used if we allow actors that display variability, we still determine an FS schedule utilizing the available time estimates as a first step towards obtaining an ST or OT schedule: after computing the FS schedule, we simply discard the timing information that is not required.

In Section 6 we show that the relation $T_{FS} \geq T_{OT} \geq T_{ST}$ holds in general for a given processor assignment and actor ordering. We then show how we can obtain an OT schedule that has an iteration period close to the lower bound T_{ST}. Specifically, we show how to determine a transaction order such that $T_{ST} \leq T_{OT} \leq \lceil T_{ST} \rceil$ holds ($\lceil T_{ST} \rceil$ is the smallest integer greater than T_{ST}). Thus we show how to find a near optimal transaction order for a given processor assignment, actor ordering, and set of actor execution time estimates. The "optimality" is in the sense that the iteration period T_{OT} lies within one time unit of its lower bound T_{ST} when the actor execution times at run time are equal to their compile time estimates.

3 Notation

The DFG (assumed to be a homogeneous SDF graph in this paper) is represented as a weighted digraph $G = (V, E)$ where the vertices $v \in V$ represent the actors, and directed edges $(v_j, v_i) \in E$ ((v_j, v_i) is an edge directed from vertex $v_j \in V$ to $v_i \in V$) represent the data dependencies in G. The function $t(v) \in Z^+$, where Z^+ is the set of non-negative integers, assigns an execution time to each actor v (the actual execution time can be interpreted as $t(v)$ cycles of a base clock), and the function $d(v_j, v_i) \in Z^+$ assigns initial tokens to each edge $(v_j, v_i) \in E$ of G. Initial tokens specify data dependencies across iterations of the DFG. For example, if tokens produced by the kth execution of actor A are consumed by the $(k+2)$th execution of actor B, then the edge (A, B) contains two initial tokens, i.e. $d(A, B) = 2$. We represent initial tokens on arcs by bullets on the edges of the DFG. Every edge (v_j, v_i) of G represents the **precedence constraint**:

$$start(v_i, k) \geq end(v_j, k - d(v_j, v_i)) \quad , \forall k \geq d(v_j, v_i) \tag{1}$$

where the function $start(v, k) \in Z^+$ represents the time that the kth execution of the actor v starts; and $end(v, k) \in Z^+$ represents the time the kth execution of v completes. We set $start(v, k) = 0$ and $end(v, k) = 0$ for $k < 0$ as “initial conditions.” Any multiprocessor schedule, in order to correctly implement the computation that G specifies, must respect the precedence constraints implied by the edges of G (i.e. must satisfy (1)). The FS scheduling strategy enforces these constraints by computing appropriate $start(v, k)$ values at compile time and enforcing them at run time. In the ST strategy, processors synchronize at run time to ascertain that the precedence constraints are met, whereas in the OT strategy, precedence constraints are guaranteed by the transaction order that we impose at run time.

We define a path "p" directed from v_1 to v_{n+1} in (V, E) to be a finite, nonempty sequence of edges $((v_1, v_2)(v_2, v_3)(v_3, v_4)\ldots(v_n, v_{n+1}))$, where $(v_k, v_{k+1}) \in E$. Each vertex v_k is said to be *on* the path p, and we indicate this by "$v_k \in p$". A path directed from a vertex to itself is called a cycle. The total number of tokens on a path p is denoted by $D(p)$, i.e.

$$D(p) = \sum_{(v_i, v_j) \in p, (v_i, v_j) \in E} d(v_i, v_j) .$$

Recall that the semantics of a DFG is that an actor v can begin execution 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. In the absence of precise timing information, we set $t(v)$ equal to the estimated execution time of actor v.

4 Fully-static schedule

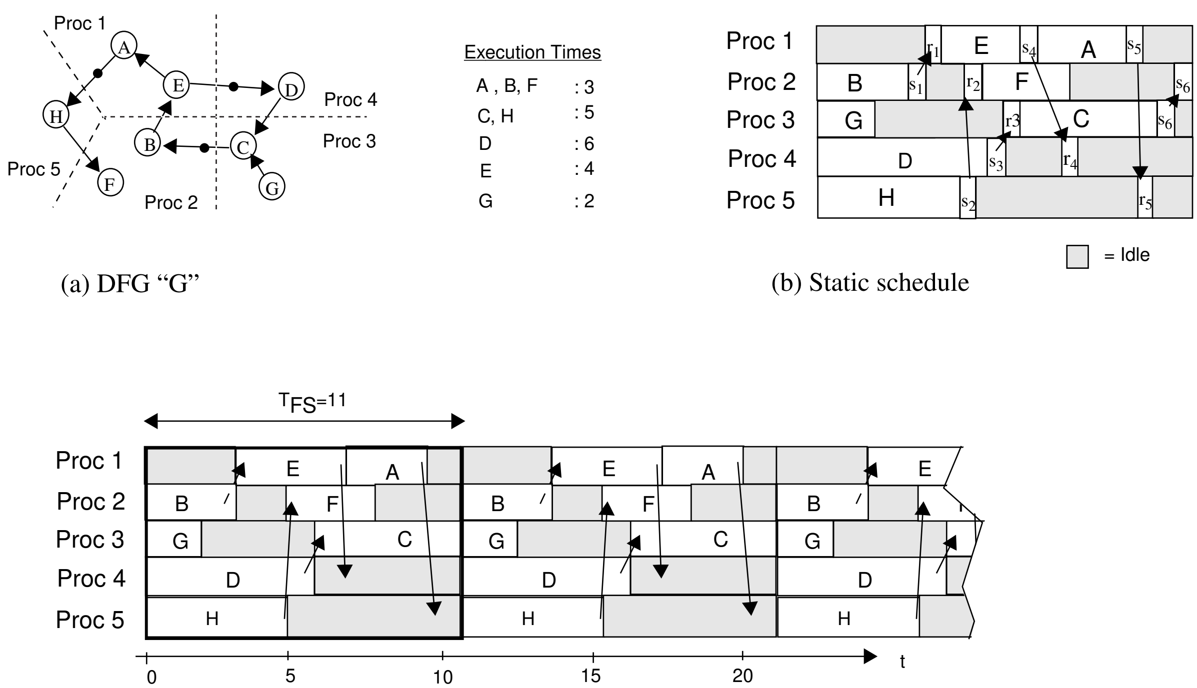
In the fully-static scheduling strategy all the three scheduling operations — assigning actors to processors, ordering of actors on processors, and determining when each actor fires — are performed at compile time [10]. Such a strategy is employed in VLIW arrays [3], and systolic arrays [19, 20]. The classic controller/datapath architecture is also in a sense an example of fully-static scheduling: the controller determines what each functional unit in the datapath does during each clock cycle. Although determining an optimal fully-static schedule (essentially a resource constrained multiprocessor scheduling problem) is NP-hard, several polynomial-time heuristics have been proposed for this problem. Some of these heuristics, e.g. Hu’s classic list scheduling technique [21] and Sih's scheduling heuristics that take IPC costs into account [2], generate a schedule assuming the program terminates after a finite number of iterations. Others explicitly take overlapping of successive iterations of the DFG into account, for example the techniques in [3, 4, 5] adapt list scheduling to generate overlapped schedules. 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 ($start(v, k)$). Since the firing times are enforced by a finite state controller in practice, a fully-static schedule is also constrained to be periodic, i.e. $start(v, k) = \sigma_t(v) + kT_{FS}$; $\sigma_t(v)$ is the starting time of the first execution of actor v (i.e. $start(v, 0) = \sigma_t(v)$) and T_{FS} is the iteration period. The $\sigma_p(v)$ function and the $\sigma_t(v)$ values are chosen so that all data precedence constraints (of equation (1)) and resource constraints are met in the final schedule. Thus a fully-static schedule specifies the triple $\{\sigma_p(v), \sigma_t(v), T_{FS}\}$. Clearly, the throughput for such a schedule is T_{FS}^{-1}.

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 of Fig. 1(a). Edges in G that cross processor boundaries after scheduling represent IPC; we call such edges **IPC edges**. 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 negligible time, then T_{FS} for this example is 11 units. The σ_p and σ_t values are as follows:

$\sigma_p(A) = \sigma_p(E) = 1$, $\sigma_p(B) = \sigma_p(F) = 2$, $\sigma_p(C) = \sigma_p(G) = 3$, $\sigma_p(D) = 4$, and $\sigma_p(H) = 5$;

$\sigma_t(A) = 7$, $\sigma_t(B) = \sigma_t(D) = \sigma_t(G) = \sigma_t(H) = 0$, $\sigma_t(C) = 6$, $\sigma_t(E) = 3$, and $\sigma_t(F) = 5$.

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 [16, 6]. The unfolded graph contains u copies of each actor of the original graph. In this case σ_p and σ_t are defined for all the vertices of the *unfolded* graph (i.e. σ_p and σ_t are defined for u invocations of each actor); T_{FS} is the iteration period for the unfolded graph, and the average iteration period for the original graph is then $\frac{T_{FS}}{u}$. In the remainder of this paper, we assume we are dealing with the unfolded graph and we 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.



(a) DFG "G"

(b) Static schedule

(c) Fully-static execution

Figure 1. Fully-static schedule on five processors

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 (σ_t) 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 when computing the FS schedule. Such worst-case estimates are often used when scheduling hardware in high-level synthesis [22]. For programmable processors, however, determining tight upper bounds on execution times is not always possible — when the object code is compiled from a high level language for example, or when processors employ pipelining and other instruction-level parallelism techniques, or due to phenomena such as cache misses or error handling. 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.

5 Self-timed schedule

The fully-static approach introduced in the previous section cannot be used when actors have variable execution times, because the FS approach will not guarantee sender-receiver synchronization in such a situation. An obvious strategy for solving this problem is to introduce explicit synchronization whenever processors communicate. This leads to the self-timed scheduling strategy of [18]. In this strategy we first obtain an FS schedule using the execution time estimates, but we only retain the processor assignment σ_p and the ordering of actors on each processor as specified by σ_t, and 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, that 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 are 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 [23]; however, since we are mainly interested in determining the best performance achievable by a ST strategy, we do not consider buffer optimization in this paper. Instead, we assume that the buffers are large enough so that their finite sizes do not affect the throughput of the system of processors. Such an ST strategy is used in wavefront arrays [23], and in situations where a fully-static approach is impractical due variability in execution times of operations.

An ST strategy is robust with respect to changes in execution times of actors, because sender-receiver synchronization is performed at run time. Such run time synchronization, however, implies higher IPC costs compared to the fully-static strategy because of the need for synchronization (e.g. using semaphore management). In addition the ST strategy faces arbitration costs: the FS schedule guarantees mutually exclusive access of shared communication resources, whereas shared resources need to be arbitrated at run time in the ST schedule. Consequently, whereas IPC in the FS schedule simply involves reading and writing from shared memory (no synchronization or arbitration needed), implying a cost of a few processor cycles for IPC, the ST strategy requires of the order of tens of processor cycles, unless special hardware is employed for synchronization and arbitration [11].

Another feature of the ST strategy is that, since processors do not re-synchronize at the end of each iteration of the DFG (there is no need for such synchronization), successive iterations of the DFG overlap in a natural manner. Fig. 2 shows how the ST schedule corresponding to the fully-static schedule in Fig. 1 evolves. This is of course an idealized scenario where IPC costs are ignored; these costs are not negligible in practice as mentioned before. Note that the ST schedule in Fig. 2 eventually settles to a periodic pattern consisting of two iterations of the DFG. It can be shown that a ST schedule always settles down into a periodic execution pattern, but the number of iterations spanned by the periodic repeating pattern can be exponential in the size of the DFG [24]. From Fig. 2, therefore, the average iteration period under the self-timed schedule is 9 units. The average iteration period (T_{ST}) for such an idealized self-timed schedule represents a **lower bound** on the iteration period achievable by *any* schedule that maintains the same processor assignment and actor ordering. This is because the only run time constraint on processors that the ST schedule imposes is due to data dependencies: each processor executes actors assigned to it (including the communication actors) according to the prescribed actor ordering as soon as data is available for each actor. Any other schedule that maintains the same processor assignment and actor ordering, and respects data precedences in G, cannot result in an execution where actors fire earlier than they do in the idealized ST schedule. Thus among all schedules with the same processor assignment and actor ordering, the idealized ST schedule achieves the minimum iteration period. In particular, the overlap of successive iterations of the DFG in the idealized ST schedule

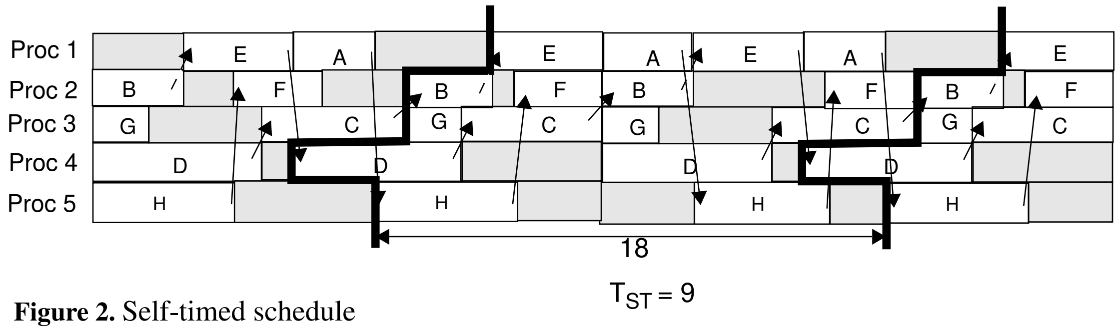


Figure 2. Self-timed schedule

ensures that $T_{ST} \leq T_{FS}$.

We would like to maintain the unconstrained execution of the self-timed schedule, and also eliminate the high IPC overhead that this strategy involves. To do this, we first develop an analytical model for self-timed execution.

The $start(v, k)$ values in the ST schedule are determined by how the schedule evolves at run time as opposed to the FS schedule where these values are computed at compile time and enforced at run time. We model the evolution of a ST schedule using a DFG $G_{ipc} = (V, E_{ipc})$ derived from the original SDF graph $G = (V, E)$. The graph G_{ipc}, which we refer to as the **IPC graph**, models the sequential execution of the actors of G assigned to the same processor, and it models constraints due to IPC.

The IPC graph has the same vertex set V as G. The ST schedule specifies the actors assigned to each processor, and the order in which they execute. For example in Fig. 1, Processor 1 executes E and then A repeatedly. We model this in G_{ipc} by constructing a cycle around the vertices corresponding to E and A, and placing a delay on the edge from A to E. The delay-free edge from E to A represents the fact that the kth execution of E precedes the kth execution of A, and the edge from A to E with a delay represents the fact that the kth execution of E can occur only after the $(k-1)$th execution of A has completed. Thus if actors $v_1, v_2, \ldots, v_n$ are assigned to the same processor in that order, then G_{ipc} would contain a cycle $((v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n), (v_n, v_1))$, with $d(v_n, v_1) = 1$. This reflects the fact that actors assigned to the same processor run sequentially.

For each IPC edge in G we add an IPC edge e in G_{ipc} between the same actors. We also set $d(e)$ equal to the delay on the corresponding edge in G. Thus in the example of Fig. 1 we add an IPC edge from E to D in G_{ipc} with a single delay on it. The delay corresponds to the fact that execution of E is allowed to lag the execution of D by one iteration. An IPC edge represents a buffer implemented in shared memory, and initial tokens on the IPC edge are used to initialize the shared buffer. IPC can be modeled explicitly by including *send* and *receive* actors in G_{ipc}, and IPC costs can be modeled by setting execution times of these actors appropriately.

G_{ipc} for the example of Fig. 1(a), (b) is shown in Fig. 3; communication actors are not included for the sake of clarity. Note, for instance, that actors B and F are linked into a cycle in G_{ipc}, reflecting 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 ensures that the kth firing of B always pre-

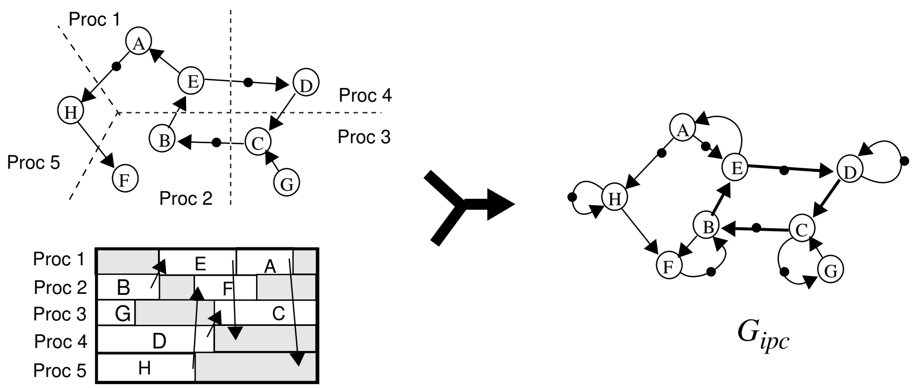


Figure 3. Construction of G_{ipc} from G and from the static schedule

cedes the kth firing of F on Proc 2.

As per dataflow semantics each edge (v_j, v_i) of G_{ipc} represents the precedence constraint (similar to equation (1)):

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

The constraints in (2) are both due to IPC edges (representing synchronization between processors) and due to edges that represent serialization of actors assigned to the same processor. Since in the ST schedule actors fire as soon as data is available at all their input edges, the evolution of the ST scheduled is precisely modeled by the "as soon as possible" (ASAP) execution of G_{ipc}. Since the execution time of an actor v is $t(v)$, we can substitute

$$end(v_j, k) = start(v_j, k) + t(v_j)$$

in (2) to obtain

$$start(v_i, k) \geq start(v_j, k - d(v_j, v_i)) + t(v_j) \quad \forall (v_j, v_i) \in E_{ipc} \tag{3}$$

ASAP execution implies:

$$start(v_i, k) = max\{ start(v_j, k - d(v_j, v_i)) + t(v_j) | (v_j, v_i) \in E_{ipc} \} \tag{4}$$

G_{ipc} is really an instance of Reiter's "computation graph" [26], also known as a "Timed Marked graph" in Petrinet theory [24, 25] (the vertices correspond to the transitions in the marked graph, edges correspond to places, and the initial tokens correspond to the initial marking). It is well known (from these references among others) that the average iteration period for the ASAP execution of such a graph is given by:

$$T_{ST} = \max_{\text{cycle C in } G_{ipc}} \left\{ \frac{\sum_{v \in C} t(v)}{D(C)} \right\} \tag{5}$$

Note that $D(C) > 0$ for every cycle C in G_{ipc} if the schedule S is deadlock free, because a cycle with zero delay implies a circular dependency in the schedule. The quantity on the right hand side of (5) is commonly referred to as the **maximum cycle mean** of G_{ipc}. A cycle in G_{ipc} that maximizes the quotient in the RHS of (5) is called a **critical cycle**.

We refer to [24, 26, 25] for the proof of (5). If we only have execution time estimates available instead of exact values, and we set $t(v)$ above to be these estimated values, then we obtain the *estimated* iteration period by calculating T_{ST}.

For example, the value of T_{ST} obtained from G_{ipc} in Fig. 3 is 9 units (corresponding to the critical cycle B→E→D→C→B, which has total weight of 18 and has two initial tokens on it). Thus the average iteration period for the ST schedule of Fig. 2 is 9. Note that T_{ST} is a rational number, but it is not necessarily an integer.

6 Ordered transactions

The self-timed scheduling strategy in the previous section introduces synchronization checks whenever processors communicate in order to allow for variations in actor execution times. These checks introduce run time synchronization and arbitration costs. The ordered transactions (OT) strategy alleviates some of these costs, and in doing so, trades off some of the flexibility afforded by the ST approach. In the OT strategy we first obtain a fully-static schedule using the execution time estimates, but we discard the precise timing information specified in the fully-static schedule; we retain the processor assignment (σ_p) and actor ordering on each processor as specified by σ_t; and, in addition, we also retain the order in which processors communicate with one another and we enforce this order at run time [10]. We formalize the concept of transaction order below.

Suppose there are k IPC edges $(s_1, r_1), (s_2, r_2), \ldots, (s_k, r_k)$ — where each (s_i, r_i) is a *send-receive* pair — in the FS schedule we obtain. Let R be the set of *receive* actors, and S be the set of *send* actors (i.e. $R \equiv \{r_1, r_2, \ldots, r_k\}$ and $S \equiv \{s_1, s_2, \ldots, s_k\}$). We define a **transaction order** to be a sequence $O = (v_1, v_2, v_3, \ldots, v_{2k-1}, v_{2k})$, where $\{v_1, v_2, \ldots, v_{2k-1}, v_{2k}\} \equiv S \cup R$ (each communication actor is present in the sequence O). We say a transaction order O (as defined above) is **imposed** on a multiprocessor if at run time the *send* and *receive* actors are forced to execute in the sequence specified by O. That is, if $O = (v_1, v_2, v_3, \ldots, v_{2k-1}, v_{2k})$, then imposing O means ensuring the constraints: $end(v_1, k) \le start(v_2, k)$, $end(v_2, k) \le start(v_3, k)$, $\ldots$, $end(v_{k-1}, k) \le start(v_k, k)$; $\forall k \ge 0$.

The OT schedule can be viewed as an ST schedule with the added transaction order constraints specified by O. Recall that an ST schedule is modeled using the construction G_{ipc}. Also, an edge (v_i, v_j) in G_{ipc} that has $d(v_i, v_j) = 0$ represents the constraint $start(v_j, k) \ge end(v_i, k)$ $\forall k \ge 0$, as per equation (2). Imposing a transaction order O can therefore be modeled by adding a set of edges between the *send* and *receive* actors in G_{ipc}. Thus, just as the ST schedule was described by $G_{ipc} = (V, E_{ipc})$, the OT schedule is described by $(V, E_{ipc} \cup E_{OT})$, where E_{OT} are the edges due to the transaction order constraints.

After an FS schedule is obtained using the execution time estimates, the transaction order is obtained from the σ_t function of the FS schedule: we simply set the transaction order to be $O = (v_1, v_2, v_3, \ldots, v_{2k-1}, v_{2k})$ such that

$$\sigma_t(v_1) \le \sigma_t(v_2) \le \ldots \le \sigma_t(v_{2k-1}) \le \sigma_t(v_{2k}) \quad .$$

The transaction order can therefore be determined by sorting the set of communication actors ($S \cup R$) according to their start times σ_t. Fig. 4 shows an example of how such an order could be derived from a given static schedule.

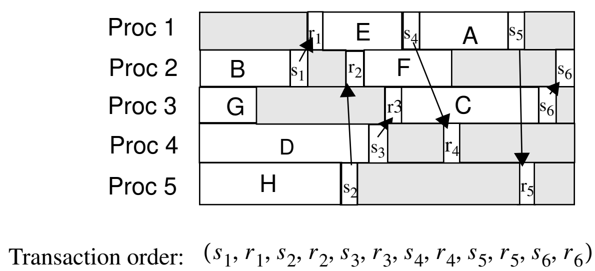


Figure 4. One possible transaction order derived from the fully-static schedule

The transaction order is enforced at run time by a controller implemented in hardware. 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 ordering processor accesses to them; this results in an efficient IPC mechanism at low hardware cost. We have built a prototype four processor DSP board, called the Ordered Memory Access (OMA) architecture, that demonstrates the ordered transactions concept. The OMA board utilizes shared memory and a single shared bus for IPC — the sender writes data to a particular shared memory location and the receiver reads that location. In this multiprocessor a very simple controller on the board enforces the pre-determined transaction order at run time, thus eliminating the need for run time bus arbitration or semaphore synchronization. This results in efficient IPC (comparable to the FS strategy) at relatively low hardware cost. The OMA multiprocessor is described in detail in [11].

As in the ST scenario, the OT strategy is tolerant of variations in execution times of actors, because the transaction order enforces correct sender-receiver synchronization; however, this strategy is more constrained than ST scheduling, which allows the order that communication actors fire to vary at run time. This is also apparent from our graph based execution model: the IPC graph for the OT schedule has additional constraint edges E_{OT}. For example, the transaction order in Fig. 4 enforces the order $(s_1, r_1, s_2, r_2, s_3, r_3, s_4, r_4, s_5, r_5, s_6, r_6)$, but the ST schedule allows reordering among these IPCs at run time. In fact we observe from Fig. 2 that once the ST schedule settles into a periodic pattern, IPCs in successive iterations are ordered differently: in the first iteration the order in which IPCs occur is indeed $(s_1, r_1, s_2, r_2, s_3, r_3, s_4, r_4, s_5, r_5, s_6, r_6)$, but this order changes in successive iterations; once the schedule settles into a periodic pattern, the order alternates between:

$(s_3, r_3, s_1, r_1, s_2, r_2, s_4, r_4, s_6, r_6, s_5, r_5)$ and $(s_1, r_1, s_3, r_3, s_4, r_4, s_2, r_2, s_5, r_5, s_6, r_6)$. In contrast, if we use the order in Fig. 4 as the transaction order, the resulting OT schedule evolves as shown in Fig. 5. Notice that enforcing this schedule introduces idle time; as a result, the average iteration period, T_{OT}, is 10 units, which is of course larger than the iteration period of the ideal ST schedule T_{ST} (9 units) but is smaller than T_{FS} (11 units). In general $T_{FS} \geq T_{OT} \geq T_{ST}$: the ST

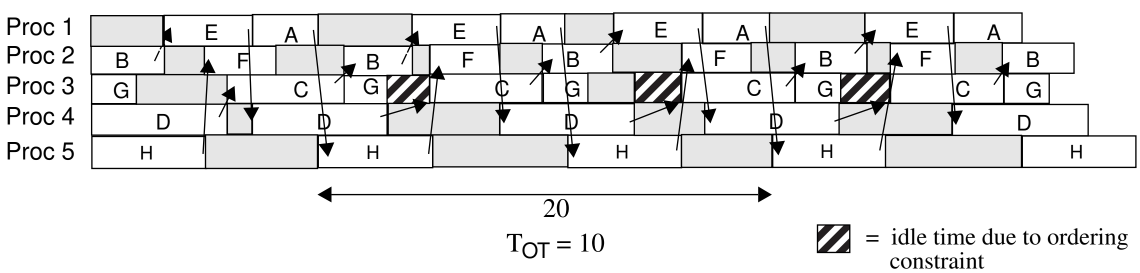


Figure 5. Schedule evolution when the transaction ordering of Fig. 4 is enforced

schedule only has assignment and ordering constraints, the OT schedule has the transaction ordering constraints in addition to the constraints in the ST schedule, whereas the FS schedule has exact timing constraints that subsume the constraints in the ST and OT schedules.

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

Recall that the OT schedule is obtained by first determining a fully-static schedule and then discarding the precise firing time information, and can therefore be viewed as a self-timed schedule with the added transaction order constraints. Clearly there is some flexibility in the transaction order we choose at compile time. The question then is: how do we pick one that is the "best" in the sense that the restrictions imposed by it do not sacrifice throughput. In other words, how do we pick a transaction order such that the resulting OT schedule has iteration period close to T_{ST}, at least when execution times of actors are equal to their estimated values. One possibility is to derive the transaction order from the repeating pattern that the ST schedule settles into. That is, instead of using the transaction order of Fig. 4, if we enforce the transaction order that repeats over two iterations in the evolution of the ST schedule of Fig. 2, the OT schedule would "mimic" the ST schedule exactly, and we would obtain an OT schedule that performs as well as the ideal ST schedule, and yet involves low IPC costs in practice. Unfortunately, the number of iterations that the repeating pattern spans depends on the structure of G_{ipc}, and it can be exponential in the size of the DFG [24]. Consequently the memory requirements on the controller that enforces the transaction order can be prohibitively large in certain cases. We therefore restrict ourselves to determining and enforcing a transaction order that spans only one iteration of the DFG; the following section discusses how such an "optimal" transaction order is obtained.

7 Optimal ordering

To summarize the previous sections, the fully-static schedule has the least run time overhead for IPC, but is not practical when tight worst-case execution time bounds are not available for all actors; the self-timed schedule retains only the processor assignment and ordering from the FS schedule, hence is more flexible, but has higher run time overhead. The ordered transactions approach lies in between.

The ordered transactions strategy is cheaper in terms of IPC costs than the self-timed approach; however, the ST approach is more flexible, and if we ignore its more expensive IPC mechanism, the average iteration period of the OT scheme may be larger than that of the ST, as shown in the example of Fig. 5. In this section we show how to determine an order O^* on the IPCs in the schedule such that, even if we ignore the larger communication costs associated with the ST schedule, imposing O^* yields an OT schedule that has iteration period within one unit of the ideal ST schedule ($T_{ST} \le T_{OT} \le \lceil T_{ST} \rceil$). Thus imposing the order we determine results in essentially no loss in performance over an unrestrained schedule, and at the same time we get the benefit of cheaper IPC.

It should be noted, however, that the resulting transaction-ordered schedule is more sensitive to variations in execution times: even though the computations performed using the OT schedule are robust with respect to execution time variations (the transaction order ensures correct sender-receiver synchronization), the ordering restriction makes the iteration period more dependent on execution time variations than the ideal ST schedule. Nevertheless, if the actual execution times do not deviate significantly from the estimated values, the difference in performance of the ST and OT strategies is minimal. If the execution times do in fact vary significantly, then even an ST strategy is not practical: it then becomes necessary to use a more dynamic strategy such as static assignment or fully dynamic scheduling [4] to make the best use of computing resources. It is possible to quantify the effects of variations in actor execution times on the average throughput achieved by an ST or an OT schedule; such an analysis is however beyond the scope of this paper. Instead, we simply assume that the variations in execution times are small enough so that an ST or an OT strategy is viable. Under this assumption we argue that it is in fact wiser to use the OT strategy rather than ST because of the cheaper IPC of the OT strategy, and because of our ability to determine the transaction order O^* such that the ordering constraints do not sacrifice performance: if the execution times of actors are close to their estimates, the OT schedule with O^* as the transaction order has iteration period close to the minimum achievable period T_{ST}.

Our approach to determining the transaction order O^* is to modify a given fully-static schedule so that the resulting FS schedule has T_{FS} equal to $\lceil T_{ST} \rceil$, and then to derive the transaction order from that modified schedule. Intuitively it appears that, for a given processor assignment and ordering of actors on processors, the ST approach *always* performs better than the FS or OT approaches ($T_{FS} > T_{OT} > T_{ST}$) simply because it allows successive iterations to overlap. The following result, however, tells us that it is always possible to modify any given fully-static schedule so that it performs nearly as well as its 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 ST schedule (as mentioned before, $T_{FS} \ge T_{ST}$). Suppose $T_{FS} > T_{ST}$; then, 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 $\lceil T_{ST} \rceil$. That is, $S' \equiv \{\sigma_p(v), \sigma'_t(v), \lceil T_{ST} \rceil\}$ where, if actors v_i, v_j are on the same processor (i.e. $\sigma_p(v_i) = \sigma_p(v_j)$) then $\sigma_t(v_i) > \sigma_t(v_j) \Rightarrow \sigma'_t(v_i) > \sigma'_t(v_j)$. Furthermore, S' is obtained by solving the following set of linear inequalities for σ'_t:

$$\sigma'_t(v_j) - \sigma'_t(v_i) \le \lceil T_{ST} \rceil \times d(v_j, v_i) - t(v_j) \quad \text{for each edge } (v_j, v_i) \text{in } G_{ipc} \; .$$

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:

$$start(v_i, k) = \sigma'_t(v_i) + kT \tag{6}$$

Also, data precedence constraints imply (as in equation (3)):

$$start(v_i, k) \ge start(v_j, k-d(v_j, v_i)) + t(v_j) \quad \text{for each edge } (v_j, v_i) \text{ in } G_{ipc} \tag{7}$$

Substituting (6) in (7):

$$\sigma'_t(v_i) + kT \ge \sigma'_t(v_j) + (k-d(v_j, v_i))T + t(v_j) \quad \forall (v_j, v_i) \in E_{ipc}$$

That is:

$$\sigma'_t(v_j) - \sigma'_t(v_i) \le T \times d(v_j, v_i) - t(v_j) \quad \forall (v_j, v_i) \in E_{ipc} \tag{8}$$

Note that the construction of G_{ipc} 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, v_i) in G_{ipc}. The relations in (8) represent a system of $|E_{ipc}|$ inequalities in $|V|$ unknowns (the quantities $\sigma'_t(v_i)$).

The system of inequalities (8) is a difference constraint problem that can be solved in polynomial time ($O(|E_{ipc}||V|)$) using the Bellman-Ford shortest-path algorithm [27, 28]. The details of this approach are well described in [28]; the essence of it is to construct a constraint graph that has one vertex for each unknown $\sigma'_t(v_i)$. Each difference constraint is then represented by an edge between the vertices corresponding to the unknowns, and the weight on that edge is set to be equal to the RHS of the difference constraint. A "dummy" vertex is added to the constraint graph, and zero weight edges are added from the dummy vertex to each of the remaining vertices in the constraint graph. Then, setting the value of $\sigma'_t(v_i)$ to be the weight of the shortest path from the dummy vertex to the vertex that corresponds to $\sigma'_t(v_i)$ in the constraint graph results in a solution to the system of inequalities, if indeed a solution exists. A feasible solution exists if and only if the constraint graph does not contain a negative weight cycle [28], which is equivalent to the following condition:

$$T \geq \max_{\text{cycle C in } G_{ipc}} \left\{ \frac{\sum_{v \in C} t(v)}{D(C)} \right\} \text{; and, from (5), this is equivalent to } T \geq T_{ST}.$$

If we set $T = \lceil T_{ST} \rceil$, then the right hand sides of the system of inequalities in (8) are integers, and the Bellman-Ford algorithm yields integer solutions for $\sigma'_t(v)$. This is because the weights on the edges of the constraint graph, which are equal to the RHS of the difference constraints, are integers if T is an integer; consequently, the shortest paths calculated on the constraint graph are integers.

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 an FS schedule can be modified by skewing the relative starting times of processors so that the resulting schedule has iteration period less than $(T_{ST} + 1)$; the resulting iteration period lies within one time unit of its lower bound for the specified processor assignment and actor ordering. 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 is usually not 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 (7), 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. The values $\sigma'_t(v)$ are: $\sigma'_t(A) = 9$, $\sigma'_t(B) = \sigma'_t(G) = 2$, $\sigma'_t(C) = 6$,

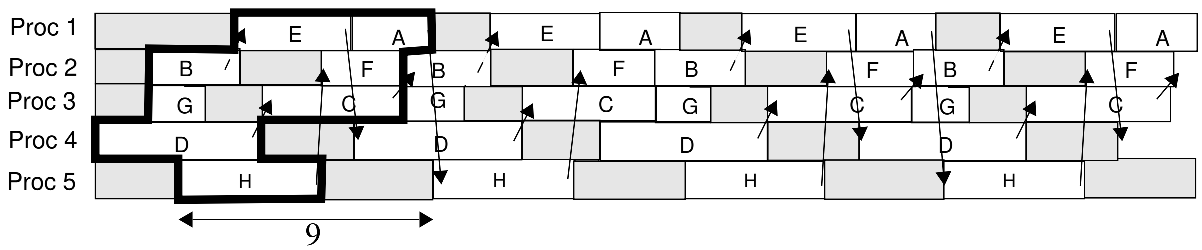


Figure 6. Modified schedule S'

$\sigma'_t(D) = 0$, $\sigma'_t(E) = 5$, $\sigma'_t(F) = 8$, and $\sigma'_t(H) = 3$.

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 an FS schedule is usually obtained using heuristic techniques that are either based on blocked non-overlapped scheduling (which use critical path based heuristics) [2] or are based on overlapped scheduling techniques that employ list scheduling heuristics [3, 5]. None of these techniques guarantee that the generated FS 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 might be able to obtain a gain in performance, essentially for free, by performing the post-processing step suggested in Claim 1. What we propose can therefore be added as an efficient post-processing step in existing schedulers. Of course, an exhaustive search procedure like the one proposed in [7] will certainly find the schedule S' directly.

We set the transaction order O^* to be the transaction order suggested by the modified schedule S' (as opposed to the transaction order from S used in Fig. 5). Thus $O^* = (s_1, r_1, s_3, r_3, s_2, r_2, s_4, r_4, s_6, r_6, s_5, r_5)$. Imposing the transaction order O^* 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', $T_{ST} \leq T_{OT} \leq \lceil T_{ST} \rceil$; thus imposing the order O^* ensures that the average period is within one unit of the unconstrained ST 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, unless T_{ST} is close to 1 ($T_{ST} \cong 1$), which is a rare situation. Thus for all practical purposes O^* is the *optimal* transaction order. The "optimality" is in the sense that the transaction order O^* we determine statically is the best possible one, given the timing information available at compile time.

The periodic schedule in (6) is similar to *affine schedules* in systolic array literature[9, 8]. Affine schedules are usually defined over a multi-dimensional index space. In our formulation, however, the index space has only one dimension, representing time, and the scaling of the index is done by the iteration period T. This allows us to solve the linear program involved using a low-complexity shortest path based approach.

8 Conclusions

Determining the order of processor transactions at compile time and enforcing this order at run time leads to a low-cost IPC mechanism. In this paper we have shown how to determine the best possible transaction order for the timing information available at compile time. The procedure, instead of simply extracting the transaction order from a 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 an ideal self-timed fashion. Using the transaction order specified by the modified schedule results in an ordered transaction 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 results in essentially no penalty over the unconstrained self-timed strategy, under the reasonable assumption that $T_{ST} \gg 1$. Our procedure is useful as an efficient post-processing step for existing scheduling algorithms, both for generating an improved fully-static schedule and for determining an optimal transaction order.

REFERENCES

[1] E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Dataflow Programs for Digital Signal Processing," *IEEE Trans. on Computers*, vol. C-36, no. 2, February 1982.

[2] G. C. Sih, "Multiprocessor Scheduling to account for Interprocessor Communication," Ph. D. Thesis, Memorandum No. UCB/ERL M91/29, Electronics Research Laboratory, University of California at Berkeley, April 1991.

[3] M. Lam, "Software Pipelining: An Effective Scheduling Technique for VLIW Machines," *Proceedings of the SIGPLAN 1988 Conference on Programming Language Design and Implementation*, June 1988, pp. 318-328.

[4] V. H. Allan, R. B. Jones, R. M. Lee, S. J. Allan, "Software pipelining," *ACM Computing Surveys*, Sept. 1995, vol.27, (no.3):367-432.

[5] 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, pp. 351-364.

[6] K. Parhi, and D. G. Messerschmitt, "Static Rate-optimal Scheduling of Iterative Data-flow Programs via Optimum Unfolding," *IEEE Transactions on Computers*, vol. 40, no. 2, February 1991, pp. 178-194.

[7] D. A. Schwartz, and T. P. Barnwell III, "Cyclo-Static Solutions: Optimal Multiprocessor Realizations of Recursive Algorithms," *VLSI Signal Processing II,* IEEE Special Publications, June 1985, pp. 117-128.

[8] S. K. Rao, and T. Kailath, "Regular Iterative Algorithms and their Implementation on Processor Arrays," *Proceedings of the IEEE*, Vol. 76, No. 3, 1988.

[9] A. Darte, Y. Robert, "Constructive methods for scheduling uniform loop nests," *IEEE Transactions on Parallel and Distributed Systems*, Aug. 1994, vol.5, (no.8):814-22.

[10] E. A. Lee, J. C. Bier, "Architectures for Statically Scheduled Dataflow," *Journal of Parallel and Distributed Computing*, vol. 10, December 1990, pp. 333-348.

[11] S. Sriram, and E. A. Lee, "Design and Implementation of an Ordered Memory Access Architecture," *Proceedings of the International Conference on Acoustics Speech and Signal Processing*, vol. 1, April 1993, pp. 345-348.

[12] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems," *International Journal of Computer Simulation*, vol. 4, January 1994, pp. 155-182.

[13] 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.

[14] D. B. Powell, E. A. Lee, and W. C. Newman, "Direct Synthesis of Optimized DSP Assembly Code from Signal Flow Block Diagrams," *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, San Francisco, March 1992.

[15] S. Ritz, M. Pankert, and H. Meyr, "High Level Software Synthesis for Signal Processing Systems," *Proceedings of the International Conference on Application Specific Array Processors*, Berkeley, August 1992, pp.679-693.

[16] E. A. Lee, "A Coupled Hardware and Software Architecture for Programmable DSPs," Ph. D. Thesis, Department of Electrical Engineering and Computer Sciences, University of California Berkeley, May 1986.

[17] J. M. Rabaey, C. Chu, P. Hoang, and M. Potkonjak, "Fast Prototyping of Datapath Intensive Architectures," *IEEE Design and Test of Computers*, vol. 8, no. 2, June 1991, pp. 40-51.

[18] E. A. Lee, and S. Ha, "Scheduling Strategies for Multiprocessor Real-Time DSP," *Proceedings of the Globecom Conference*, Dallas Texas, November 1989, pp. 1279-1283.

[19] S. Borkar *et. al.*, "iWarp: An Integrated Solution to High-Speed Parallel Computing", *Proceedings of Supercomputing 1988 Conference*, Orlando, Florida, 1988.

[20] L. Thiele, "Resource constrained scheduling of uniform algorithms," *Journal of VLSI Signal Processing*, Aug. 1995, vol.10, (no.3):295-310

[21] T. C. Hu, "Parallel Sequencing and Assembly Line Problems," *Operations Research*, vol. 9(6), November 1961, pp. 841-848.

[22] G. De Micheli, "*Synthesis and Optimization of Digital Circuits*," McGraw Hill Inc., New Jersey, 1994.

[23] 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, pp. 592-618.

[24] F. Baccelli, G. Cohen, G. J. Olsder, and J.-P. Quadrat, "*Synchronization and Linearity*," John Wiley & Sons Inc., New York, 1992.

[25] J. L. Peterson, "*Petri Net Theory and the Modelling of Systems*", Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1981.

[26] R. Reiter, Scheduling Parallel Computations, *Journal of the Association for Computing Machinery*, vol. 15. No. 4, October 1968, pp. 590-599.

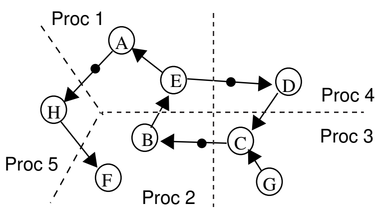
[27] E. L. Lawler, "*Combinatorial Optimization: Networks and Matroids*," Holt, Rinehart and Winston, New York, pp. 65-80, 1976.

[28] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, "*Introduction to Algorithms*," The MIT Press and the McGraw Hill Book Company, Sixth printing, Chapter 25, pp. 542-543, 1992.

Acknowledgment of Support

This research was 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), Semiconductor Research Corporation (project 94-DC-008), National Science Foundation (MIP-9201605), Office of Naval Technology (via Naval Research Laboratories), the State of California MICRO program, and the following companies: Bell Northern Research, Cadence, Dolby, Hitachi, Mentor Graphics, Mitsubishi, NEC, Pacific Bell, Philips, Rockwell, and Synopsys.

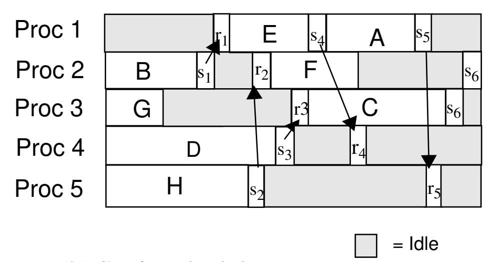
List of Figure Captions



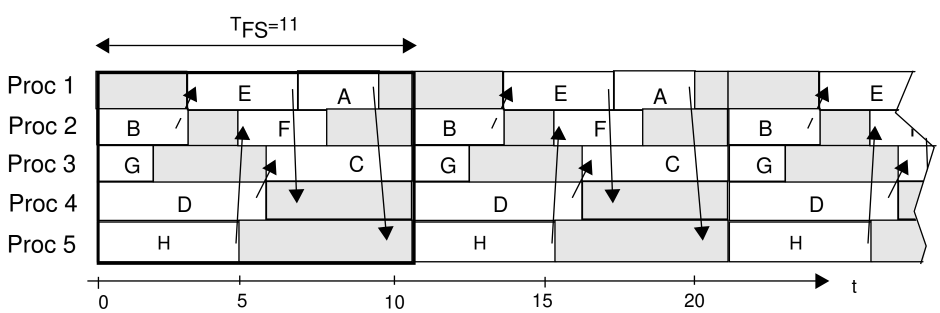
(a) DFG “G”

Execution Times

A , B, F	: 3
C, H	: 5
D	: 6
E	: 4
G	: 2



(b) Static schedule



(c) Fully-static execution

Figure 1 (a), (b), (c)

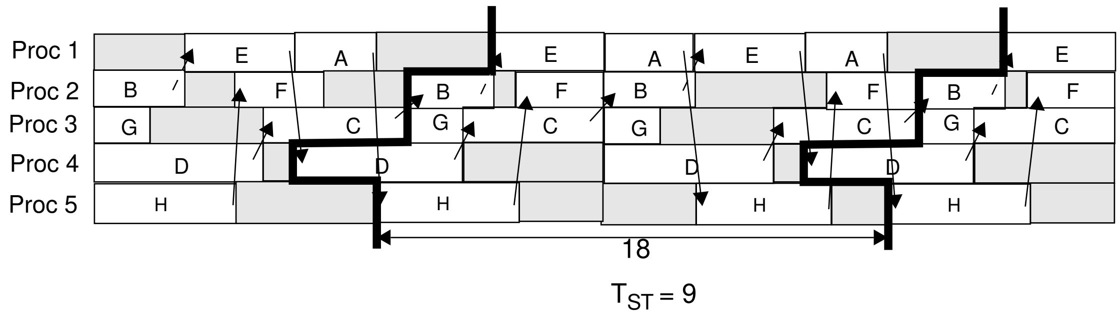
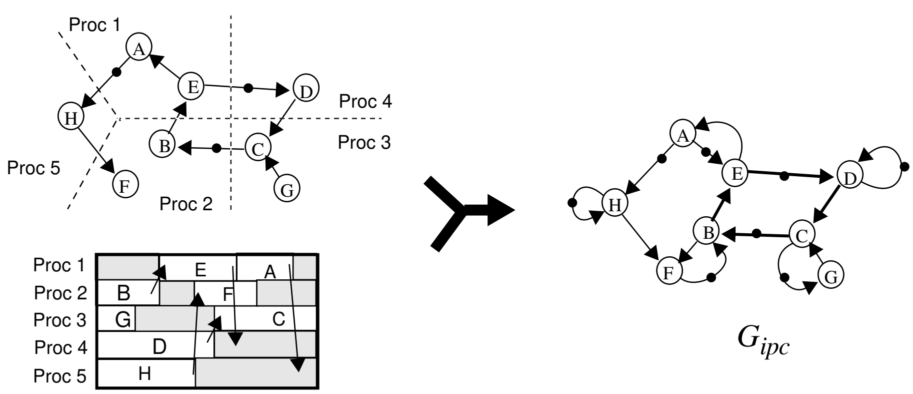

Proc 1
Proc 2
Proc 3
Proc 4
Proc 5
E
A
E
A
E
A
E
B
F
B
F
B
F
B
F
G
C
G
C
G
C
G
C
D
D
D
D
H
H
H
H
18
T_ST = 9

Figure 2


Proc 1
Proc 4
Proc 3
Proc 5
Proc 2
A
E
D
H
B
C
F
G
Proc 1
Proc 2
Proc 3
Proc 4
Proc 5
E
A
B
F
G
C
D
H
A
E
D
H
B
C
F
G
G_ipc

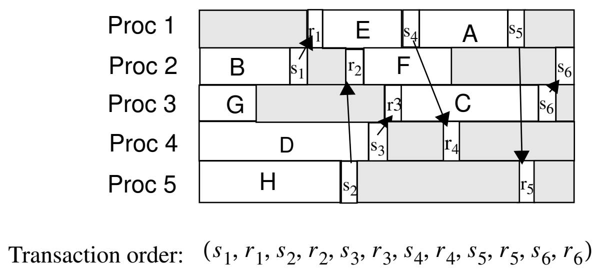


Figure 4

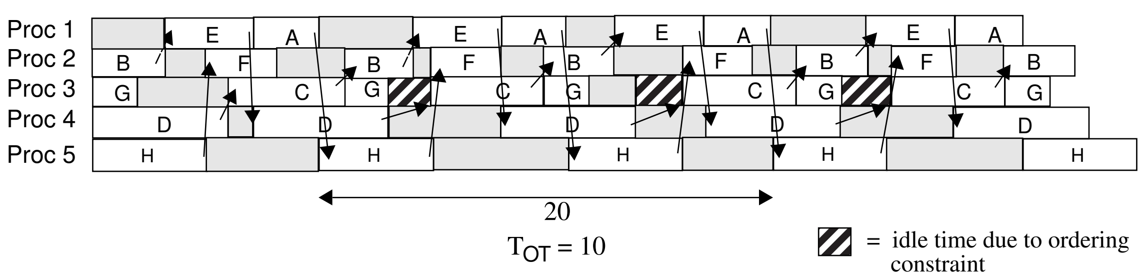


Figure 5

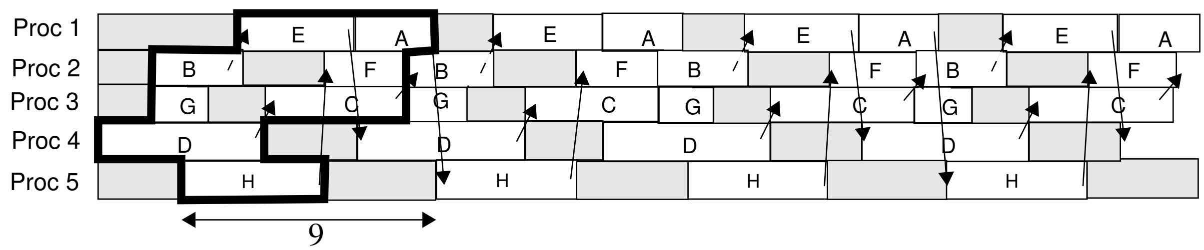


Figure 6