Contention-Conscious Transaction Ordering in Multiprocessor DSP Systems

Mukul Khandelia, Neal K. Bambha, and Shuvra S. Bhattacharyya, *Member, IEEE*

***Abstract*—This paper explores the problem of efficiently ordering interprocessor communication (IPC) operations in statically scheduled multiprocessors for iterative dataflow graphs. In most digital signal processing (DSP) applications, the throughput of the system is significantly affected by communication costs. By explicitly modeling these costs within an effective graph-theoretic analysis framework, we show that ordered transaction schedules can significantly outperform self-timed schedules even when synchronization costs are low. However, we also show that when communication latencies are nonnegligible, finding an optimal transaction order given a static schedule is an NP-complete problem, and that this intractability holds both under iterative and noniterative execution. We develop new heuristics for finding efficient transaction orders, and perform an extensive experimental comparison to gauge the performance of these heuristics.**

***Index Terms*—Dataflow, multiprocessor, scheduling, synchronization.**

I. Background

THIS paper explores the problem of efficiently ordering interprocessor communication (IPC) operations in statically scheduled multiprocessors for iterative dataflow specifications. An iterative dataflow specification consists of a dataflow representation of the body of a loop that is to be iterated indefinitely. Dataflow programming in this form is used widely in the design and implementation of digital signal processing (DSP) systems. In this paper, we assume that we are given a dataflow specification of an application, and an associated multiprocessor schedule (e.g., derived from scheduling techniques such as those presented in [1]–[4]). Our objective is to reduce the overall IPC cost of the multiprocessor implementation, and the associated performance degradation, since IPC operations result in significant execution time and power consumption penalties, and are difficult to optimize thoroughly during the scheduling stage.

High-density, low-power, and inexpensive multiprocessor DSP solutions are in growing demand in telecom, internet routing, and IP telephony markets. Typical application areas include voice compression, echo cancellation, modem banks, and voice-over-IP. As transistor sizes shrink and processors become less expensive, it is increasingly common for embedded systems to incorporate multiple processor architectures. Multiple DSP cores can now be placed on a single chip. For example, the Texas Instruments TNETV3010 multiprocessor, targeted at voice-over-IP applications, integrates six high-performance DSP cores. It consumes 50 times less power than a general-purpose processor core, with over three times the transistor count on a die one-fifth the size. Another example is the Texas Instruments OMAP 59xx family of single-chip application processors.

This paper targets lower-cost, shared memory embedded architectures. IPC is assumed to take place through shared memory, which could be global memory between all processors, or could be distributed between pairs of processors (e.g., hardware first-in-first-out queues or dual ported memory). Such simple communication mechanisms, as opposed to cross bars and elaborate interconnection networks, are common in embedded systems, due to their simplicity and low cost.

A. Scheduling Dataflow Graphs

Our study of multiprocessor implementation strategies in this paper is in the context of *homogeneous synchronous dataflow* (HSDF) specifications. In HSDF, an application is represented as a directed graph in which vertices (*actors*) represent computational tasks of arbitrary complexity; edges (*arcs*) specify data dependencies; and the number of data values (*tokens*) produced and consumed by each actor is fixed. An actor executes or "fires" when it has enough tokens on its input arcs, and during execution, it produces tokens on its output arcs. HSDF imposes the restriction that on each invocation, each actor consumes exactly one token from each input arc, and produces one token on each output arc. HSDF and closely-related models are used extensively for multiprocessor implementation of embedded signal processing systems (e.g., see [1], [5]–[7]). We refer to an HSDF representation of an application as an *application graph*.

For multiprocessor implementation of dataflow graphs, actors in the graph need to be scheduled. Scheduling can be divided into three steps [8]—assigning actors to processors (*processor assignment*), ordering the actors assigned to each processor (*actor ordering*), and determining when each actor should commence execution. All of these tasks can either be performed at run-time or at compile time to give us different scheduling strategies. To reduce run-time overhead and improve predictability, it is often desirable in embedded applications to carry out as many of these steps possible at compile time [8].

Typically, there is limited information available at compile time since the execution times of the actors are often estimated values. These may be different from the actual execution times due to actors that display run-time variation in their execution

Manuscript received April 1, 2004; revised January 21, 2005. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Chaitali Chakrabarti.

M. Khandelia is with the Synopsys, Inc., Mountain View, CA 94043 USA (e-mail: Mukul.Khandelia@synopsys.com).

N. K. Bambha is with the Department of Electrical and Computer Engineering, University of Maryland, College Park, Alexandria, VA 22309 USA (e-mail: nbambha@eng.umd.edu).

S. S. Bhattacharyya is with the Department of Electrical and Computer Engineering, and Institute for Advanced Computer Studies University of Maryland College Park, MD 20742 USA (e-mail: ssb@eng.umd.edu).

Digital Object Identifier 10.1109/TSP.2005.861074

times because of conditionals or data-dependent loops within them, for example. However, in a number of important embedded domains, such as DSP, it is widely accepted that execution time estimates are reasonably accurate, and that good compile-time decisions can be based on them. In this paper, we focus on scheduling methods that extensively make use of execution time estimates, and perform the first two steps—processor assignment and actor ordering—at compile time.

In relation to the scheduling taxonomy of Lee and Ha [8], there are three general strategies with which we are primarily concerned in this paper. In the *fully static* (FS) strategy, all three scheduling steps are carried out at compile time, including the determination of an exact firing time for each actor. In the *self-timed* (ST) strategy, on the other hand, processor assignment and actor ordering are performed at compile time, but run-time synchronization is used to determine actor firing times: an ST schedule executes by firing each actor invocation A as soon as it can be determined via synchronization that the actor invocations on which A is dependent have all completed execution.

The FS and ST methods represent two extremes in the class of scheduling algorithms considered in this paper. The ST method is the least constrained scheme since the only constraints are the IPC dependencies, and it is tolerant of variations in execution times, while the FS strategy only works when tight worst case execution times are available, and forces system performance to conform to the available worst case bounds. When we ignore IPC costs, the ST schedule consequently gives us a lower bound on the average iteration period of the schedule since it executes in an ASAP (as soon as possible) manner.

The *ordered transaction* (OT) method [7], [9] falls in-between these two strategies. It is similar to the ST method but also adds the constraint that a linear ordering of the communication actors is determined at compile time, and enforced at run-time. The linear ordering imposed is called the *transaction order* of the associated multiprocessor implementation.

The FS and OT strategies have significantly lower overall IPC cost since all of the sequencing decisions associated with communication are made at compile time. The ST method, on the other hand, requires more IPC cost since it requires synchronization checks to guarantee the fidelity of each communication operation—that is, to guarantee that buffer underflow and overflow are consistently avoided. Significant compile-time analysis can be performed to streamline this synchronization functionality [10], [11].

The metric of interest to us in this paper is the *average iteration period* T. Intuitively, in an iterative execution of a dataflow graph, the iteration period is the number of cycles it takes for each of the actors in a schedule to execute exactly once—i.e., to complete a single graph iteration. Note that it is not necessary in a self-timed schedule for the iteration period to be the same from one graph iteration to the next, even when actor execution times are fixed [12]. The inverse of the average iteration period T gives us the throughput T^{-1}, which is the average number of graph iterations carried out per unit time.

B. Terminology and Notation

We denote the set of positive integers by $\mathbb{Z}^+$, the set of natural numbers $\{0, 1, 2, \ldots\}$ by $\mathbb{N}$, and the number of elements in a finite set S by $|S|$. With each actor $\nu \in V$ in an HSDF specification (V, E), we associate an integer $\mathrm{exec}(\nu)$, which denotes the execution time estimate of ν, and an integer $\mathrm{proc}(\nu)$, which denotes the processor to which ν is assigned in the assignment step. Each edge $(\nu_i, \nu_j) \in E$ has a nonnegative integer *delay* associated with it, which is denoted by $\mathrm{delay}(\nu_i, \nu_j)$. These delays represent initial tokens, and specify dependencies between iterations of actors in iterative execution. For example, if the tokens produced by an actor ν_i on its kth invocation are consumed by actor ν_j on its $(k+2)$th invocation, the edge between ν_i and ν_j would have a delay of 2.

Every edge (ν_i, ν_j) induces the precedence constraint

$$\mathrm{start}(\nu_j, k) \geq \mathrm{start}(\nu_i, k - \mathrm{delay}(\nu_i, \nu_j)) + \mathrm{exec}(\nu_i) \quad (1)$$

where $\mathrm{start}(x, k) \in \mathbb{Z}^+$ denotes the starting time of the kth invocation of an actor x. Here, $\mathrm{start}(\nu_i)$ is set to 0 for $k \leq 0$ as an initial condition.

A *path* in a directed graph (V, E) is a finite sequence $(e_1, e_2, \ldots, e_n)$, where each e_i is in E, and $\mathrm{snk}(e_i) = \mathrm{src}(e_{i+1})$, for $i = 1, 2, \ldots, (n-1)$. We say that the path $(e_1, e_2, \ldots, e_n)$ is *directed from* $\mathrm{src}(e_1)$ to $\mathrm{snk}(e_n)$. A path that is directed from some vertex to itself is called a *cycle*. Given a path $p = (e_1, e_2, \ldots, e_n)$, the *path delay* of p, denoted $\mathrm{delay}(p)$, is given by

$$\mathrm{delay}(p) = \sum_{i=1}^{n} \mathrm{delay}(e_i). \quad (2)$$

Each cycle c in a dataflow graph must satisfy $\mathrm{delay}(c) > 0$ to avoid deadlock.

The evolution of a self-timed implementation can be modeled by Sriram's *IPC graph model* [12]. Given an application graph and an associated self-timed schedule, the IPC graph, denoted G_{IPC}, is constructed by instantiating a vertex for each application graph actor, connecting an edge from each actor to the actor that succeeds it on the same processor, and adding an edge that has unit delay from the last actor on each processor to the first actor on the same processor. Also, for each application graph edge (x, y) that connects actors that execute on different processors, an *inter-processor* edge is instantiated in G_{IPC} from x to y. A sample application graph and a self-timed schedule are illustrated in Fig. 1, and the corresponding IPC graph is illustrated in Fig. 2.

IPC costs (estimated transmission latencies through the multiprocessor network) can be incorporated into the IPC graph model by explicitly including *communication* (*send* and *receive*) actors, and setting the execution times of these actors to equal the associated IPC costs.

The IPC graph is an instance of Reiter's *computation graph* model [13], also known as the *timed marked graph* model in Petri net theory [14], and from the theory of such graphs, it is well known that in the ideal case of unlimited bus bandwidth, the average iteration period for the ASAP execution of an IPC graph is given by the *maximum cycle mean (MCM)* of G_{IPC}, which is defined by

$$\mathrm{MCM}(G_{\mathrm{IPC}}) = \max_{\text{cycle C in } G_{\mathrm{IPC}}} \left\{ \frac{\sum_{\nu \in C} \mathrm{exec}(\nu)}{\mathrm{delay}(C)} \right\}. \quad (3)$$

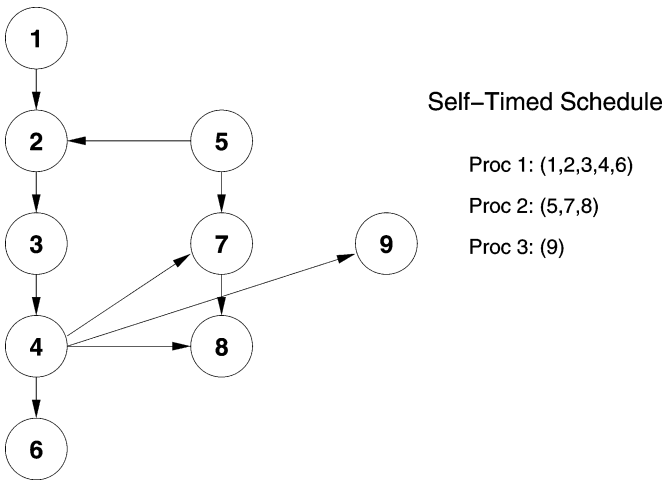

Fig. 1. Example of an application graph and an associated self-timed schedule.

The quotient in (3) is referred to as the *cycle mean* of the associated cycle C. A similar data structure that is useful in analyzing OT implementations is Sriram's *ordered transaction graph model* [12]. Given an ordering $O = \{o_1, o_2, \ldots, o_p\}$ for the communication actors in an IPC graph $G_{\mathrm{IPC}} = (V_{\mathrm{IPC}}, E_{\mathrm{IPC}})$, the corresponding ordered transaction graph $\Gamma(G_{\mathrm{IPC}}, O)$ is defined as the directed graph $G_{\mathrm{OT}} = (V_{\mathrm{OT}}, E_{\mathrm{OT}})$, where $V_{\mathrm{OT}} = V_{\mathrm{IPC}}, E_{\mathrm{OT}} = E_{\mathrm{IPC}} \cup E_O$

$$E_O = \{(o_p, o_1), (o_1, o_2), (o_2, o_3), \ldots, (o_{p-1}, o_p)\} \quad (4)$$

$\mathrm{delay}(o_i, o_{i+1}) = 0$ for $1 \leq i < p$, and $\mathrm{delay}(o_p, o_1) = 1$. Thus, an IPC graph can be modified by adding edges obtained from the ordering O to create the ordered transaction graph.

II. Previous Work

Sriram and Lee [9], [12] discuss some of the advantages and disadvantages of the OT strategy compared to the ST strategy, namely lower synchronization and arbitration costs for the IPC mechanism at the expense of some run-time flexibility and the small additional hardware cost of a simple transaction controller. They also develop a method to compute an optimum transaction order when a fully-static schedule is given beforehand. In this approach, a set of inequalities is constructed using the timing information of the given FS schedule and represented as a graph. The Bellman–Ford shortest path algorithm is applied to this graph to obtain new starting times of the actors, thereby modifying the original FS schedule. A transaction order is then obtained by sorting the starting times of the communication actors. We shall term this method of finding the transaction orders, which is an efficient polynomial-time algorithm, the *Bellman–Ford-Based (BFB)* method. Under an assumption that the cost (latency) of IPC is zero, Sriram shows that the transaction order determined by the BFB technique is always optimal.

More specifically, the developments in [9] show that optimal transaction orders can be derived in polynomial time if IPC costs are negligible; however, the performance of the self-timed schedule is an upper bound on the performance of corresponding ordered transaction schedules under negligible IPC costs. Conversely, we show in this paper that when IPC costs are not negligible, as is frequently and increasingly the case in practice, the problem of determining an optimal transaction order is NP-hard, but at the same time, the performance of a self-timed schedule can be exceeded significantly by a carefully constructed transaction order. Thus, constructing optimal transaction orders is harder under nonnegligible IPC costs, but the potential benefit of employing efficient transaction orders is greater. Furthermore, under nonzero IPC costs, we must resort to heuristics for efficient solutions, the polynomial-time BFB algorithm is no longer optimal, and alternative techniques that account for IPC costs are preferable.

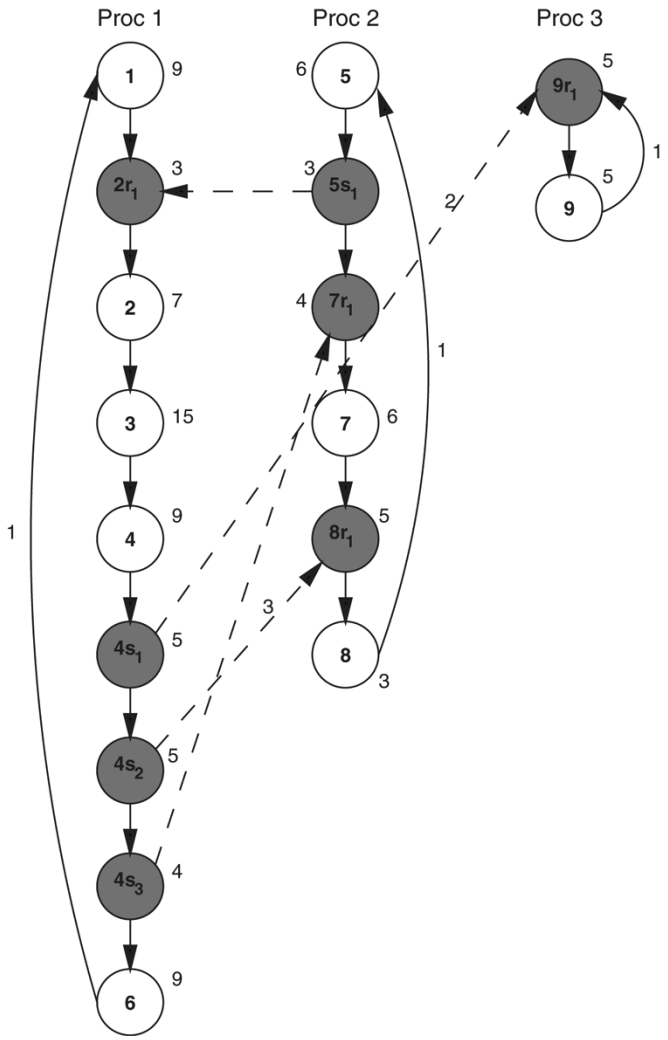

Fig. 2. IPC graph constructed from application graph of Fig. 1.

Numerous approaches have been proposed for incorporating IPC costs into the assignment and ordering steps of scheduling (e.g., [4], [15]). The techniques that we propose in this paper are complementary to these approaches in that they provide a means for mapping the resulting schedules into efficient OT implementations, which eliminate the performance and power consumption overhead associated with run-time synchronization and contention resolution.

For multiprocessor networks utilizing point-to-point links, Surma and Sha [16], [17] have investigated static scheduling of messages using a collision graph model, and have shown that

embedding this model within existing scheduling algorithms can lead to significant improvement.

III. Comparison of Self-Timed and Ordered Transaction Strategies

Given an application graph, an associated multiprocessor schedule, and an FS implementation, an OT implementation, and an ST implementation for the schedule, suppose $T_{\rm FS}, T_{\rm OT}$, and $T_{\rm ST}$, respectively, denote the average iteration periods of the corresponding schedules. In general, when IPC costs are negligible, $T_{\rm FS} \geq T_{\rm OT} \geq T_{\rm ST}$ [12]. This is because the ST method has the fewest constraints. The ST schedule only has assignment and ordering constraints, while the OT schedule has transaction ordering constraints in addition to those constraints, and the FS schedule has exact timing constraints that subsume the constraints in the ST and OT schedules. ST schedules overlap in a natural manner, and eventually settle into a periodic pattern of iterations. This pattern can be exponential in size, and therefore, the ST schedule has the advantage that in successive iterations, the transaction order may be different, while this flexibility is not available for the OT and FS schedules.

In practical cases, however, the IPC cost is nonzero. Depending on the bandwidth of the bus, IPC costs may be quite significant. The throughput of the ST schedule can be computed easily when IPC costs are ignored by calculating the MCM of the corresponding dataflow graph (i.e., via (3). However, when IPC costs are taken into account, this can no longer be done since the notion of bus contention comes into the picture. Not only do the communication actors in the dataflow graph have to wait for sufficient tokens on the input arcs to fire, they also have to wait for the bus to be available—i.e., no other communication actor should be accessing the bus at the same instant of time. Therefore, the throughput of the self-timed schedule is typically derived using simulation techniques, which are time-consuming. On the other hand, the throughput of the OT schedule can still be obtained by calculating the MCM of the transaction order graph since there will be no bus contention when a linear order is imposed on the communication actors [9].

The relation $T_{\rm FS} \geq T_{\rm OT} \geq T_{\rm ST}$ is also no longer valid in the presence of nonzero IPC costs. To see why this is true, assume that two communication actors become *enabled* (have sufficient input tokens to fire) at more or less the same time. Then the ST method will schedule the communication actor that becomes enabled earlier. Doing this may result in a lower throughput since, for example, the processor that contains the communication actor that is scheduled later might be more heavily loaded. The FS and the OT methods avoid such pitfalls by analyzing the schedules at compile time, and producing an exact firing time assignment, or a transaction order that takes the entire schedule into consideration. Intuitively, the ST method follows a more greedy, ASAP approach in choosing which communication actor to schedule next, and this can result in inefficient execution patterns.

Example 1: To illustrate how an ST schedule might perform worse than an OT schedule, consider the IPC graph of Fig. 2. Dashed edges represent inter-processor data dependencies. Numbers beside actors show their execution times, numbers beside edges indicate nonzero delays, xs_y denotes the yth send actor of computation actor x, and xr_y denotes the yth receive actor of x. Fig. 3 shows the periodic pattern into which the ST schedule eventually settles. Although Processor 1 is most heavily loaded, we see that there are instances when the processor is idling waiting for the bus to become free. In contrast, when the transaction order $(5s_1, 2r_1, 7r_1, 4s_1, 4s_2, 4s_3, 8r_1, 9r_1)$ is enforced (Fig. 4), an 11% lower average iteration period results. This is because the transaction order is computed in a fashion that enables the heavily loaded Processor 1 to access the bus whenever required. Such an ability to prioritize strategically-selected transactions is especially important in heterogeneous multiprocessors, which often have unbalanced loads due to variations in processing capabilities of the computing resources.

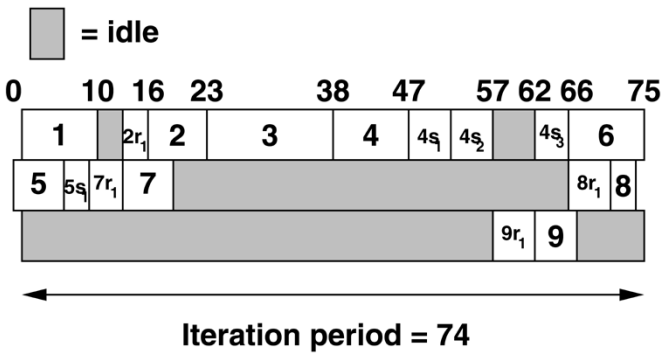

Fig. 3. Gantt chart for ST schedule in Example 1.

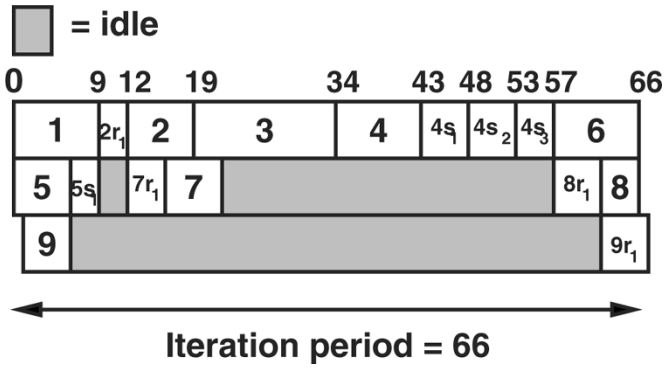

Fig. 4. Gantt chart for OT schedule in Example 1.

The ST approach has the further disadvantage that in the presence of execution time uncertainties, there is no known method for computing a tight worst-case iteration period, *even using simulation techniques*. In particular, the period of the ST schedule obtained by using worst case execution time estimates of the actors does not necessarily give us the worst case iteration period of a schedule. This can prove to be a big disadvantage in real-time systems where worst-case bounds are needed beforehand.

Example 2: Consider the IPC graph of Fig. 5, and suppose that Actor 1 has a worst-case execution time of 21, and a best case execution time of 19. Fig. 6 shows the ST schedule that results when Actor 1 has an execution time of 21. An iteration

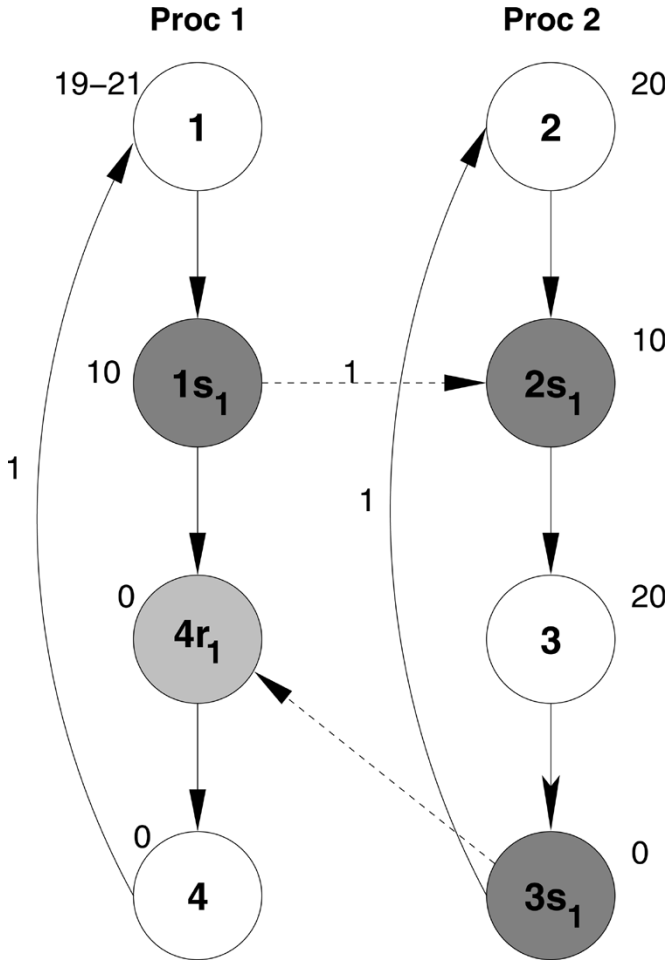

Fig. 5. IPC graph for Example 2.

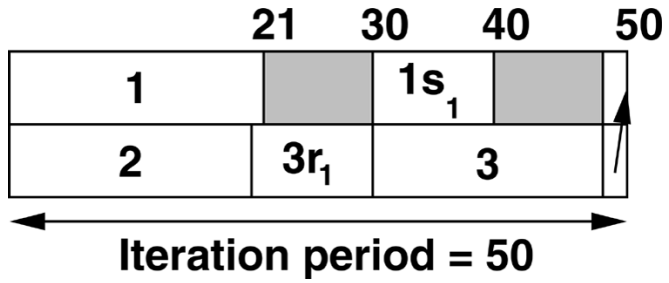

Fig. 6. Gantt chart for ST schedule when $\mathrm{exec}(1) = 21$.

period of 50 is obtained. However, when the same schedule is simulated for an execution time of 19, we obtain an iteration period of 59 as shown in Fig. 7.

In contrast, the iteration period obtained by computing the MCM of the ordered transaction graph with worst-case actor execution times is the worst-case iteration period. This is because the MCM is an accurate measure of performance for ordered transaction implementations [9], [12] and the MCM can only increase or remain the same when the execution time of an actor is increased.

IV. Finding Optimal Transaction Orders

In the *transaction ordering problem*, our objective is to determine a transaction order O for a given IPC graph such that the

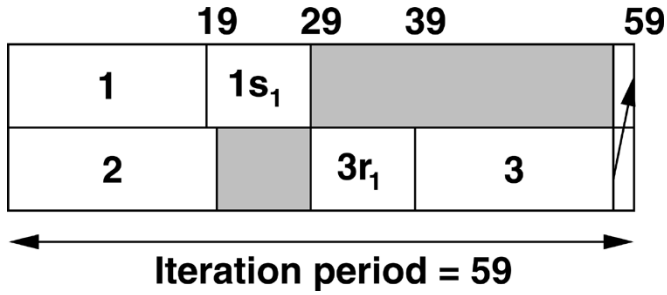

Fig. 7. Gantt chart for ST schedule with $\mathrm{exec}(1) = 19$.

MCM of the resulting ordered transaction graph is minimized (so that throughput is maximized). As mentioned in Section II, it has been shown that this problem is tractable when IPC costs are ignored. In this section, we show that when IPC costs are considered, the transaction ordering problem becomes NP-complete.

We show this by first showing that determining an optimal transaction order for noniterative implementations, which is a more restricted (easier) problem, is NP-complete. To convert an iterative IPC graph to a noniterative one, it suffices to remove all edges in the graph that have delays of one or more. This results in an acyclic graph since any cycle in the original graph must have a delay of one or more for the graph not to be deadlocked.

Definition 1: Given an IPC graph $G_{\mathrm{IPC}} = (V, E)$, the associated *noniterative inter-processor communication (NIPC) graph* is defined as $G_{\mathrm{NIPC}} = (V, E_{\mathrm{IPC}})$ where $E_{\mathrm{NIPC}} = \{e | (e \in E)$ **and** $(\mathrm{delay}(e) = 0)\}$.

Definition 2: Given an NIPC graph $G_{\mathrm{NIPC}} = (V, E_{\mathrm{IPC}})$, and an ordering $O = \{o_1, o_2, \ldots, o_p\}$, the corresponding *noniterative ordered transaction (NOT) graph* $G_{\mathrm{NOT}} = \prod(G_{\mathrm{NIPC}}, O)$ is defined as $G_{\mathrm{NOT}} = (V_{\mathrm{NOT}}, E_{\mathrm{NOT}})$, where $V_{\mathrm{NOT}} = V, E_{\mathrm{NOT}} = (E_{\mathrm{NIPC}} \cup E_O) - \{(o_p, o_1)\}$, and E_O is as defined in (4).

By definition, the total execution time *(makespan)* of a NOT graph G_{NOT} is finite, and this execution time can be determined in polynomial time—as the length of the longest cumulative-execution-time path in G_{NOT}—since G_{NOT} is acyclic and the execution times of all actors are nonnegative. However, given an IPC graph, finding a transaction order that minimizes the makespan of the associated NOT graph is intractable.

Definition 3: The *noniterative transaction ordering* problem is defined as follows. Given an NIPC graph $G_{\mathrm{NIPC}} = (V, E_{\mathrm{NIPC}})$, and a positive integer k, does there exist a transaction order $O = \{o_1, o_2, \ldots, o_n\}$ such that $G_{\mathrm{NOT}} = \prod(G_{\mathrm{NIPC}}, O)$ has a makespan that is less than or equal to k?

To show that noniterative transaction ordering is NP hard, we derive a reduction from the *sequencing with release times and deadlines (SRTD)* problem, which is known to be NP-complete [18]. The SRTD problem is defined as follows.

Definition 4: (The SRTD problem). Given an instance set T of tasks, and for each task $t \in T$, a length (duration) $l(t) \in \mathbb{N}$, a release time $r(t) \in \mathbb{N}$, and a deadline $d(t) \in \mathbb{N}$, is there a single-processor schedule for T that satisfies the release time constraints and meets all the deadlines? That is, is there a one-to-one function (called a *valid SRTD schedule*) $\sigma : T \to \mathbb{N}$,

with $(\sigma(t) > \sigma(t')) \Rightarrow (\sigma(t) \geq \sigma(t') + l(t'))$, and for all $t \in T, \sigma(t) \geq r(t)$, and $\sigma(t) + l(t) \leq d(t)$?

Theorem 1: The noniterative transaction ordering problem is NP-complete.

Proof: This problem is clearly in NP since we can verify in polynomial time whether the longest path length (in terms of cumulative execution time) of the graph is less than or equal to a given positive integer.

Now suppose that we are given an instance of the SRTD problem (T, r, l, d) with $T = \{t_1, t_2, \ldots, t_p\}$. We construct an NIPC graph G_{NIPC} from this instance by carrying out the following steps. Here, all edges instantiated are delay-less unless otherwise specified. Let k be equal to at least the maximum deadline of the tasks in the given instance of the SRTD problem.

For each $t_i \in T$, we have the following:

1) instantiate a send actor u_i when i is odd, or a receive actor u_i when i is even with $\mathrm{exec}(u_i) = l(t_i)$ and $\mathrm{proc}(u_i) = i$;
2) instantiate a computation actor m_i with $\mathrm{exec}(m_i) = r(t_i)$ and $\mathrm{proc}(m_i) = i$;
3) instantiate a computation actor n_i with $\mathrm{exec}(n_i) = k - d(t_i)$ and $\mathrm{proc}(n_i) = i$;
4) instantiate an edge (m_i, u_i) and another edge (u_i, n_i).

Each send actor u_i is connected to the receive actor u_{i+1} by an interprocessor edge (u_i, u_{i+1}) with a delay of unity. Since each of the interprocessor edges has a delay of unity, these edges are not present in G_{NIPC}. Without loss of generality, we assume that there are an even number of tasks, so that the number of send and receive actors is the same (if the number of tasks is not even to begin with, we can instantiate an appropriately-defined dummy actor to generate an equivalent "even-task" instance). Observe from our construction that from the p tasks in the given instance of the SRTD problem, we construct a graph G_{NIPC} that involves p processors, p communication actors, $2p$ computation actors, and $2p$ edges.

Claim: If there exists a transaction order O for $G_{\mathrm{NOT}} = \prod(G_{\mathrm{NIPC}}, O)$ that will have a makespan that is less than or equal to k, then there exists a valid SRTD schedule for the given instance of the SRTD problem.

The reasoning behind our construction and the above claim is that we make the communication actors of the ordered transaction correspond exactly to the tasks of the SRTD problem. We do this by making the execution time of the computation actor before each corresponding communication actor equal to the release time of the associated task and, thus, guarantee that the communication actors cannot begin execution before their respective release times. Also, since computation actors will begin execution from time 0 as each is on a different processor, the release times correspond to the time at which they complete execution. Similarly, the execution times of the computation actors that follow the communication actors are chosen to be $k - d(t_i)$ so that the corresponding communication actors must complete their execution before $d(t_i)$ for the makespan to be less than or equal to k. This is true because the computation actor can begin execution immediately after the communication actor has finished. Therefore, the valid SRTD schedule corresponds exactly to the shared bus schedule in the derived instance of the noniterative transaction ordering problem. If we can find a transaction order that has a makespan less than or equal to k, we have a bus schedule that schedules the communication actors in the same manner as an appropriate single-processor schedule for the corresponding SRTD tasks. Conversely, if a transaction order cannot be found that satisfies the given makespan constraint, it is easily seen that there is no valid SRTD schedule for the given instance of the SRTD problem. $Q.E.D.$.

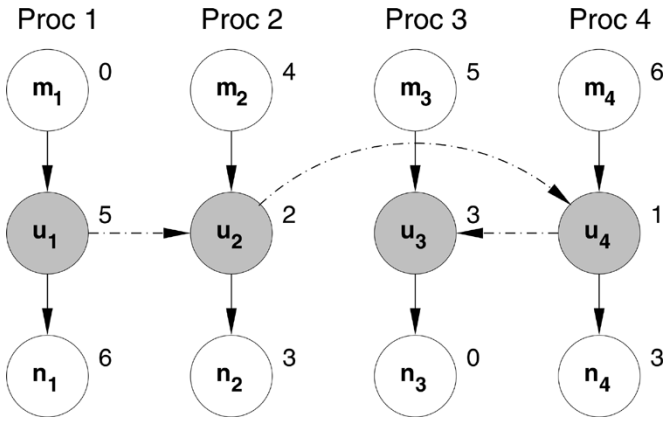

Fig. 8. NOT graph constructed in Example 3.

Note that in Theorem 1, we have simplified the problem greatly by assuming the interprocessor edges to have unit delays. This removes the interdependencies that are imposed by these edges, but even with this simplification, the problem remains NP-complete.

Example 3: Suppose that we are given an instance of the SRTD problem with task set $T = \{t_1, t_2, t_3, t_4\}$; and respective release times $r(T) = \{0, 4, 5, 6\}$, lengths $l(T) = \{5, 2, 3, 1\}$, and deadlines $d(T) = \{5, 8, 11, 8\}$. To construct an instance of the noniterative transaction ordering problem with $k = 11$, we create 4 processors each with 3 vertices. The execution times are determined from above—e.g., $u_1 = 5, m_1 = 0, n_1 = 6$. The resulting NOT graph is illustrated in Fig. 8. Dash-dot edges indicate OT edges. Removing the dash-dot edges that represent the transaction order edges gives us the NIPC graph constructed from above. This figure shows a transaction order (u_1, u_2, u_4, u_3) where the schedule length of 11 is satisfied. This means that there exists a valid SRTD schedule for the given SRTD problem instance. The start times of the tasks can be obtained by finding the longest path lengths between the source nodes and the corresponding communication actors. Setting the starting times of the tasks (t_1, t_2, t_3, t_4) to equal $(0, 5, 8, 7)$, respectively, we obtain a valid SRTD schedule for the SRTD problem instance.

As demonstrated by Theorem 2 below, we can extend the Proof of Theorem 1 to show that the transaction ordering problem is NP-complete in the iterative context as well as the noniterative case.

Definition 5: The *iterative transaction ordering* problem (also called the *transaction ordering problem*) is defined as follows. Given an IPC graph G_{IPC} and a positive integer k, does there exist a transaction order O such that $G_{\mathrm{OT}} = \Gamma(G_{\mathrm{IPC}}, O)$ satisfies $\mathrm{MCM}(G_{\mathrm{OT}}) \leq k$?

Theorem 2: The iterative transaction ordering problem is NP-complete.

Proof Sketch: The MCM can be found in polynomial-time, therefore, the problem is in NP.

To establish NP-hardness, we again derive a reduction from the SRTD problem, and we modify the graph construction from the Proof of Theorem 1 so that the MCM equals the makespan. Details are omitted due to page limitations.

V. The Transaction Partial Order Heuristic

The BFB technique does not take bus contention into consideration while scheduling the transaction order. Instead, it tries to find a transaction order that will be close to or equal to that of the associated self-timed schedule. However, we have demonstrated that in the presence of nonzero IPC costs, the OT method can, in fact, perform significantly better than the ST method, and thus, more direct consideration of OT execution is clearly worthwhile when scheduling transactions. For this purpose, we propose in this section a heuristic, called the *transaction partial order (TPO) algorithm*, that simultaneously takes IPC costs and the serialization effects of transaction ordering into account when determining the transaction order. Note that OT edges added to the IPC graph can only increase the MCM of the IPC graph, or leave the MCM unchanged. The MCM of the original IPC graph therefore represents a lower bound on the achievable average iteration period. By adding OT edges, we are effectively removing bus contention by making sure that no two communication actors submit conflicting bus requests, and this generally increases the MCM of the IPC graph. The TPO heuristic finds a transaction order on the basis that an OT edge that increases the MCM of the IPC graph by a comparatively smaller amount should be given preference. Therefore, to determine which communication actor should be scheduled first, we insert OT edges between communication actors that are contending for the bus (during the transaction ordering process), and calculate the corresponding MCM of the IPC graph. Actors whose corresponding MCM's are more favorable under such an evaluation are scheduled earlier in the transaction order.

We note that in a *correct* transaction order, the OT edges should not introduce any zero delay cycles into the IPC graph. Such cycles would create deadlock in the system. Recall that if we remove the nonzero delay edges from G_{IPC}, the resulting graph G_{NOT} is acyclic. We define a graph G'_{NOT} which is constructed by adding the OT edges to G_{NOT}. In a correct transaction order, G'_{NOT} is also acyclic, and so we can perform a topological sort of G'_{NOT}. In any correct transaction ordering, there must exist some topological sort of G'_{NOT} such that for all OT edges e, the source vertex of e precedes the target vertex of e in the topological sort. We therefore see that the solution space of possible transaction orderings is a subset of the possible topological sorts of G'_{NOT}. Unfortunately, there can be an exponential number of possible topological sorts for a graph. For example, a complete bipartite graph with $2n$ nodes has $(n!)^2$ possible topological sorts.

Instead of operating directly on G_{NOT}, we derive a *transaction partial order (TPO)* graph G_{TPO}. The transaction partial order graph incorporates the minimum set of dependencies imposed among different processors by the communication actors of the IPC graph. These dependencies must be obeyed by any ordering of the communication operations. Fig. 9 gives pseudo-code for generating G_{TPO}. The algorithm attempts to remove as many noncommunication nodes as possible while maintaining dependencies between communication actors. Since we allow for IPC graphs with overlapped communication and computation, some of the computation nodes have both multiple predecessors and multiple successors. These cannot be removed since this would require imposing some additional dependencies on these nodes, and we want G_{TPO} to represent the minimal set of dependencies. The algorithm proceeds in multiple passes, and terminates when no more computation actors can be removed.

Algorithm V.1: GENERATE TPO GRAPH(G_{IPC})

```
input: IPC Graph G_IPC
output: Transaction Partial Order TPO Graph
finished ← FALSE
for (∀ edges e ∈ G_IPC)
  do { if (e is a feedback edge)
       then {Delete e
while (finished = FALSE)
  do { finished = TRUE
       for (∀ nodes v ∈ G_IPC)
         do { if (v is a computation node)
              then { if ((indeg(v) = 0) OR (outdeg(v) = 0))
                     then { Delete v
                            finished = FALSE
                   else if (indeg(v) = 1)
                     then { p ← predecessor node of v
                            for (∀ successors s of v)
                              do {Create edge (p, s)
                            Delete v
                            finished = FALSE
                   else if (outdeg(v) = 1)
                     then { s ← successor node of v
                            for (∀ predecessors p of v)
                              do { Create edge (p, s)
                                   Delete v
                                   finished = FALSE
```

Fig. 9. Pseudo-code for generating the TPO graph from the IPC graph.

Example 4: Fig. 10 shows a TPO graph derived from an IPC graph using the algorithm given in Fig. 9. Two passes were required to reduce the graph. Notice that all the dependencies imposed by the IPC graph are retained in the TPO graph.

The TPO heuristic proceeds by considering—one by one—each vertex of G_{TPO} that has no input edges (vertices in the TPO graph that have no input edges are called *ready* vertices) as a *candidate* to be scheduled next in the transaction order. Interprocessor edges are drawn from each candidate vertex to all other ready vertices in G_{IPC}, and the corresponding MCM is measured. The candidate whose corresponding MCM is the least when evaluated in this fashion is chosen as the next vertex in the ordered transaction, and deleted from G_{TPO}. This process is repeated until all communication actors have been

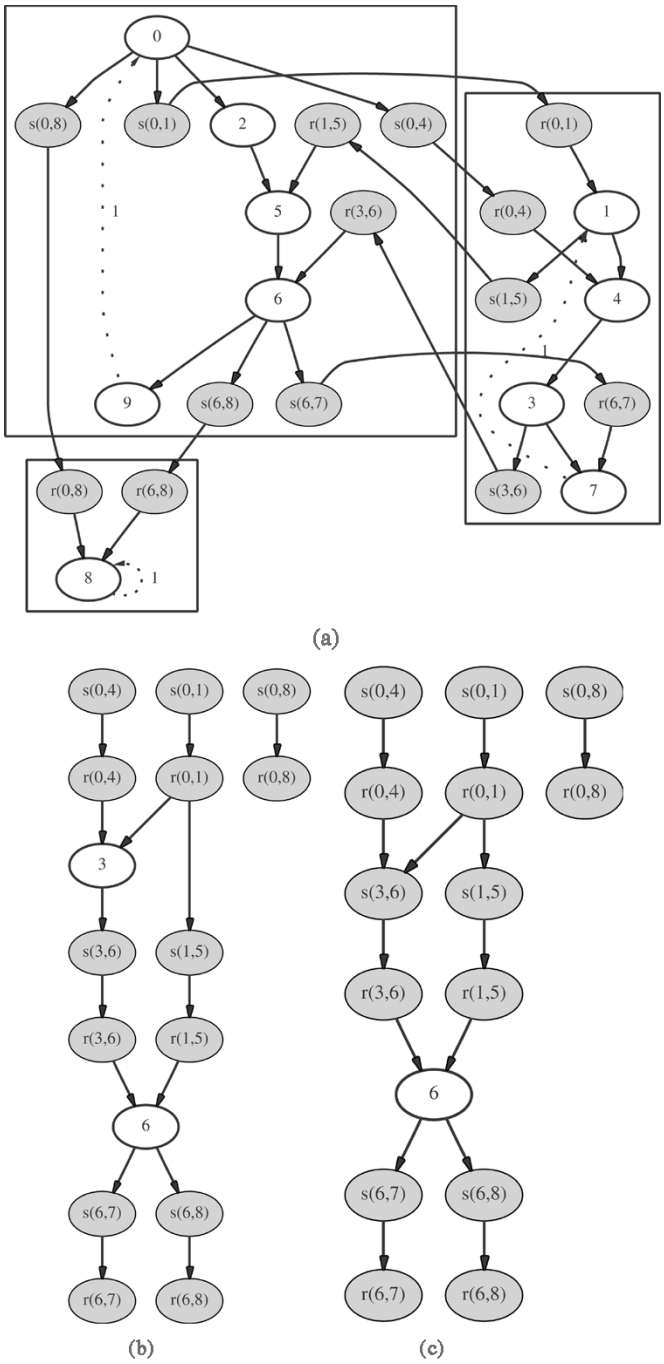

Fig. 10. The TPO graph derived from the IPC graph in (a) after 1 pass (b) and two passes (c) of the algorithm given in Fig. 9. Note the labeling of the IPC tasks (shaded)—$s(i,j)$ represents a *send* from computational task i to computational task j while $r(i,j)$ represents a *receive* from computational task i to computational task j.

Algorithm V.2: CHOOSE COMMUNICATON ACTOR(G_{IPC}, readyList)

input: ipc graph G_{IPC}
input: list of actors readyList
output: communication actor ν
if readyList.size() $\equiv 1$
  then {$\nu \leftarrow$ readyList.head()
for $x \in$ readyList
  do { **for** $y \in$ readyList
    do { **if** $x \neq y$
      then { $e = G_{\text{IPC}}$.addedge(x,y)
        temp.add(e)
      criteria$[x] \leftarrow$ MCM(G_{IPC})
      for $e \in$ temp
        do {G_{IPC}.delete(e)
$\nu \leftarrow$ min(criteria$[x]$)

(a)

Algorithm V.3: TPO HEURISTIC(G_{IPC}, transactionOrder)

input: IPC graph G_{IPC}
output: linear list of communication actors transactionOrder
compute G_{TPO} from G_{IPC}
for $\nu \in G_{\text{IPC}}$
  do { mark$[\nu] \leftarrow$ FALSE
    if indegree$(\nu) = 0$
      then {readyList.append(ν)

complete $\leftarrow$ FALSE
first $\leftarrow$ TRUE
while (complete $\neq$ TRUE)
  { $\nu \leftarrow$ CHOOSE-COMMUNICATION-ACTOR$(G_{\text{IPC}}, G_{\text{TPO}}$, readyList)
  mark$[\nu] \leftarrow$ TRUE
  transactionOrder.append(ν)
  if (first)
    then {first $\leftarrow$ FALSE
    else {G_{IPC}.addedge(w,ν)
  $w \leftarrow \nu$
  do { **for** $u \in (\nu,u) \in E$
    do { flag $\leftarrow$ TRUE
      for $s \in (s,u) \in E$
        do { **if** (mark(s) = FALSE)
          then {flag $\leftarrow$ FALSE
      if (flag)
        then {readyList.append(u)
  if (readyList.empty() = TRUE)
    then {complete $\leftarrow$ TRUE

(b)

Fig. 11. Pseudocode for the TPO heuristic.

scheduled into a linear ordering. A pseudocode specification of the TPO heuristic is given in Fig. 11.

The algorithm makes sense intuitively since the dependencies imposed by the edges drawn from the candidate vertices will remain when the transaction ordering O is enforced. These edges represent constraints in addition to the interprocessor edges that are already present in G_{IPC} and, thus, they can only increase the MCM or leave the MCM unchanged. Since we are interested in minimizing the MCM, we choose candidate vertices that increase the MCM by the least possible amounts. Thus, the algorithm follows a greedy strategy in choosing vertices, but it explicitly takes communication serialization and IPC costs into account.

When we apply the TPO heuristic to the IPC graph of Fig. 2, the schedule we obtain is illustrated by the Gantt chart of Fig. 4.

The OT edges corresponding to the actors that have already been scheduled are added as the heuristic proceeds since they represent the schedule of the bus, and hence, make the heuristic more accurate for the later stages of the transaction order. The maximum number of nodes in the ready list at any given instant

is P (where P is the number of processors). The complexity of the algorithm is thus $O(P|V|^2|E_{\rm OT}|)$ since the complexity of computing the MCM of a graph (V, E) is $O(|V||E|)$.

The edge of the transaction order that connects the last communication actor in the ordering with the first one has a delay of unity (to represent the transition to the next graph iteration). We can improve the performance of the TPO algorithm by introducing this edge at the beginning because it will give a more accurate estimate of the MCM in choosing vertices later as the heuristic proceeds. Under this modification, the heuristic proceeds as before, except that the "last" (unit-delay) transaction ordering edge is drawn at the beginning. Since $G_{\rm TPO}$ has a maximum of P communication actors that can be scheduled last in the transaction order, the modified heuristic is repeated for each of these candidate communication actors that can be scheduled in the end, and the best solution that results is selected. This increases the complexity of the algorithm by a factor of P to $O(P^2|V|^2|E_{\rm OT}|)$.

VI. Branch and Bound Strategy

Since the transaction ordering problem is intractable, we are unable to efficiently find optimal transaction orders on a consistent basis. We have implemented a *branch and bound (BB)* strategy to explore the search space comprehensively. The branch and bound approach gives us a lower bound on the iteration period that can be achieved and provides us with a useful measure of comparison.

The branch and bound strategy initially computes the list of actors that are ready from $G_{\rm TPO}$. A candidate vertex is chosen and deleted from $G_{\rm TPO}$ and the ready list is updated. In successive iterations, an edge is drawn in $G_{\rm IPC}$ from the previous candidate actor to the current candidate and the MCM is computed. If the computed MCM is less than the lowest calculated MCM then the process is repeated recursively; otherwise, the candidate is discarded and the next one chosen. The branch and bound approach has the advantage of being able to prune the search space effectively but since the MCM has to be computed after each edge is added rather than after all the edges have been added, it is unable to find optimum transaction orders for complex graphs.

VII. Genetic Algorithm for Transaction Scheduling

The branch and bound approach requires excessive amounts of time for graphs that have significant numbers of IPC edges. To develop an alternative to this branch and bound approach, and the TPO heuristic, we have implemented a genetic algorithm (GA) to search for the best transaction order. The GA exploits the increased tolerance for compile time that is available for many embedded applications [19], and can leverage the TPO heuristic by incorporating its solution in the "initial population."

In our GA formulation, candidate transaction orders are encoded using the matrix-based sequence-encoding method described in [20]. Using this method, the partial order of the communication actors is converted into a precedence matrix and randomly completed to yield a random transaction order that is valid. Mutation is carried out by swapping rows and columns, and recombination is performed using the *intersection operator* explained in [20]. The intersection operator takes subsequences that are common among the parents by taking the boolean "and" of the two parent matrices to form the "offspring," and the undefined part is randomly completed.

The mutation step takes $O(|V|^2)$ time multiplied by the number of swaps carried out since each time we have to check whether the swap was valid by comparing it with the partial boolean matrix $M_{\rm TPO}$ corresponding to the transaction partial order graph $G_{\rm TPO}$. The recombination step takes $O(|V|^2)$ time, and the evaluation step takes $O(|V||E_{\rm OT}|)$ time. The overall complexity of each iteration is also influenced by the population size and the overhead involved in generating random numbers.

VIII. Dynamic Reordering

Once we obtain a transaction order (e.g., using the TPO heuristic or the GA approach defined in Section VII), it is possible to swap the position of consecutive communication actors in the transaction order as long as the new positions do not violate the dependencies imposed by the transaction partial order. This method has the advantage that it cannot degrade the transaction order since we can discard any solution that is worse. The concept is similar to *dynamic variable reordering* used in ordered binary decision diagrams (OBDDs) [21]. OBDD's are data structures used for representation and manipulation of Boolean functions often applied in VLSI CAD. The choice of the variable ordering largely determines the size of the BDD; its size may vary from linear to exponential. Dynamic reordering is a technique by which neighboring variables are swapped to find a better variable order. Swapping of neighboring variables can be performed in linear time. We have implemented an adaptation to ordered transaction scheduling, called *dynamic transaction reordering (DTR)*, of the *sifting algorithm* introduced by Rudell [22]. The sifting algorithm tries to find the best position for a variable. Each variable is exchanged with its successor variable until the variable is sifted to the bottom of the directed acyclic graph (DAG). Then the variable is exchanged with its predecessor variable until it is shifted to the top of the DAG. The best DAG size seen during the search is remembered and the position of the variable is restored. We have observed that from DTR, we consistently obtain improvements in the iteration period, regardless of the method used to find the transaction order.

IX. Simulator

We have developed a software simulator of the execution of self-timed iterative schedules. It was developed under UNIX using the LEDA C++ programming library [23]. The simulated system is a shared-memory architecture. In this architecture, synchronizations are performed by accessing the shared memory bus. Data tokens associated with the IPC constraints are transferred via the shared memory. When a processor tries to gain access to the bus for a synchronization or interprocessor data communication, the system permits access if the bus is not in use, otherwise it denies access and the processor waits a specified back-off time, then tries again.

The synchronization cost for OT is much lower than the synchronization costs for ST. In the OT strategy a shared bus access takes no more than a single read or write cycle on the processor,

and the overall cost of communicating one data sample is two or three instruction cycles [12]

Our simulator for ST operation implements both the *Unbounded Buffer Synchronization (UBS)* and the *Bounded Buffer Synchronization (BBS)* protocols [10]. In the BBS protocol (used with feedback synchronization edges), the protocol requires one local memory increment operation (the local write pointer) and one write to shared memory (store write pointer) occur after the source node of the synchronization edge has executed. The initial value of the write pointer is set to the delay of the synchronization edge e. The BBS synchronization before the execution of $\mathrm{snk}(e)$ involves a comparison between the value stored in shared memory and a local value (the local read pointer). The initial value of the local read pointer is 0. This comparison is repeated until the read pointer and the saved write pointer are not equal, at which time the actor $\mathrm{snk}(e)$ can execute.

In the UBS protocol (used with feed-forward synchronization edges), the local read and write pointers are maintained and initialized in the same manner as in the BBS protocol. The value stored in shared memory is no longer a copy of the write pointer, but is rather a count (initialized to the edge delay) of the number of unread tokens. After $\mathrm{src}(e)$ executes, the shared count is repeatedly examined until it is found to be less than the feed-forward buffer size, at which point the IPC operation can proceed. The count is incremented and $\mathrm{snk}(e)$ can execute. Before $\mathrm{snk}(e)$ executes, the value of the shared count is repeatedly checked until it is nonzero. Then the read operation can proceed, and the count is decremented.

We assume an architecture where all synchronization and memory accesses occur in a single shared memory. We define a parameter β to be the ratio of the synchronization time to the IPC time. Since we are considering HSDF graphs with one data token produced per IPC operation, we have $\beta \geq 2$ for BBS (at least 2 memory accesses for synchronization for every data memory access) and $\beta \geq 4$ for UBS.

A. Task Execution Times

For many DSP applications, it is possible to obtain accurate statistics on task execution times. Probabilities for events such as cache misses, pipeline stalls, and conditional branches can be obtained by using sampling techniques or simulation of the target architecture [24]. We utilize the task execution model in [25], where each task v_i in the task graph $G = (V, E)$ is associated with three possible execution times e_0, e_1, or e_2 with probabilities p_0, p_1, and p_2 respectively. Here, e_0 is the task execution time given in the benchmark specification, $e_1 = 2e_0$ and $e_2 = 4e_0$. We define a single parameter p for the degree of randomness of the task execution times, where $p_0 = (1-p), p_1 = p(1-p)$, and $p_2 = p^2$. Note that for this probability distribution, $p = 0.5$ corresponds to the highest degree of randomness. Under these assumptions, we note that the FS strategy is not practical for any $p > 0$, no matter how small, since a FS architecture must operate with e_2 for all tasks in order to assure correctness.

X. Results

Experiments were carried out to compare the ST method and the OT method, and to measure the performance of the TPO, GA, and DTR heuristics in finding transaction orders. The algorithms presented in Sections V–VIII were implemented in C/C++ using the LEDA [23] framework for fundamental graph theoretic data structures and algorithms. The benchmarks are standard DSP applications that have been scheduled using the *DLS* algorithm [26].

The IPC graphs are fairly complicated, ranging from between 9–764 nodes, and the numbers of processors involved range from 3 to 8. The examples *fft1, fft2*, and *fft3* result from three representative schedules for Fast Fourier Transforms based on examples given in [20]; *karp10* is a music synthesis application based on the Karplus Strong algorithm in 10 voices; the *video coder* is taken from [27], *cddat* is a CD to digital audio tape converter taken from the Ptolemy suite [1], *laplace* is a Laplace transform from [28], and irr is an adaptation of a physics algorithm [29].

The iteration period for the ST schedule is obtained using the simulator described in Section IX, while the iteration period of the OT schedule is obtained from the MCM of G_{OT}. We used the task execution model from Section IX-A, and calculated the average iteration periods over 10 000 iterations.

We define a parameter α that quantifies the IPC overhead in a given schedule. It is calculated from the ratio of the total IPC time (synchronization plus data communication) to the total execution time spent on computational tasks over all processors. Thus, α is a function of p, the schedule, and the relative speed of processor to memory. We note that VLSI is trending toward higher relative processor-to-memory speeds (higher α) as gate lengths decrease.

Fig. 12 plots T_{OT} and T_{ST} versus the parameter p that governs the degree of randomness of the task execution times, for different values of β, the ratio of synchronization-to-IPC overhead described in Section IX. The calculations for $\beta = 0$ do not correspond to any synchronization protocol in our architectural model, but are given as a point of reference. Values of $\beta < 2$ would be possible if a separate (faster) memory were allocated to the synchronization variables. Fig. 13 plots these same measures for the fft1 and Laplace benchmarks.

The iteration period increases with p since the average execution time increases with p. For many DSP applications $p < 0.1$ is a reasonable assumption. For example, if e_2 corresponds to a cache miss, e_1 to a processor pipeline stall, and e_0 to the base execution time for a task, $p = 0.1$ corresponds to a 1% cache miss probability, a 9% pipeline stall probability, and a 90% probability for the base execution time.

From Fig. 12 we see that T_{ST} increases more slowly as a function of p than does T_{OT}. This is because the self-timed schedule is able to adapt to changes in execution times, while the OT schedule is fixed. This behavior was observed with all the benchmarks. We also see that it is possible for OT to outperform ST for $\beta = 0$, as explained in Section III, but only for small p. Comparing Fig. 12(a) and (b), we see that the slopes of the curves decrease as α increases. This is because the IPC operations are not random, and so as IPC increases, a smaller fraction of the overall execution time comes from random tasks.

We also observe that the relative improvement of OT over ST increases as α increases. Figs. 14 and 15 plot the ratio $T_{\mathrm{ST}}/T_{\mathrm{OT}}$

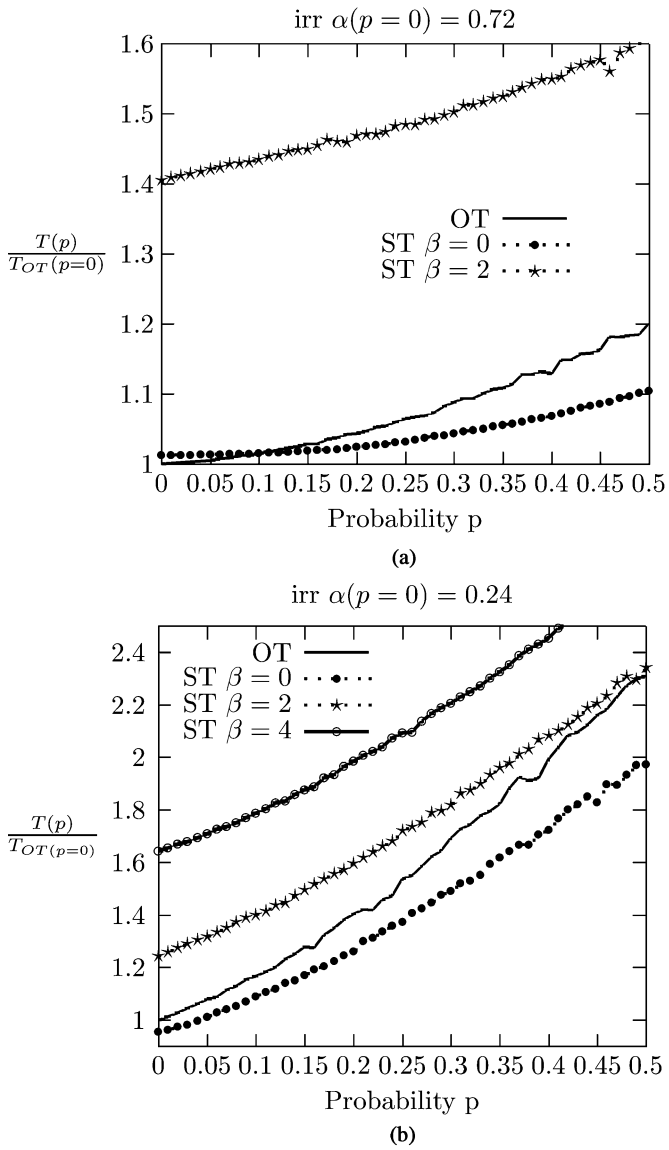

Fig. 12. Iteration periods T_{OT} and T_{ST} for the *irr* benchmark versus amount of randomness in task execution times (parameter p) for two different values of IPC overhead (parameter α).

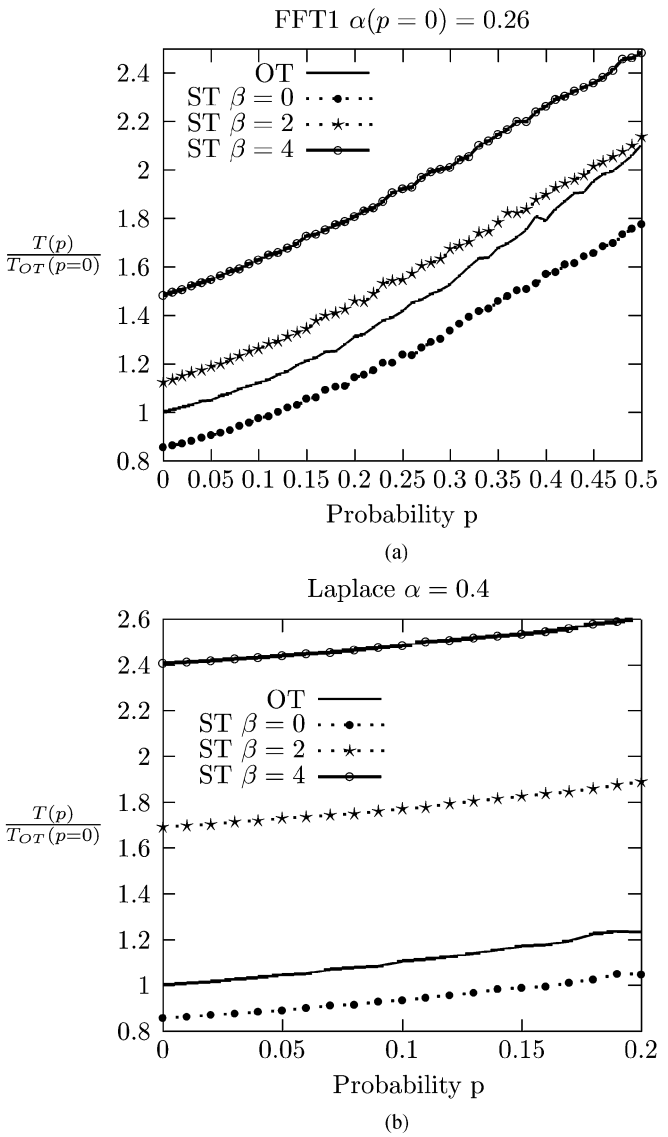

Fig. 13. Iteration periods T_{OT} and T_{ST} for the *fft1* (a) and *Laplace* (b) benchmarks versus the amount of randomness in task execution times (parameter p) for two different values of IPC overhead (parameter α).

versus α for the irr and fft1 benchmarks. For the irr benchmark, $T_{\mathrm{ST}}/T_{\mathrm{OT}} > 1$ for all $\beta \geq 2$ and $p \leq 0.4$. For the fft1 benchmark with $\beta = 2$ and $p = 0.4, T_{\mathrm{ST}}/T_{\mathrm{OT}} > 0.96$, and $T_{\mathrm{ST}}/T_{\mathrm{OT}} > 1$ elsewhere. As discussed above, $p = 0.4$ represents a high degree of uncertainty for task execution times in DSP applications (with $p = 0.5$ representing the highest possible degree of randomness in the probability distribution).

Table I compares the performance (iteration period) of the ST and the OT schedules. In all cases, we observe that the OT strategy outperforms the ST strategy for $\beta \geq 2$. As noted before, $\beta = 2$ and $\beta = 4$ represent lower bounds for the BBS and UBS synchronization protocols, respectively.

Table II gives us a comparison between the different heuristics in finding transaction orders. Each entry is the iteration period when the transaction order found by the heuristic is enforced. Column 2 shows the iteration period when a randomly-generated transaction order is enforced and T_C indicates the iteration period when DTR is used to refine the transaction order obtained using TPO and the solution is incorporated into the initial population of the GA. From the table we can conclude that all the heuristics work fairly well compared to the random transaction order. The TPO heuristic for which the results have been shown is the modified version that inserts the delay beforehand. This consistently gives us a slight improvement. Generally, the TPO heuristic works better than the BFB technique—especially for *cddat*—and the heuristic that combines the TPO heuristic and DTR performs quite well (even better than the GA which, takes significantly more time to execute). The GA was implemented with a population size of 100 and the number of iterations was set to 1000. The GA for the experiments that we tried generally stabilized before the 1000 iteration limit was reached.

When we use the transaction ordering obtained by the TPO heuristic combined with DTR in the initial population of the GA, we achieve the best results since we simultaneously obtain the benefits of all three approaches. The branch and bound strategy shows us the lower bound that the OT method can achieve but

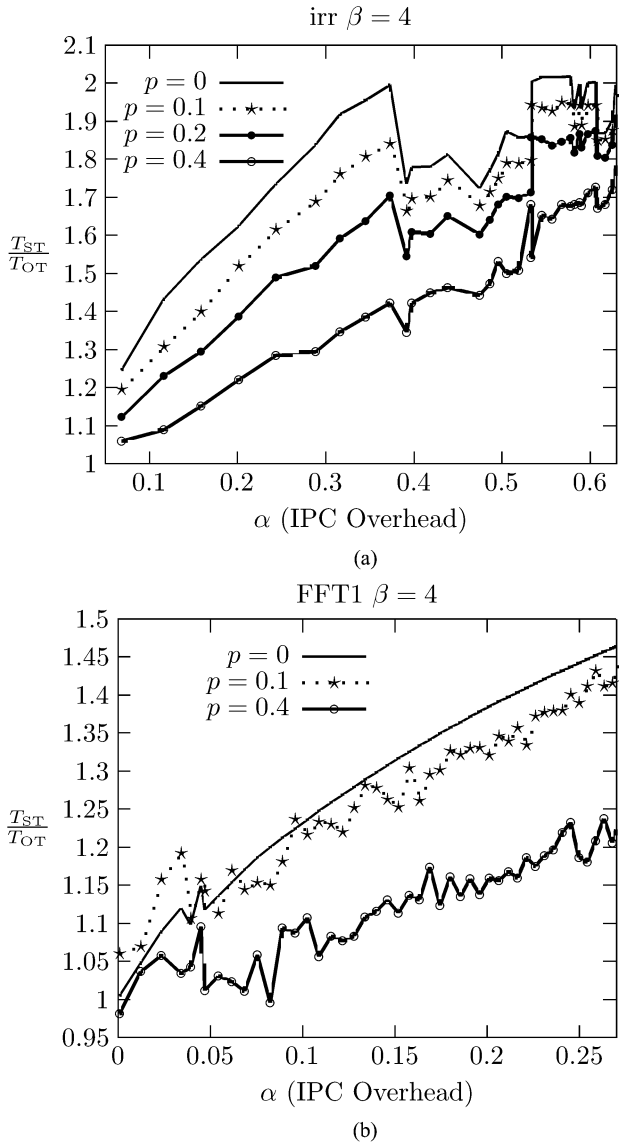

Fig. 14. Ratio of T_{ST} to T_{OT} versus IPC overhead for irr and fft1, with the synchronization-to-IPC ratio $\beta = 4$.

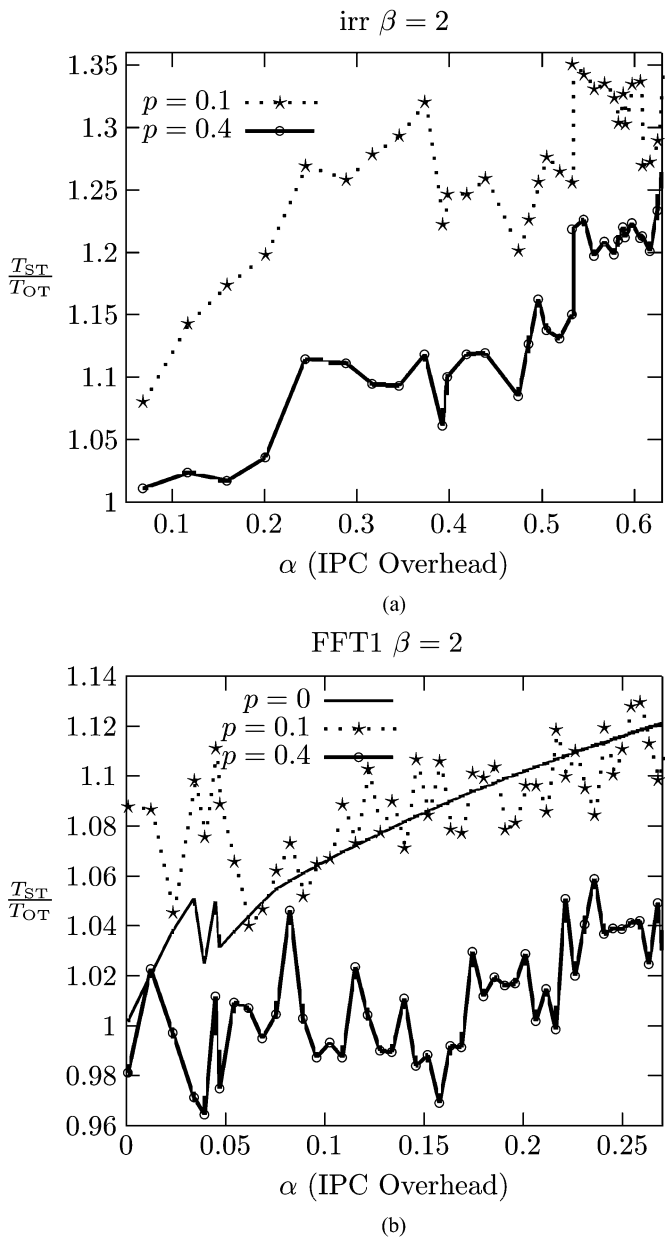

Fig. 15. Ratio of T_{ST} to T_{OT} versus IPC overhead for irr and fft1, with the synchronization-to-IPC ratio $\beta = 2$.

since the complexity of the BB method is exponential, we are unable to achieve results for larger benchmarks. The results are shown in Table II.

XI. Conclusion

We have demonstrated that the ordered transaction method—which is superior to the self-timed method in its predictability, and its total elimination of synchronization overhead—can significantly outperform self-timed implementation, even though the ordered transaction implementation offers less run-time flexibility due to a fixed ordering of communication operations. AWhen synchronization cost is taken into account, the ordered transaction method performs significantly better than the self-timed method. We have studied the relative behavior of OT and ST implementations under a realistic model for task execution times. The OT strategy performs better relative to the ST strategy for lower p (degree of randomness in task execution times), higher β (synchronization costs), and higher α (IPC overhead). The ranges in which OT favors ST encompass the design space that we are targeting—namely, low-cost shared bus embedded multiprocessor DSP systems.

TABLE I
T_{ST}/T_{OT} FOR ST AND OT SCHEDULES

			$p = 0$			$p = 0.1$		
Application	$(\|V\|, \|E\|)$	α	$\beta = 0$	$\beta = 2$	$\beta = 4$	$\beta = 0$	$\beta = 2$	$\beta = 4$
fft1	(28, 32)	0.04	0.970	1.024	1.097	0.997	1.064	1.088
fft1	(28, 32)	0.26	0.858	1.146	1.483	0.872	1.135	1.452
fft2	(28, 32)	0.02	0.995	1.035	1.083	1.042	1.058	1.118
fft3	(28, 32)	0.04	1.020	1.665	2.147	1.001	1.632	2.104
karp10	(21, 29)	0.19	0.896	1.372	1.895	0.896	1.273	1.687
video coder	(9, 9)	0.735	1.014	1.117	1.441	0.933	1.028	1.326
cddat	(760, 764)	0.375	1.081	1.271	1.640	1.006	1.182	1.525
irr	(41, 69)	0.72	1.013	1.405	2.140	1.001	1.414	2.130
irr	(41, 69)	0.24	0.956	1.246	1.645	0.934	1.082	1.470
laplace	(16, 24)	0.4	0.857	1.692	2.406	0.847	1.603	2.248
laplace	(16, 24)	0.14	1.025	1.316	1.608	0.979	1.186	1.376

TABLE II
COMPARISON OF ALGORITHMS

Application	T_{random}	T_{BFB}	T_{TPO}	T_{GA}	$T_{\text{TPO+DTR}}$	T_C	T_{BB}
fft1	392	280	245	255	245	245	
fft2	395	340	320	300	300	295	
fft3	390	300	255	255	245	245	
karp10	482	312	309	308	309	305	303
video coder	182	147	145	145	145	145	145
video decoder	33	33	33	33	33	33	33
cddat	5675	4509	4086	4236	4038	4022	
multirate1	1746	1312	1316	1312	1312	1312	
multirate2	364	322	318	316	316	316	
qmf4	196	148	145	140	145	140	140

We have also shown that in the presence of nonzero IPC costs, finding an optimal transaction order is an NP-complete problem, and we have developed a variety of heuristic techniques to find efficient transaction orders. These techniques include a low-complexity, deterministic heuristic for rapid design space exploration, a dynamic reordering technique for improving a given transaction order, and a genetic algorithm for exploiting extra compile time when generating final implementations. We have also developed a detailed simulator to measure the performance of the self-timed schedule under different constraints.

The benefits of OT can be expected to increase with the general trend in VLSI technology for increasing processor/memory performance disparity. Some of this benefit may be offset, however, by another trend, which is for decreasing predictability in application behavior (and thus execution times) due to the use of more and more sophisticated and adaptive types of algorithms. The evolution of the MPEG standards is an example of this. Further research on OT methods to efficiently handle such lower degrees of predictability is therefore an interesting and important direction for further study.

Useful directions for further work include incorporating transaction ordering considerations into the scheduling process; integrating transaction ordering and retiming of synchronous dataflow graphs [30]; the exploration of hybrid scheduling strategies that can combine ordered transaction, self-timed, and fully-static strategies in the same implementation based on subsystem characteristics; and extension to more general DSP modeling techniques, such as stream-based functions [31], cyclo-static dataflow [32], multidimensional dataflow [33], and parameterized dataflow [34].

REFERENCES

[1] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems," *Int. J. Comput. Simul.*, vol. 4, pp. 155–182, Apr. 1994.

[2] T. C. Hu, "Parallel sequencing and assembly line problems," *Oper. Res.*, vol. 9, no. 6, pp. 841–848, Nov. 1961.

[3] K. Parhi and D. G. Messerschmitt, "Static rate optimal scheduling of iterative dataflow programs via optimum folding," *IEEE Trans. Comput.*, vol. 40, no. 2, pp. 178–194, Feb. 1991.

[4] G. C. Sih, "Multiprocessor Scheduling to Account for Interprocessor Communication," Ph.D. Dissertation, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, CA, Apr. 1991.

[5] S. Y. Kung, P. S. Lewis, and S. C. Lo, "Performance analysis and optimization of VLSI dataflow array," *J. Parallel Distrib. Comput.*, pp. 592–618, 1987.

[6] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous dataflow programs for digital signal processing," *IEEE Trans. Comput.*, vol. C-36, no. 2, Feb. 1982.

[7] E. A. Lee and J. C. Bier, "Architectures for statically scheduled dataflow," *J. Parallel Distrib. Comput.*, vol. 10, pp. 333–348, Dec. 1990.

[8] E. A. Lee and S. Ha, "Scheduling strategies for multiprocessor real-time DSP," in *Proc. GLOBECOM*, Nov. 1989.

[9] S. Sriram and E. A. Lee, "Determining the order of processor transactions in statically scheduled multiprocessors," *J. VLSI Signal Process.*, vol. 15, no. 3, pp. 207–220, Mar. 1997.

[10] S. S. Bhattacharyya, S. Sriram, and E. A. Lee, "Optimizing synchronization in multiprocessor DSP systems," *IEEE Trans. Signal Process.*, vol. 45, no. 6, pp. 1605–1618, Jun. 1997.

[11] S. Sriram and S. S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*. New York: Marcel Dekker, 2000.

[12] S. Sriram, "Minimizing Communication and Synchronization Overhead in Multiprocessors for Digital Signal Processing," Ph.D. Dissertation, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, CA, 1995.

[13] R. Reiter, "Scheduling parallel computations," *J. Assoc. Comput. Machinery*, vol. 15, no. 4, pp. 590–599, Oct. 1968.

[14] J. L. Peterson, *Petri Net Theory and Modeling of Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1981.

[15] S. Banerjee, T. Hamada, P. M. Chau, and R. D. Fellman, "Macro pipelining based scheduling on high performance heterogeneous multiprocessor systems," *IEEE Trans. Signal Process.*, vol. 43, no. 6, pp. 1468–1484, Jun. 1995.

[16] D. R. Surma and E. Sha, "Application specific communication scheduling on parallel systems," in *Proc. 8th Int. Conf. Parallel and Distributed Computing Systems*, Sep. 1995, pp. 137–139.

[17] D. R. Surma, S. Tongsima, and E. Sha, "Optimal communication scheduling based on collision graph model," in *Proc. 1996 Int. Conf. Acoustics, Speech and Signal Processing*, vol. VI, May 1996, pp. 3319–3322.

[18] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York: Freeman, 1991.

[19] P. Marwedel and G. Goosens, *Code Generation for Embedded Processors*. New York: Kluwer Academic, 1995.

[20] C. L. McCreary, A. A. Kahn, J. J. Thompson, and M. E. McArdle, "A comparison of heuristics for scheduling DAG's on multiprocessors," in *Proc. Int. Paralel Processing Symp.*, 1994.

[21] G. de Micheli, *Synthesis and Optimization of Digital Circuits*. New York: McGraw Hill, 1994.

[22] R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," *IEEE Trans. Comput.-Aided Design*, 1993.

[23] K. Melhorn and S. Näher, *LEDA A Platform for Combinatorial and Geometric Computing*. Cambridge, U.K.: Cambridge University Press, 1999.

[24] T. Tia, Z. Deng, M. Shankar, M. Storch, J. Sun, L.-C. Wu, and J.-S. Liu, "Probabilistic performance guarantee for real-time tasks with varying computation times," in *Proc. Real-Time Technology and Applications Symp.*, 1995, pp. 164–173.

[25] S. Hua, G. Qu, and S. Bhattacharyya, "Energy reduction technique for multimedia applications with tolerance to deadline misses," *Proc. Design Automation Conf.*, pp. 131–136, Jun. 2003.

[26] G. C. Sih and E. A. Lee, "A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, no. 2, pp. 75–87, Feb. 1993.

[27] J. Teich and T. Blickle, "System-level synthesis using evolutionary algorithms," *J. Design Automat. Embedded Syst.*, vol. 3, no. 1, pp. 23–58, Jan. 1998.

[28] M. Wu and D. Gajski, "Hypertool: A programming aid for message-passing architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 1, no. 3, pp. 101–119, Jul. 1990.

[29] A. Kahn, C. McCreary, J. Thompson, and M. McArdle, "A comparison of multiprocessor scheduling heuristics," in *Proc. 1994 Int. Conf. Parallel Processing*, vol. II, 1994, pp. 243–250.

[30] T. O'Neil and E. Sha, "Retiming synchronous data-flow graphs to reduce execution time," *IEEE Trans. Signal Process.*, vol. 49, no. 10, pp. 2397–2407, Oct. 2001.

[31] B. Kienhuis and E. F. Deprettere, "Modeling stream-based applications using the SBF model of computation," in *Proc. IEEE Workshop on Signal Processing Systems*, Sep. 2001, pp. 385–394.

[32] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete, "Cyclo-static dataflow," *IEEE Trans. Signal Process.*, vol. 44, no. 2, pp. 397–408, Feb. 1996.

[33] P. K. Murthy and E. A. Lee, "Multidimensional synchronous dataflow," *IEEE Trans. Signal Process.*, vol. 50, no. 8, pp. 2064–2079, Aug. 2002.
[34] B. Bhattacharya and S. S. Bhattacharyya, "Parameterized dataflow modeling for DSP systems," *IEEE Trans. Signal Process.*, vol. 43, no. 10, pp. 2408–2421, Oct. 2001.

Mukul Khandelia received the B.Tech. degree in computer science and engineering from the Indian Institute of Technology, Kharagpur, India, and the Master's degree from the Department of Electrical and Computer Engineering at the University of Maryland, College Park, in 2001.

He is currently a Senior R&D Engineer in the Verification group at Synopsys, Inc., Mountain View, CA. His current research interests include hardware/software codesign, constrained randomization and coverage-driven verification.

Neal K. Bambha received B.S. degrees in physics and electrical engineering from Iowa State University, Ames, and the M.S. degree in electrical engineering from Princeton University, Princeton, NJ. He received the Ph.D. degree from the Department of Electrical and Computer Engineering at the University of Maryland, College Park, in 2004.

He is currently a Member of the Technical Staff at the U.S. Army Research Laboratory. His research interests include hardware/software codesign, signal processing, optical interconnects within digital systems, and evolutionary algorithms.

Shuvra S. Bhattacharyya (S'87–M'95) received the B.S. degree from the University of Wisconsin at Madison, and the Ph.D. degree from the University of California at Berkeley.

He is an Associate Professor in the Department of Electrical and Computer Engineering, and the Institute for Advanced Computer Studies (UMIACS) at the University of Maryland, College Park. He is also an Affiliate Associate Professor in the Department of Computer Science. He is coauthor of two books and the author or coauthor of more than 60 refereed technical articles. His research interests include signal processing, embedded software, and hardware/software codesign. He has held industrial positions as a Researcher at the Hitachi America Semiconductor Research Laboratory, and as a Compiler Developer at Kuck & Associates.