This work appeared originally in the *Proceedings of the International Conference on Application Specific Systems, Architectures, and Processors*, pages 276-285, Boston, Massachusetts, July, 2000.
This is an updated and improved version of the aforementioned paper.

Contention-conscious Transaction Ordering in Embedded Multiprocessors[1]

Mukul Khandelia
Dept. of Elec. & Comp. Engr., and
Inst. for Advanced Comp. Studies
Univ. of Maryland, College Park
mukulk@eng.umd.edu

Shuvra S. Bhattacharyya
Dept. of Elec. & Comp. Engr., and
Inst. for Advanced Comp. Studies
Univ. of Maryland, College Park
ssb@eng.umd.edu

Abstract

This paper explores the problem of efficiently ordering interprocessor communication operations in statically-scheduled multiprocessors for iterative dataflow graphs. In most digital signal processing 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 non-negligible, finding an optimal transaction order given a static schedule is an NP-complete problem, and that this intractability holds both under iterative and non-iterative execution. We develop new heuristics for finding efficient transaction orders, and perform an experimental comparison to gauge the performance of these heuristics.

1. 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 discussed in [11, 14, 19]). 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. 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.

1. This research was sponsored by the US National Science Foundation (CAREER, MIP9734275); and the Army Research Laboratory under Contract No. DAAL01-98-K-0075, and the MICRA program.

1.1 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 [4, 8, 9]). 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 [9] — 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 as possible at compile time [9].

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 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 [9], 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 [18] 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 [2, 3].

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 [19]. The inverse of the average iteration period T gives us the *throughput* $T^{\angle 1}$, which is the average number of graph iterations carried out per unit time.

1.2 Terminology and notation

We denote the set of positive integers by Z^+, the set of natural numbers $\{0, 1, 2, \ldots\}$ by $\aleph$, and the number of elements in a finite set S by $|S|$.

With each actor $v \in V$ in an HSDF specification (V, E), we associate an integer $exec(v)$, which denotes the execution time estimate of v, and an integer $proc(v)$, which denotes the processor that v is assigned to in the assignment step. Each edge $(v_i, v_j) \in E$ has a non-negative integer *delay* associated with it, which is denoted by $delay(v_i, v_j)$. These delays represent initial tokens, and specify dependencies between iterations of actors in iterative execution. For example, if the tokens produced by an actor v_i on its kth invocation are consumed by actor v_j on its $(k+2)$th invocation, the edge between v_i and v_j would have a delay of 2.

Every edge (v_i, v_j) induces the precedence constraint

$$start(v_j, k) \geq start(v_i, k \angle delay(v_i, v_j)) + exec(v_i), \tag{1}$$

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

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 $snk(e_i) = src(e_{i+1})$, for $i = 1, 2, \ldots, (n \angle 1)$. We say that the path $(e_1, e_2, \ldots, e_n)$ is *directed from* $src(e_1)$ *to* $snk(e_n)$. 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 $Delay(p)$, is given by

$$Delay(p) = \sum delay(e_i). \tag{2}$$

Each cycle c in a dataflow graph must satisfy $Delay(c) > 0$ to avoid deadlock.

The evolution of a self-timed implementation can be modeled by Sriram's *IPC graph* model [18]. 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 Figure 1, and the corresponding IPC graph is illustrated in Figure 3.

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 [16], also known as the *timed marked graph* model in Petri net theory [15], 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

$$MCM(G_{ipc}) = \max_{\text{cycle C in } G_{ipc}} \left\{ \frac{\sum_{v \in C} exec(v)}{Delay(C)} \right\} . \tag{3}$$

The quotient in (3) is referred to as the *cycle mean* of the associated cycle C. (4)

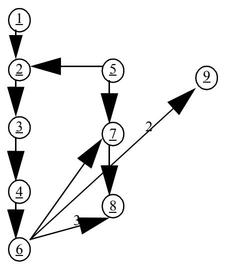
A similar data structure that is useful in analyzing OT implementations is Sriram's *ordered transaction graph* model [19]. Given an ordering $O = \{o_1, o_2, \ldots o_p\}$ for the communication actors in an IPC graph $G_{ipc} = (V_{ipc}, E_{ipc})$, the corresponding ordered transaction graph $\Gamma(G_{ipc}, O)$ is defined as the directed graph, $G_{OT} = (V_{OT}, E_{OT})$, where $V_{OT} = V_{ipc}$, $E_{OT} = E_{ipc} \cup E_O$,

$$E_O = \{(o_p, o_1), (o_1, o_2), (o_2, o_3), \ldots, (o_{p \angle 1}, o_p)\} , \tag{5}$$

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

2. Previous Work

In [18], Sriram and Lee discuss some of the advantages and disadvantages of the OT strategy compared to the ST strategy — in particular, lower synchronization and arbitration costs for the IPC mechanism at the expense of some run-time flexibility. 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



Self-Timed Schedule
Proc 1: (1, 2, 3, 4, 6)
Proc 2: (5, 7, 8)
Proc 3: (9)

Figure 1. An example of an application graph, and an associated self-timed schedule.

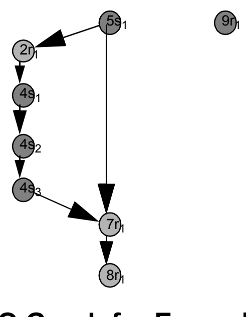


Figure 2. TPO Graph for Example 1.

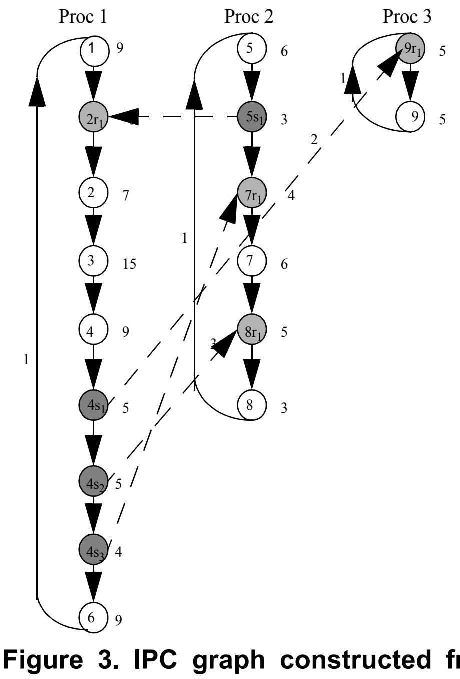


Figure 3. IPC graph constructed fro the example of Figure 1.

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 optimum.

However, in this paper, we show 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. Thus, under nonzero IPC costs, we must resort to heuristics for efficient solutions. Furthermore, the polynomial-time BFB algorithm is no longer optimal, and alternative techniques that account for IPC costs are preferable.

3. 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 MCM of the resulting ordered transaction graph is minimized (so that throughput is maximized). As mentioned in Section 2, 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 non-iterative implementation, which is a more restricted (easier) problem, is NP-complete. To convert an iterative IPC graph to a non-iterative 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 $G_{ipc} = (V, E)$, the associated *non-iterative inter-processor communication* (*NIPC*) *graph* is defined as $G_{nipc} = (V, E_{nipc})$, where $E_{nipc} = \{e|(e \in E \textbf{ and } (delay(e) = 0))\}$.

Definition 2: Given an NIPC graph $G_{nipc} = (V, E_{nipc})$, and an ordering O of the communication actors, the corresponding *non-iterative ordered transaction (NOT) graph* $G_{NOT} = \Pi(G_{nipc}, O)$ is defined as $G_{NOT} = (V_{NOT}, E_{NOT})$, where $V_{NOT} = V$, $E_{NOT} = (E_{nipc} \cup E_O) \angle \{(o_p, o_1)\}$, and E_O is as defined in (5).

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

To show that non-iterative transaction ordering is NP hard, we have derived a reduction from the *sequencing with release times and deadlines* (*SRTD*) problem, which is known to be NP-complete [6].

The following result, established in [7] based on a reduction from the SRTD problem, tells us that optimal transaction ordering is intractable even in a non-iterative context.

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

In multiprocessor implementation of reactive applications, such as those that arise in DSP, we are typically interested in the performance under iterative execution. The iterative transaction ordering problem pertains to this context.

Definition 4: 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 $G_{OT} = \Gamma(G_{ipc}, O)$ satisfies $MCM(G_{OT}) \leq k$?

The proof of Theorem 1 in [7] can be extended to establish the intractability of iterative transaction ordering (again using a reduction from the SRTD problem — details of this reduction can also be found in [7]). This yields the following result.

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

Given the intractability of computing optimal transaction orders, we must resort to heuristics to achieve scalable compilation techniques. In the next three sections, we develop a number of effective heuristic approaches for deriving efficient transaction orders, and in Section 7, we present an experimental analysis of these approaches.

4. 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 is similar to the pattern of transactions in the associated self-timed schedule. However, we have demonstrated that in the presence of non-zero IPC, the OT method can, in fact, perform significantly better than the ST method [7], 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 MCMs are more favorable under such an evaluation are scheduled earlier in the transaction order.

More specifically, a partial order of the communication (send and receive) actors is first computed from the IPC graph G_{ipc}: the *transaction partial order (TPO) graph* G_{TPO} is computed by first deleting all edges in G_{ipc} that have delays of one or more, and then deleting all of the computation actors.

Example 1: The transaction partial order graph computed from the IPC graph of Figure 3 is illustrated in Figure 2. Notice that all the dependencies imposed by the IPC graph are retained in G_{TPO} but only for the communication actors.

The 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}. The process is repeated until all communication actors have been scheduled into a linear ordering. A complete pseudocode specification of the TPO heuristic can be found in [7].

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.

Example 2: When we apply the TPO heuristic to the IPC graph of Figure 3, the schedule that we obtain is illustrated by the Gantt chart of Figure 4 The corresponding OT graph is illustrated in Figure 5.

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 $O(P|V|^2|E_{OT}|)$ 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 to 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_{TPO} has a maximum of P communication actors that can be scheduled last in the transaction order, the modified heuristic has a complexity of $O(P^2|V|^2|E|)$.

5. Genetic algorithm for transaction scheduling

Since the transaction ordering problem is intractable, we are unable to efficiently find optimal

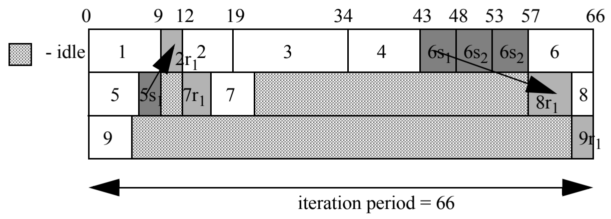


Figure 4. Gantt Chart for the OT schedule that results when the TPO heuristic is applied to the example of Figure 3.

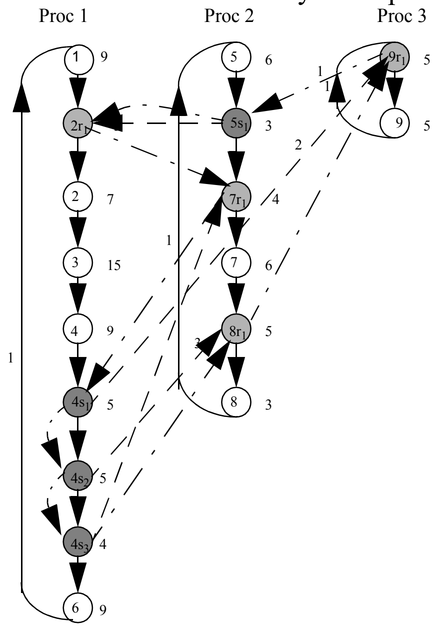


Figure 5. OT Graph obtained by applying TPO heuristic in Example 2.

transaction orders on a consistent basis. We have implemented a branch and bound strategy to explore the search space comprehensively, but this technique 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 [10], 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 [5]. 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 [5]. 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.

For details on the underlying GA concepts, we refer the reader to [1]. 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_{TPO} corresponding to the transaction partial order graph G_{TPO}. The recombination step takes $O(|V|^2)$ time, and the evaluation step takes $O(|V||E_{OT}|)$ time. The overall complexity of each iteration is also influenced by the population size and the overhead involved in generating random numbers.

Further details on this GA technique for transaction ordering can be found in [7].

6. Dynamic reordering

Once we obtain a transaction order (e.g., using the TPO heuristic or the GA approach defined in Section 5), 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 OBDD's (Ordered Binary Decision Diagrams) [13]. We have implemented an adaptation to ordered transaction scheduling, called *dynamic transaction reordering* (*DTR*), of the *Sifting Algorithm* introduced by Rudell [17], and have observed that from DTR, we consistently obtain improvements in the iteration period, regardless of the method used to find the transaction order.

7. 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 4-6 were implemented in C/C++ using the *LEDA* [12] framework for fundamental graph-theoretic data structures and algorithms. The benchmarks are standard DSP applications that have been scheduled using the classic *HLFET* algorithm [6].

The IPC graphs are fairly complicated, ranging from between 50-150 nodes, and the numbers of processors involved range from 2 to 8. The examples *fft1, fft2,* and *fft3* result from three representative schedules for Fast Fourier Transforms based on examples given in [11]; *karp10* is a music synthesis application based on the Karplus Strong algorithm in 10 voices; and *qmf4* is a 4 channel multi-resolution QMF filter bank for signal compression.

In the simulation of the ST schedule, we ignore the overhead of synchronization so as to give us a worst-case comparison with the OT schedule. In practice, of course, synchronization has nonzero cost, and thus, depending on the actual synchronization overhead in the target architecture, the benefit of the OT schedules examined will be even more that what the results here demonstrate. Thus, our analysis in this section gives a lower bound on the improvement we can expect using the OT implementation strategy in conjunction with our proposed transaction ordering techniques.

Table 1 compares the performance (iteration period) of the ST and the OT schedules. Here, the average iteration period (T_{OT}) of the OT schedule is obtained by taking the best performance using the algorithms proposed in Sections 4-6, and T_{ST} denotes the average iteration period of the corresponding ST schedule. In each of the cases, we see that the OT strategy can outperform the ST strategy, and that this holds even though we are ignoring synchronization costs, which gives us a very optimistic view of the performance under ST execution.

Table 1. Comparison of ST and OT schedules.

Application	T_{ST}	T_{OT}
fft1	263	245
fft2	312	300
fft3	263	245
karp10	312	308
qmf4	147	140

Table 2 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. 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 are demonstrated is the enhanced version where the delays are inserted beforehand. In all cases, the TPO heuristic performs better than the BFB technique — especially for *fft1* and *fft3* — and the heuristic that combines the TPO heuristic and DTR performs best (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.

Table 2. Comparison of algorithms.

Application	T_{random}	T_{BFB}	T_{TPO}	T_{GA}	$T_{TPO+DTR}$
fft1	392	280	245	255	245
fft2	395	340	320	300	300
fft3	390	300	255	255	245
karp10	482	312	309	308	309
qmf4	196	148	145	140	145

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 results are shown in Table 3.

8. Conclusions

We have demonstrated that in the presence of accurate estimates for actor execution times, 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 ordered transaction implementation offers less run-time flexibility due to a fixed ordering of communication operations. We have also shown that in the presence of non-zero 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, and a genetic algorithm for exploiting extra compile time when generating final implementations. Useful directions for further work include integrating transaction ordering considerations into the scheduling process, and 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.

9. References

[1] T. Back, U. Hammel, and H-P Schwefel, "Evolutionary computation: Comments on the history and current state," *IEEE Transactions on Evolutionary Computation*, 1(1):3-17, 1997.

[2] S. S. Bhattacharyya, S. Sriram and E. A. Lee, "Minimizing Synchronization Overhead in Statically Scheduled Multiprocessor Systems," *Proceedings of the International Conference on Application Specific Array Processors*, July, 1995.

[3] S. S. Bhattacharyya, S. Sriram and E. A. Lee. "Latency-constrained Resynchronization for Multiprocessor DSP Implementation," *Proceedings of the International Conference on Application Specific Systems, Architectures and Processors*, August, 1996.

[4] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems," *International Journal of Computer Simulation*, Vol. 4, pp. 155-182, Jan. 1994.

[5] B. R. Fox and M. B. McMahon, "Genetic Operators for Sequencing Problems," G. Rawlins, *Foundations of Genetic Algorithms*, Morgan Kaufman Publishers Inc., 1991.

[6] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, New York, 1999.

[7] M. Khandelia and S. S. Bhattacharyya, *Contention-conscious transaction ordering in embedded multiprocessor systems*. Technical Report UMIACS-TR-2000-09, Institute for Advanced Computer Studies, University of Maryland at College Park, March 2000.

[8] S. Y. Kung and P. S. Lewis and S. C. Lo, "Performance Analysis and Optimization of VLSI Dataflow Arrays" *Journal of Parallel and Distributed Computing*, pp. 592-618, 1987.

[9] E. A. Lee and S. Ha, "Scheduling strategies for Multiprocessor real-time DSP", *Proceeding of the Globecom Conference,* Dallas, Texas, pp. 1279-1283, Nov. 1989.

[10] P. Marwedel and G. Goossens, *Code Generation for Embedded Processors*, Kluwer Academic Publishers, 1995.

[11] C. L. McCreary, A. A. Kahn, J. J. Thompson, M. E. McArdle, "A Comparison of Heuristics for Scheduling DAGS on Multiprocessors," *International Parallel Processing Symposium*, 1994.

[12] K. Mehlhorn and S. Naher and M. Seel and C. Uhrig, *The LEDA User Manual*, Max-Planck--Institut for Informatik, Saarbrucken, Germany, Version 3.7.

[13] G. De. Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 1994.

[14] K. Parhi and D. G. Messerschmitt, "Static Rate Optimal Scheduling of Iterative Dataflow Programs via Optimum Folding," *IEEE Transactions on Computers*, Vol. 40, No. 2, pp. 178-194, Feb. 1991.

[15] J. L. Peterson, *Petri Net Theory and Modelling of Systems*, Prentice-Hall Inc., Englewoods Cliffs, NJ, 1981.

[16] R. Reiter, "Scheduling Parallel Computations," *Journal of the Association for Computing Machinery*, Vol. 15, No. 4, pp. 590-599, Oct. 1968.

[17] R. Rudell, "Dynamic Variable Ordering for Ordered Binary Decision Diagrams," *IEEE Trans. on CAD*, 1993.

[18] S. Sriram and E. A. Lee, "Determining the Order of Processor Transactions in Statically Scheduled Multiprocessors," *Journal of VLSI Signal Processing*, pp. 207-220, 1997.

[19] S. Sriram and S. S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*, Marcel Dekker Inc., 2000.

Table 3. Results when GA is applied to the TPO heuristic in conjunction with DTR.

Application	$T_{\text{TPO+DTR+GA}}$
fft1	245
fft2	295
fft3	245
karp10	305
qmf4	140