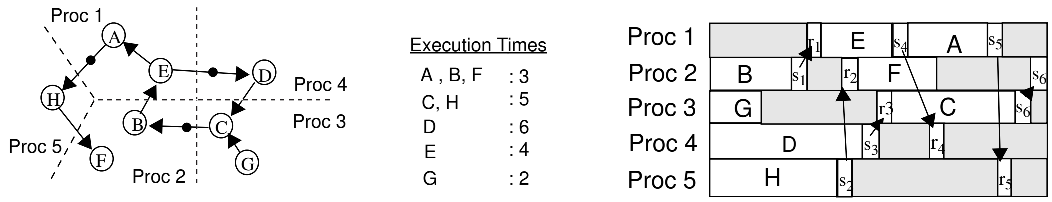
4

AN ANALYSIS OF THE OT STRATEGY

In this chapter we systematically analyze the limits of the OT scheduling strategy. Recall that the ST schedule is obtained by first generating a fully-static (FS) schedule $\{\sigma_p(v), \sigma_t(v), T_{FS}\}$, and then ignoring the exact firing times specified by the FS schedule; the FS schedule itself is derived using compile time estimates of actor execution times of actors. The OT strategy is essentially the self-timed strategy with the added ordering constraints O that force processors to communicate in an order predetermined at compile time. The questions we try to address in this chapter are: What exactly are we sacrificing by imposing such a restriction? Is it possible to choose a transaction such that this penalty is minimized? What is the effect of variations of task (actor) execution times on the throughput achieved by a self-timed strategy and by an OT strategy?

The effect of imposing a transaction order on a self-timed schedule is best illustrated by the following example. Let us assume that we use the dataflow graph and its schedule that was introduced in Chapter 1 (Fig. 1.2), and that we enforce the transaction order of Fig. 3.1; we reproduce these for convenience in Fig. 4.1 (a) and (b).

If we observe how the scheduled "evolves" as it is executed in a self-timed manner (essentially a simulation in time of when each processor executes actors as-



Transaction order: $\left(s_1, r_1, s_2, r_2, s_3, r_3, s_4, r_4, s_5, r_5, s_6, r_6\right)$

(a) HSDFG “G”

(b) Schedule and Transaction order

Figure 4.1. Fully-static schedule on five processors

signed to it), we get the “unfolded” schedule of Fig. 4.2; successive iterations of the HSDFG overlap in a natural manner. This is of course an idealized scenario where IPC costs are ignored; we do so to avoid unnecessary detail in the diagram, since IPC costs can be included in our analysis in a straightforward manner. Note that the ST schedule in Fig. 4.2 eventually settles to a periodic pattern consisting of two iterations of the HSDFG; the average iteration period under the self-timed schedule is 9 units. The average iteration period (which we will refer to as T_{ST}) for such an idealized (zero IPC cost) 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 compile time determined order. An actor at the head of this ordered list is executed as soon as data is available for it. 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. In particular, the overlap of successive iterations of the HSDFG in the idealized ST schedule ensures that $T_{ST} \le T_{FS}$ in general.

The ST schedule allows reordering among IPCs at run time. In fact we observe from Fig. 4.2 that once the ST schedule settles into a periodic pattern, IPCs in

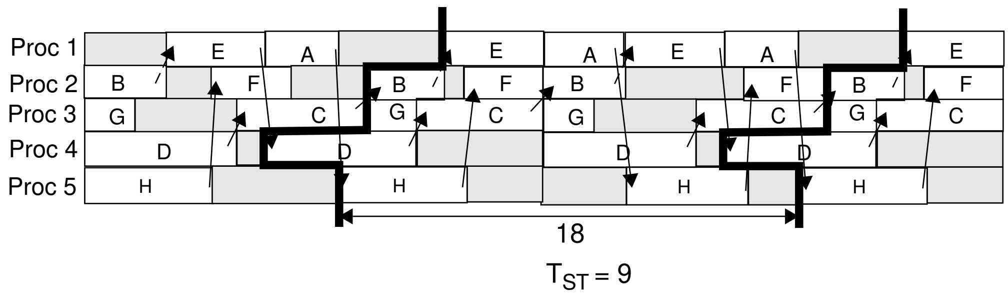


Figure 4.2. Self-timed schedule

successive iterations are ordered differently: in the first iteration the order in which IPCs occur is indeed $(s_1, r_1, s_2, r_2, s_3, r_3, s_4, r_4, s_5, r_5, s_6, r_6)$; 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 impose the transaction order in Fig. 3.1 that enforces the order $(s_1, r_1, s_2, r_2, s_3, r_3, s_4, r_4, s_5, r_5, s_6, r_6)$, the resulting OT schedule evolves as shown in Fig. 4.3. Notice that enforcing this schedule introduces idle time (hatched rectangles); as a result, T_{OT}, the average iteration period for the OT schedule, is 10 units, which is (as expected) larger than the iteration period of the ideal ST schedule T_{ST} (9 units) but is smaller than T_{FS} (11 units). In general

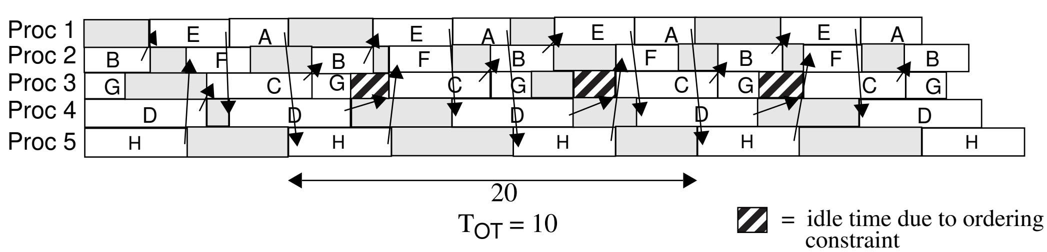


Figure 4.3. Schedule evolution when the transaction order of Fig. 3.1 is enforced

$T_{FS} \geq T_{OT} \geq T_{ST}$: the ST 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 question we would like to answer is: is it possible to choose the transaction ordering more intelligently than the straightforward one (obtained by sorting) chosen in Fig. 3.1?

As a first step towards determining how such a "best" possible access order might be obtained, we attempt to model the self-timed execution itself and try to determine the precise effect (e.g. increase in the iteration period) of adding transaction ordering constraints. Note again that as the schedule evolves in a self-timed manner in Fig. 4.2, it eventually settles into a periodic repeating pattern that spans two iterations of the dataflow graph, and the average iteration period, T_{ST}, is 9. We would like to determine these properties of self-timed schedules analytically.

4.1 Inter-processor Communication graph (G_{ipc})

In a self-timed strategy a schedule S specifies the actors assigned to each processor, including the IPC actors *send* and *receive*, and specifies the order in which these actors must be executed. At run time each processor executes the actors assigned to it in the prescribed order. When a processor executes a send it writes into a certain buffer of finite size, and when it executes a receive, it reads from a corresponding buffer, and it checks for buffer overflow (on a send) and buffer underflow (on a receive) before it performs communication operations; it blocks, or suspends execution, when it detects one of these conditions.

We model a self-timed schedule using an HSDFG $G_{ipc} = (V, E_{ipc})$ derived from the original SDF graph $G = (V, E)$ and the given self-timed schedule. The graph G_{ipc}, which we will refer to as the **inter-processor communication modelling graph**, or **IPC graph** for short, models the fact that actors of G assigned to the same processor execute sequentially, and it models constraints due to inter-processor communication. For example, the self-timed schedule in Fig. 4.1 (b) can be modelled by the IPC graph in Fig. 4.4.

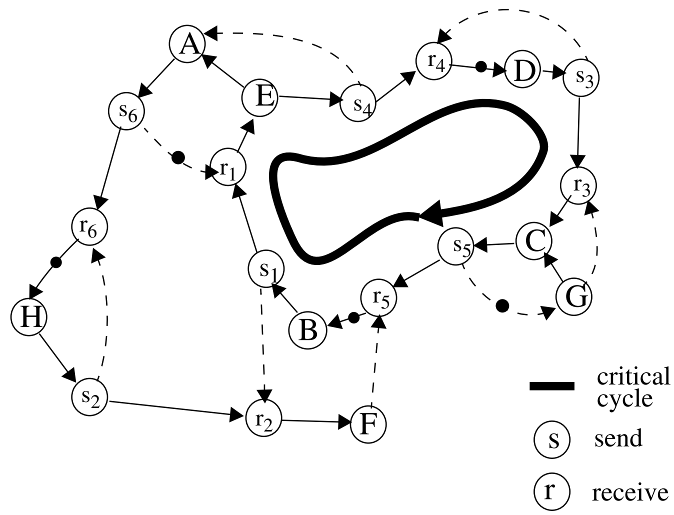


Figure 4.4. The IPC graph for the schedule in Fig. 4.1.

The IPC graph has the same vertex set V as G, corresponding to the set of actors in G. The self-timed schedule specifies the actors assigned to each processor, and the order in which they execute. For example in Fig. 4.1, processor 1 executes A and then E repeatedly. We model this in G_{ipc} by drawing a cycle around the vertices corresponding to A and E, and placing a delay on the edge from E to A. The delay-free edge from A to E represents the fact that the kth execution of A precedes the kth execution of E, and the edge from E to A with a delay represents the fact that the kth execution of A can occur only after the $(k-1)$th execution of E 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 have a cycle $((v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n), (v_n, v_1))$, with $delay((v_n, v_1)) = 1$ (because v_1 is executed first). If there are P processors in the schedule, then we have P such cycles corresponding to each processor. The additional edges due to these constraints are shown dashed in Fig. 4.4.

As mentioned before, edges in G that cross processor boundaries after scheduling represent inter-processor communication. Communication actors (*send* and *receive*) are inserted for each such edge; these are shown in Fig. 4.1.

The IPC graph has the same semantics as an HSDFG, and its execution models the execution of the corresponding self-timed schedule. The following definitions are useful to formally state the constraints represented by the IPC graph. Time is modelled as an integer that can be viewed as a multiple of a base clock.

Recall that the function $start(v, k) \in Z^+$ represents the time at which the kth execution of actor v starts in the self-timed schedule. The function $end(v, k) \in Z^+$ represents the time at which the kth execution of the actor v ends and v produces data tokens at its output edges, and we set $start(v, k) = 0$ and $end(v, k) = 0$ for $k < 0$ as the "initial conditions". The $start(v, 0)$ values are specified by the schedule: $start(v, 0) = \sigma_t(v)$.

Recall from Definition 2.2, as per the semantics of an HSDFG, each edge (v_j, v_i) of G_{ipc} represents the following data dependence constraint:

$$start(v_i, k) \geq end(v_j, k - delay((v_j, v_i))),$$
$$\forall (v_j, v_i) \in E_{ipc}, \forall k \geq delay(v_j, v_i) \tag{4-1}$$

The constraints in (Eqn. 4-1) are due both to communication edges (representing synchronization between processors) and to edges that represent sequential execution of actors assigned to the same processor.

Also, to model execution times of actors we associate execution time $t(v)$ with each vertex of the IPC graph; $t(v)$ assigns a positive integer execution time to each actor v (which can be interpreted as $t(v)$ cycles of a base clock). Inter-processor communication costs can be represented by assigning execution times to the *send* and *receive* actors. Now, we can substitute $end(v_j, k) = start(v_j, k) + t(v_j)$ in (Eqn. 4-1) to obtain

$$start(v_i, k) \geq start(v_j, k - delay((v_j, v_i))) + t(v_j) \quad \forall (v_j, v_i) \in E_{ipc} \qquad (4\text{-}2)$$

In the self-timed schedule, actors fire as soon as data is available at all their input edges. Such an "as soon as possible" (ASAP) firing pattern implies:

$$start(v_i, k) = max\left(\{ start(v_j, k - delay((v_j, v_i))) + t(v_j) \mid (v_j, v_i) \in E_{ipc} \} \right)$$

In contrast, recall that in the FS schedule we would force actors to fire periodically according to $start(v, k) = \sigma_t(v) + kT_{FS}$.

The IPC graph has the same semantics as a Timed Marked graph in Petri net theory [Peter81][Ram80] — the *transitions* of a marked graph correspond to the nodes of the IPC graph, the *places* of a marked graph correspond to edges, and the *initial marking* of a marked graph corresponds to initial tokens on the edges. The IPC graph is also similar to Reiter's computation graph [Reit68]. The same properties hold for it, and we state some of the relevant properties here. The proofs here are similar to the proofs for the corresponding properties in marked graphs and computation graphs in the references above.

Lemma 4.1: The number of tokens in any cycle of the IPC graph is always conserved over all possible valid firings of actors in the graph, and is equal to the path delay of that cycle.

Proof: For each cycle C in the IPC graph, the number of tokens on C can only change when actors that are on it fire, because actors not on C remove and place tokens only on edges that are not part of C. If $C = ((v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n), (v_n, v_1))$, and any actor v_k $(1 \leq k \leq n)$ fires, then exactly one token is moved from the edge (v_{k-1}, v_k) to the edge (v_k, v_{k+1}), where $v_0 \equiv v_n$ and $v_{n+1} \equiv v_1$. This conserves the total number of tokens on C. *QED.*

Definition 4.1: An HSDFG G is said to be **deadlocked** if at least one of its actors cannot fire an infinite number of times in any valid sequence of firings of actors in G. Thus in a deadlocked HSDFG, some actor v fires $k < \infty$ number of times, and is never enabled to fire subsequently.

Lemma 4.2: An HSDFG G (in particular, an IPC graph) is free of deadlock if and only if it does not contain delay free cycles.

Proof: Suppose there is a delay free cycle $C = ((v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n), (v_n, v_1))$ in G (i.e. $Delay(C) = 0$). By Lemma 4.1 none of the edges $(v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n), (v_n, v_1)$, can contain tokens during any valid execution of G. Then each of the actors $v_1, \ldots, v_n$ has at least one input that never contains any data. Thus none of the actors on C are ever enabled to fire, and hence G is deadlocked.

Conversely, suppose G is deadlocked, i.e. there is one actor v_1 that never fires after a certain sequence of firings of actors in G. Thus, after this sequence of firings, there must be an input edge (v_2, v_1) that never contains data. This implies that the actor v_2 in turn never gets enabled to fire, which in turn implies that there must be an edge (v_3, v_2) that never contains data. In this manner we can trace a path $p = ((v_n, v_{n-1}), \ldots, (v_3, v_2), (v_2, v_1))$ for $n = |V|$ back from v_1 to v_n that never contains data on its edges after a certain sequence of firing of actors in G. Since G contains only $|V|$ actors, p must visit some actor twice, and hence must contain a cycle C. Since the edges of p do not contain data, C is a delay free cycle. *QED*.

Definition 4.2: A schedule S is said to be **deadlocked** if after a certain finite time at least one processor blocks (on a buffer full or buffer empty condition) and stays blocked.

If the specified schedule is deadlock free then the corresponding IPC graph is deadlock free. This is because a deadlocked IPC graph would imply that a set of processors depend on data from one another in a cyclic manner, which in turn implies a schedule that displays deadlock.

Lemma 4.3: The asymptotic iteration period for a *strongly connected* IPC graph G when actors execute as soon as data is available at all inputs is given by:

$$T = \max_{\text{cycle } C \text{ in } G} \left\{ \frac{\sum_{v \text{ is on } C} t(v)}{Delay(C)} \right\} \tag{4-3}$$

Note that $Delay(C) > 0$ for an IPC graph constructed from an admissible schedule. This result has been proved in so many different contexts ([Kung87a][Peter81][Ram80][Reit68][Renf81]) that we avoid presenting another proof of this fact here.

The quotient in Eqn. 4-3 is called the **cycle mean** of the cycle C. The entire quantity on the right hand side of Eqn. 4-3 is called the "maximum cycle mean" of the strongly connected IPC graph G. If the IPC graph contains more than one SCC, then different SCCs may have different asymptotic iteration periods, depending on their individual maximum cycle means. In such a case, the iteration period of the overall graph (and hence the self-timed schedule) is the *maximum* over the maximum cycle means of all the SCCs of G_{ipc}, because the execution of the schedule is constrained by the slowest component in the system. Henceforth, we will define the maximum cycle mean as follows.

Definition 4.3: The **maximum cycle mean** of an IPC graph G_{ipc}, denoted by $MCM(G_{ipc})$, is the maximal cycle mean over all strongly connected components of G_{ipc}: That is,

$$MCM(G_{ipc}) = \max_{\text{cycle } C \text{ in } G_{ipc}} \left\{ \frac{\sum_{v \text{ is on } C} t(v)}{Delay(C)} \right\}.$$

Note that $MCM(G)$ may be a non-integer rational quantity. We will use the term MCM instead of $MCM(G)$ when the graph being referred to is clear from the context. A fundamental cycle in G_{ipc} whose cycle mean is equal to MCM is called a **critical cycle** of G_{ipc}. Thus the throughput of the system of processors executing a particular self-timed schedule is equal to the corresponding $\frac{1}{MCM}$ value.

For example, in Figure 4.4, G_{ipc} has one SCC, and its maximal cycle mean is 7 time units. This corresponds to the critical cycle $((B, E), (E, I), (I, G), (G, B))$. We have not included IPC costs in this calculation, but these can be included in a straightforward manner by appropriately setting the execution times of the *send* and *receive* actors.

The maximum cycle mean can be calculated in time $O(|V||E_{ipc}|\log_2(|V| + D + T))$, where D and T are such that $delay(e) \le D$ $\forall e \in E_{ipc}$ and $t(v) \le T$ $\forall v \in V$ [Law76].

4.2 Execution time estimates

If we only have execution time estimates available instead of exact values, and we set $t(v)$ in the previous section to be these estimated values, then we obtain the *estimated* iteration period by calculating MCM. Henceforth we will assume that we know the **estimated throughput** $\frac{1}{MCM}$ calculated by setting the $t(v)$ values to the available timing estimates. As mentioned in Chapter 1, for most practical scenarios, we can only assume such compile time estimates, rather than clock-cycle accurate execution time estimates. In fact this is the reason we had to rely on self-timed scheduling, and we proposed the ordered transaction strategy as a means of achieving efficient IPC despite the fact that we do not assume knowledge of exact actor execution times.

4.3 Ordering constraints viewed as edges added to G_{ipc}

The ordering constraints can be viewed as edges added to the IPC graph: an edge (v_j, v_i) with zero delays represents the constraint $start(v_i, k) \geq end(v_j, k)$. The ordering constraints can therefore be expressed as a set of edges between communication actors. For example, the constraints

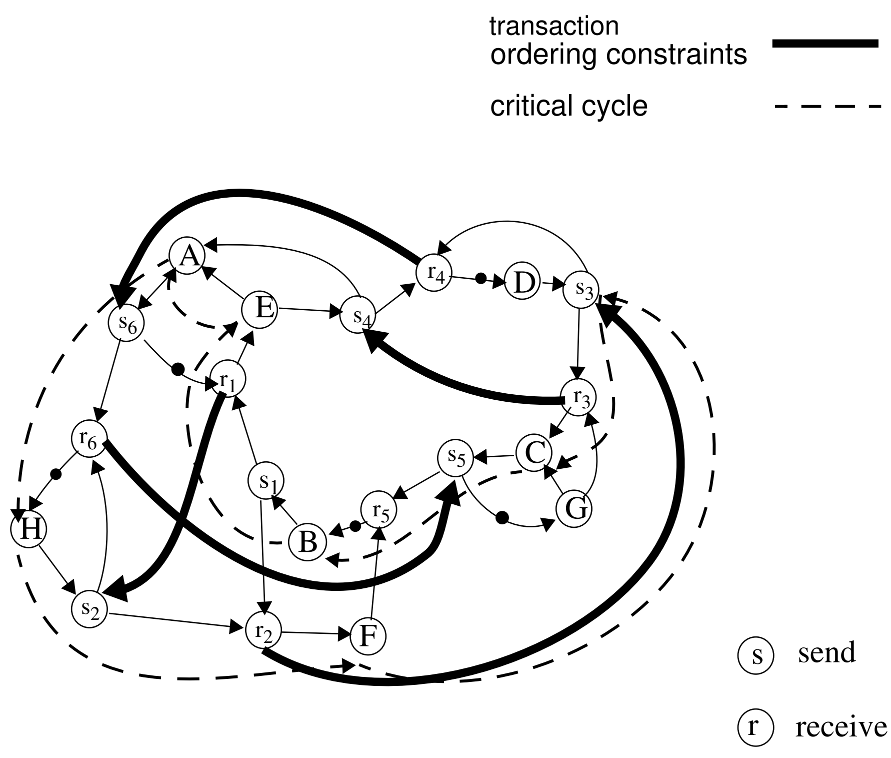


Figure 4.5. Transaction ordering constraints

$O = (s_1, r_1, s_2, r_2, s_3, r_3, s_4, r_4, s_5, r_5, s_6, r_6)$ applied to the IPC graph of Fig. 4.4

is represented by the graph in Fig. 4.5. If we call these additional ordering constraint edges E_{OT} (solid arrows in Fig. 4.5), then the graph $(V, E_{ipc} \cup E_{OT})$ represents constraints in the OT schedule, as it evolves in Fig. 4.3. Thus the maximum cycle mean of $(V, E_{ipc} \cup E_{OT})$ represents the effect of adding the ordering constraints. The critical cycle C of this graph is drawn in Fig. 4.5; it is different from the critical cycle in Fig. 4.4 because of the added transaction ordering constraints. Ignoring communication costs, the MCM is 9 units, which was also observed from the evolution of the transaction constrained schedule in Fig. 4.3.

The problem of finding an "optimal" transaction order can therefore be stated as: Determine a transaction order O such that the resultant constraint edges E_{OT} do not increase the MCM, i.e.

$$MCM((V, E_{ipc})) = MCM((V, E_{ipc} \cup E_{OT})) .$$

4.4 Periodicity

We noted earlier that as the ST schedule in Fig. 4.2 evolves, it eventually settles into a periodic repeating pattern that spans two iterations of the dataflow graph. It can be shown that a ST schedule always settles down into a periodic execution pattern; in [Bacc92] the authors show that the firing times of transitions in a marked graph are periodic asymptotically. Interpreted in our notation, for any strongly connected HSDFG:

$$\exists K, N \text{ s.t.}$$

$$start(v_i, k+N) = start(v_i, k) + MCM(G_{ipc}) \times N \qquad \forall v_i \in V, \forall k > K$$

Thus after a "transient" that lasts K iterations, the ST schedule evolves into a periodic pattern. The periodic pattern itself spans N iterations; we call N the **periodicity**. The periodicity depends on the number of delays in the critical cycles of G_{ipc}; it can be as high as the number least common multiple of the number of delays in the critical cycles of G_{ipc} [Bacc92]. For example, the IPC graph of Fig. 4.4 has one critical cycle with two delays on it, and thus we see a periodicity of two for the schedule in Fig. 4.2. The "transient" region defined by K (which is 1 in Fig. 4.2) can also be exponential.

The effect of transients followed by a periodic regime is essentially due to properties of longest paths in weighted directed graphs. These effects have been studied in the context of instruction scheduling for VLIW processors [Aik88][Zaky89], as-soon-as-possible firing of transitions in Petri nets [Chre83], and determining clock schedules for sequential logic circuits [Shen92]. In [Aik88] the authors note that if instructions in an iterative program for a VLIW processor (represented as a dependency graph) are scheduled in an as-soon-as-possible fashion, a pattern of parallel instructions "emerges" after an initial transient, and the authors show how determining this pattern (essentially by simulation) leads to efficient loop parallelization. In [Zaky89], the authors propose a max-algebra [Cun79] based technique for determining the "steady state" pattern in the VLIW program. In [Chre83] the author studies periodic firing patterns of transitions in timed Petri nets. The iterative algorithms for determining clock schedules in [Shen92] have convergence properties similar to the transients in self-timed schedules (their algorithm converges when an equivalent self-timed schedule reaches a periodic regime).

Returning to the problem of determining the optimal transaction order, one possible scheme 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. 3.1, if we enforce the transaction order that repeats over two iterations in the evolution of the ST schedule of Fig. 4.2, 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. However, as pointed out above, the number of iterations that the repeating pattern spans depends on the critical cycles of G_{ipc}, and it can be exponential in the size of the HSDFG [Bacc92].

In addition the "transient" region before the schedule settles into a repeating pattern can also be exponential. Consequently the memory requirements for the controller that enforces the transaction order can be prohibitively large in certain cases; in fact, even for the example of Fig. 4.2, the doubling of the controller memory that such a strategy entails may be unacceptable. We therefore restrict ourselves to determining and enforcing a transaction order that spans only one iteration of the HSDFG; in the following section we show that there is no sacrifice in imposing such a restriction and we discuss how such an "optimal" transaction order is obtained.

4.5 Optimal order

In this section we show how to determine an order O^* on the IPCs in the schedule such that 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.

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 4.1: 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:
$\sigma'_t(v_j) - \sigma'_t(v_i) \le \lceil T_{ST} \rceil \times d(v_j, v_i) - t(v_j)$ for each edge (v_j, v_i) 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:

$$start(v_i, k) = \sigma'_t(v_i) + kT \tag{4-4}$$

Also, data precedence constraints imply (as in Eqn. 4-2):

$$start(v_i, k) \ge start(v_j, k - delay(v_j, v_i)) + t(v_j) \quad \forall (v_j, v_i) \in E_{ipc} \tag{4-5}$$

Substituting Eqn. 4-4 in Eqn. 4-5:

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

That is:

$$\sigma'_t(v_j) - \sigma'_t(v_i) \le T \times d(v_j, v_i) - t(v_j) \quad \forall (v_j, v_i) \in E_{ipc} \tag{4-6}$$

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 Eqn. 4-6 represent a system of $|E_{ipc}|$ inequalities in $|V|$ unknowns (the quantities $\sigma'_t(v_i)$).

The system of inequalities in Eqn. 4-6 is a difference constraint problem that can be solved in polynomial time ($O(|E_{ipc}||V|)$) using the Bellman-Ford shortest-path algorithm [Law76][Corm92]. The details of this approach are well described in [Corm92]; 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 [Corm92], which is equivalent to the following condition:

$$T \geq \max_{\text{cycle C in } G_{ipc}} \left\{ \frac{\sum_{v \in C} t(v)}{D(C)} \right\}$$

; and, from Eqn. 4-3, 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 4-6 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. *QED.*

Remark: Claim 4.1 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. Recall that a "time unit" is essentially the clock period; therefore, one time unit can usually be neglected.

For example the static schedule S corresponding to Fig. 4.1 has $T_{FS} = 11 > T_{ST} = 9$ units. Using the procedure outlined in Claim , we can skew the starting times of processors in the schedule S to obtain a schedule S', as shown in (4-5), that has a period equal to 9 units (Fig. 4.6). Note that the processor assignment and actor ordering in the schedule of Fig. 4.6 is identical to that of the schedule in Fig. 4.1. The values $\sigma'_t(v)$ are: $\sigma'_t(A) = 9$, $\sigma'_t(B) = \sigma'_t(G) = 2$,

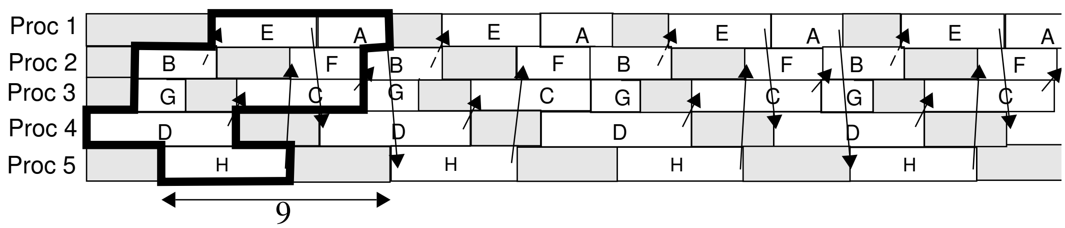


Figure 4.6. Modified schedule S'

$\sigma'_t(C) = 6$, $\sigma'_t(D) = 0$, $\sigma'_t(E) = 5$, $\sigma'_t(F) = 8$, and $\sigma'_t(H) = 3$.

Claim 4.1 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 4.1? Recall 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) [Sih91] or are based on overlapped scheduling techniques that employ list scheduling heuristics [deGroot92][Lam88]]. 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 4.1. 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 [Schw85] 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. 3.1). 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. 4.6 results in T_{OT} of 9 units instead of 10 that we get if the transaction order of Fig. 3.1 is used. Under the transaction order specified by S', $T_{ST} \le T_{OT} \le \lceil T_{ST} \rceil$; 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. 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.

4.6 Effects of changes in execution times

We recall that the execution times we use to determine the actor assignment and ordering in a self-timed schedule are compile time estimates, and we have been stating that static scheduling is advantageous when we have "reasonably good" compile time estimates of execution time of actors. Also, intuitively we expect an ordered transaction schedule to be more sensitive to changes in execution times than an unconstrained ST schedule. In this section we attempt to formalize these notions by exploring the effect of changes in execution times of actors on the throughput achieved by a static schedule.

Compile time estimates of actor execution times may be different from their actual values at run time due to errors in estimating execution times of actors that otherwise have fixed execution times, and due to actors that display run time variations in their execution times, because of conditionals or data-dependent loops within them for example. The first case is simple to model, and we will show in section 4.6.1 how the throughput of a given self-timed schedule changes as a function of actor execution times. The second case is inherently difficult; how do we model run time changes in execution times due to data-dependencies, or due to events such as error handling, cache misses, and pipeline effects? In section 4.6.2 below we briefly discuss a very simple model for such run time variations; we assume actors have random execution times according to some known probability distribution. We conclude that analysis of even such a simple model for the expected value of the throughput is often intractable, and we discuss efficiently computable upper and lower bounds for the expected throughput.

4.6.1 Deterministic case

Consider the IPC graph in Fig. 4.7, which is the same IPC graph as in Fig. 4.4 except that we have used a different execution time for actor H to make the example more illustrative. The numbers next to each actor represents execution times of the actors. We let the execution time of actor C be $t(C) = t_C$, and we determine the iteration period as a function of given a particular value of t_C ($T_{ST}(t_C)$). The

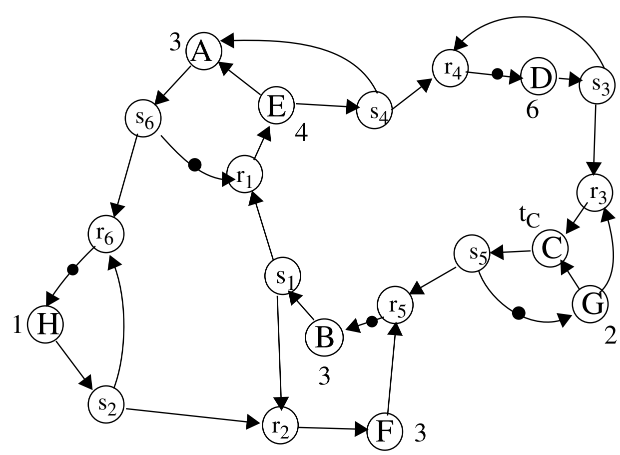


Figure 4.7. G_{ipc}, actor C has execution time t_c, constant over all invocations of C

iteration period is given by $MCM(G_{ipc})$, the maximum cycle mean. The function $T_{ST}(t_C)$ is shown in Fig. 4.8. When $0 \le t_C \le 1$, the cycle

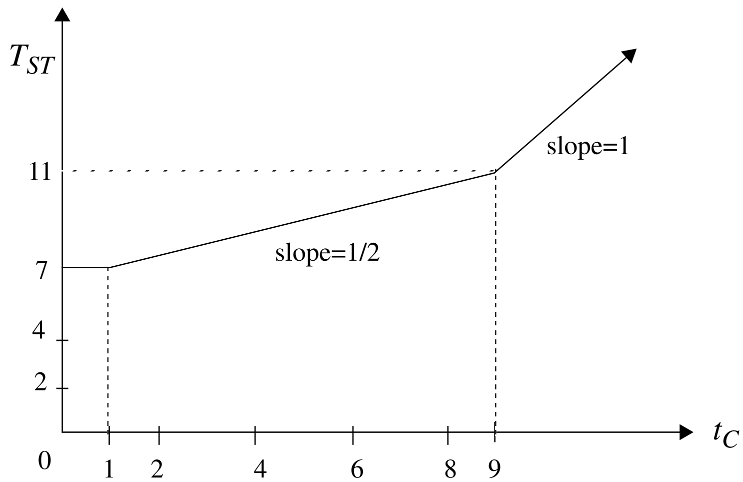


Figure 4.8. $T_{ST}(t_C)$

$((A, s_6)\ (s_6, r_1)\ (r_1, E)\ (E, A))$ is critical, and the MCM is constant at 7; when $1 \le t_C \le 9$, the cycle

$((B, s_1)\ (s_1, r_1)\ (r_1, E)\ (E, s_4)\ (s_4, r_4)\ (r_4, D)\ (D, s_3)\ (s_3, r_3)\ (r_3, C)\ (C, s_5)$
$(s_5, r_5)\ (r_5, B))$

is critical, and since this cycle has two delays, the slope of $T_{ST}(t_C)$ is half in this region; finally, when $9 \le t_C$ the cycle $((C, s_5)\ (s_5, G)\ (G, C))$ becomes critical, and the slope now is one because on that cycle.

Thus the effect of changes in execution times of each actor is piecewise linear, and the slope depends on the number of delays on the critical cycle that the actor lies on. The slope is at most one (when the critical cycle containing the particular actor has a single delay on it). The iteration period is a **convex function** of actor execution times.

Definition 4.4: A function $f(x)$ is said to be convex over an interval (a, b) if for every $x_1, x_2 \in (a, b)$ and $0 \le \lambda \le 1$,

$f(\lambda x_1 + (1 - \lambda) x_2) \le \lambda f(x_1) + (1 - \lambda) f(x_2)$.

Geometrically, if we plot a convex function $f(x)$ along x, a line drawn between two points on the curve lies above the curve (but it may overlap sections of the curve).

It is easily verified geometrically that $T_{ST}(t_C)$ is convex: since this function is piecewise linear with a slope that is positive and non-decreasing, a line joining two points on it must lie above (but may coincide with) the curve.

We can also plot T_{ST} as a function of execution times of more than one actor (e.g. $T_{ST}(t_A, t_B, \ldots)$); this function will be a convex surface consisting of intersecting planes. Slices of this surface along each variable look like Fig. 4.8, which is a slice parallel to the t_C axis, with the other execution times held constant ($t_A = 3$, $t_B = 3$, etc.).

The modelling described in this section is useful for determining how "sensitive" the iteration period is to fixed changes in execution times of actors, given a processor assignment and actor ordering. We observe that the iteration period increases linearly (with slope one) at worst, and does not change at all at best, when execution time of an actor is increased beyond its compile time estimate.

4.6.2 Modeling run time variations in execution times

The effect of variations in execution times of actors on the performance of statically scheduled hardware is inherently difficult to quantify, because these variations could occur due to a large number of factors — conditional branches or data dependent loops within an actor, error handling, user interrupts etc. — and because these variations could have a variety of different characteristics, from being periodic, to being dependent on the input statistics, and to being completely random. As a result thus far we have had to resort to statements like "for a static scheduling strategy to be viable, actors must not show significant variations in execution times." In this section we point out the issues involved in determining the effects of variations in execution times of actors.

A very simple model for actors with variable execution times is to assign to each actor an execution time that is a random variable (r.v.) with a discrete probability distribution (p.d.f.); successive invocations of each actor are assumed statistically independent, execution times of different actors are assumed independent, and the statistics of the random execution times are assumed to be time invariant. Thus, for example, an actor A could have execution time t_1 with probability (w.p.) p and execution time t_2 w.p. $(1-p)$. The model is essentially that A flips a coin each time it is invoked to decide what its execution time should be for that invocation. Such a model could describe a data-dependent conditional branch for example, but it is of course too simple to capture many real scenarios.

Dataflow graphs where actors have such random execution times have been studied by Olsder *et. al.* [Ols89][Ols90] in the context of modeling data-driven networks (also called wave-front arrays [Kung87a]) where the multiply operations in the array display data-dependent execution times. The authors show that the behaviour of such a system can be described by a discrete-time Markov chain. The idea behind this, briefly, is that such a system is described by a state space consisting of a set of state vectors s. Entries in each vector s represent the kth starting time of each actor normalized with respect to one (any arbitrarily chosen) actor:

$$s = \begin{pmatrix} 0 \\ start(v_2, k) - start(v_1, k) \\ start(v_3, k) - start(v_1, k) \\ \cdots \\ start(v_n, k) - start(v_1, k) \end{pmatrix}$$

The normalization (with respect to actor v_1 in the above case) is done to make the state space finite; the number of distinct values that the vector s (as defined above) can assume is shown to be finite in [Ols90]. The states of the Markov chain correspond to each of the distinct values of s. The average iteration period, which is defined as:

$$T = \lim_{K \to \infty} \frac{start(v_i, K)}{K}$$

can then be derived from the stationary distribution of the Markov chain. There are several technical issues involved in this definition of the average iteration period; how do we know the limit exists, and how do we show that the limit is in fact the same for all actors (assuming that the HSDFG is strongly connected)? These questions are fairly non-trivial because the random process $\{start(v_i, k)\}_k$ may not even be stationary. These questions are answered rigorously in [Bacc92], where it is shown that:

$$T = \lim_{K \to \infty} \frac{start(v_i, K)}{K} = E[T] \quad \forall v_i \in V.$$

Thus the limit T is in fact a constant *almost surely.*

The problem with such exact analysis, however, is the very large state space that can result. We found that for an IPC Graph similar to Fig. 4.4, with certain choices of execution times, and assuming that only t_C is random (takes two different values based on a weighted coin flip), we could get several thousand states for the Markov chain. A graph with more vertices leads to an even larger state space. The upper bound on the size of the state space is exponential in the number of vertices (exponential in $|V|$). Solving the stationary distribution for such Markov chains would require solving a set of linear equations equal in number to the number of states and is highly compute intensive. Thus we conclude that this approach has limited use in determining effects of varying execution times; even for unrealistically simple stochastic models, computation of exact solutions is prohibitive.

If we assume that all actors have exponentially distributed execution times, then the system can be analyzed using continuous-time Markov chains [Moll82]. This is done by exploiting the memoryless property of the exponential distribution: when an actor fires, the state of the system at any moment does not depend on how long that actor has spent executing its function; the state changes only when that actor completes execution. The number of states for such a system is equal to the number of different valid token configurations on the edges of the dataflow graph, where by "valid" we imply any token configuration that can be reached by a sequence of firings of enabled actors in the HSDFG. This is also equal to the number of *valid retimings* [Lei91] that exist for the HSDFG. This number, unfortunately, is again exponential in the size of the HSDFG.

Analysis of such graphs with exponentially distributed execution times has been extensively studied in the area of stochastic Petri nets (in [Mur89] Murata provides a large and comprehensive list of references on Petri nets — 315 in all — a number of which focus on stochastic Petri nets). There is a considerable body of work that attempts to cope with the state explosion problem. Some of these works attempt to divide a given Petri net into parts that can be solved separately (e.g. [Yao93]), some others propose simplified solutions when the graphs have particular structures (e.g. [Cam92]), and others propose approximate solutions for values such as the expected firing rate of transitions (e.g. [Hav91]). None of these methods are general enough to handle even a significant class of IPC graphs. Again, exponentially distributed execution times for *all* actors is clearly a crude approximation to any realistic scenario to make the computations involved in exact calculations worthwhile.

As an alternative to determining the exact value of $E[T]$ we discuss how to determine efficiently computable bounds for it.

Definition 4.5: Given an HSDFG $G = V, E$ that has actors with random execution times, define $G_{ave} = (V, E)$ to be an equivalent graph with actor execution times equal to the expected value of their execution times in G.

Fact 4.1: [Durr91] (Jensen's inequality) If $f(x)$ is a convex function of x, then: $E[f(x)] \geq f(E[x])$.

In [Rajs94] the authors use Fact 4.1 to show that $E[T] \geq MCM(G_{ave})$. This follows from the fact that $MCM(G_{ave})$ is a convex function of the execution times of each of its actors. This result is interesting because of its generality; it is true no matter what the statistics of the actor execution times are (even the various independence assumptions we made can be relaxed!).

One might wonder what the relationship between $E[T]$ and $E[MCM(G)]$ might be. We can again use Fact 4.1along with the fact that the maximum cycle mean is a convex function of actor execution times to show the following:

$$E[MCM(G)] \geq MCM[G_{ave}] .$$

However, we cannot say anything about $E[T]$ in relation to $E[MCM(G)]$; we were able to construct some graphs where $E[T] > E[MCM(G)]$, and others where $E[T] < E[MCM(G)]$.

If the execution times of actors are all bounded ($t_{min}(v) \leq t(v) \leq t_{max}(v)$ $\forall v \in V$, e.g. if all actors have execution times uniformly distributed in some interval $[a, b]$) then we can say the following:

$$\boldsymbol{MCM(G_{max}) \geq E[T] \geq MCM(G_{ave}) \geq MCM(G_{min})} \tag{4-7}$$

where $G_{max} = (V, E)$ is same as G except the random actor execution times are replaced by their upper bounds ($t_{max}(v)$), and similarly $G_{min} = (V, E)$ is the same as G except the random actor execution times are replaced by their lower bounds ($t_{min}(v)$).

Equation (4-7) summarizes the useful bounds we know for expected value of the iteration period for graphs that contain actors with random execution times. It should be noted that good upper bounds on $E[T]$ are not known. Rajsbaum and Sidi propose upper bounds for exponentially distributed execution times [Rajs94]; these upper bounds are typically more than twice the exact value of $E[T]$, and hence not very useful in practice. We attempted to simplify the Markov chain model (i.e. reduce the number of states) for the self-timed execution of a stochastic HSDFG by representing such an execution by a set of self-timed schedules of deterministic HSDFGs, between which the system makes transitions randomly. This representation reduces the number of states of the Markov chain to the number of different deterministic graphs that arise from the stochastic HSDFG. We were able to use this idea to determine a upper bound for $E[T]$; however, this bound also proved to be too loose in general (hence we omit the details of this construction here).

4.6.3 Implications for the OT schedule

Intuitively, an OT 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. This is apparent from our IPC graph model; the transaction ordering constraints add additional edges (E_{OT}) to G_{ipc}. The IPC graph with transaction ordering constraints represented as dashed arrows is shown in Fig. 4.9 (we use the transaction order $O^* = (s_1, r_1, s_3, r_3, s_2, r_2, s_4, r_4, s_6, r_6, s_5, r_5)$ determined in section 4.5 and, again, communication times are not included). The graph for $T_{OT}(t_C)$ is now different and is plotted in Fig. 4.8. Note that the $T_{OT}(t_C)$ curve for the OT schedule (solid) is "above" the corresponding curve for the unconstrained ST schedule (dashed): this shows precisely what we mean by an OT schedule being more sensitive to variations in execution times of actors. The "optimal" transaction order O^* we determined ensures that the transaction constraints do not sacrifice throughput (ensures $T_{OT} = T_{ST}$) when actor execution times are *equal to their compile time estimates;* O^* was calculated using $t_C = 3$ in section 4.5, and sure enough, $T_{OT}(t_C) = T_{ST}(t_C)$ when $t_C = 3$.

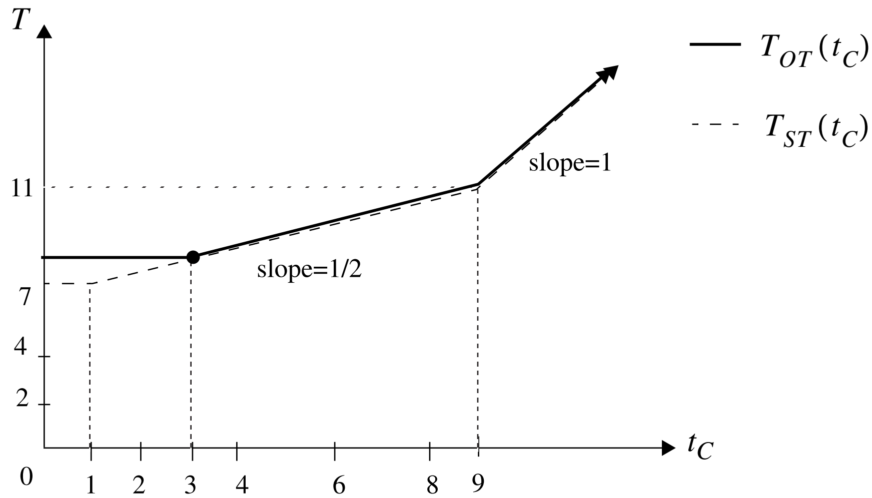


Figure 4.10. $T_{ST}(t_C)$ and $T_{OT}(t_C)$

Modeling using random variables for the OT schedule can again be done as before, and since we have more constraints in this schedule, the expected iteration period will in some cases be larger than that for an ST schedule.

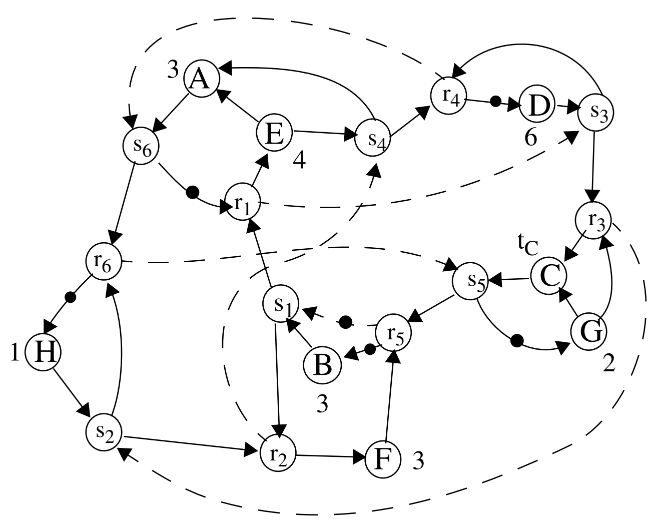


Figure 4.9. G_{ipc} with transaction ordering constraints represented as dashed lines

4.7 Summary

In this chapter we presented a quantitative analysis of ST and OT schedules and showed how to determine the effects of imposing a transaction order on an ST schedule. 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 [Lee89] to make the best use of computing resources. Under the assumption that the variations in execution times are small enough so that an ST or an OT strategy is viable, 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. This is because we can 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}. Thus we make the best possible use of compile time information when we determine the transaction order O^*.

We also presented the complexities involved in modeling run time variations in execution times of actors; even highly simplified stochastic models are difficult to analyze precisely. We pointed out bounds that have been proposed in Petri net literature for the value of the expected iteration period, and concluded that although a lower bound is available for this quantity for rather general stochastic models (using Jensen's inequality), tight upper bounds are still not known, except for the trivial upper bound using maximum execution times of actors ($MCM(G_{max})$).