



mance of these heuristics.

We also propose a new method called the software ordered transaction (SOT) schedule that takes advantages of the properties of both the self-timed and the ordered transaction schedule but does not require hardware support. By adding extra synchronization edges derived from the transaction order, we show that it is possible to improve the throughput of self-timed schedules. For non-zero IPC costs, we prove that finding an optimal software transaction order is NP hard and suggest some ways of finding efficient software transaction orders. Practical DSP applications are used to demonstrate the efficacy of our techniques.

COMMUNICATION SCHEDULING IN EMBEDDED MULTIPROCESSOR  
SYSTEMS

by

Mukul Khandelia

Thesis submitted to the Faculty of the Graduate School of the  
University of Maryland, College Park in partial fulfillment  
of the requirements for the degree of  
Master of Science  
2001

Advisory Committee:

Dr. Shuvra S. Bhattacharyya, Chair  
Dr. Donald Yeung  
Dr. Rajeev Barua

## DEDICATION

For all my friends in College Park, Maryland.

## ACKNOWLEDGEMENTS

I am indebted to my advisor, Dr. Shuvra Bhattacharyya for all his guidance throughout my thesis work and also for providing me the necessary motivation. I would also like to thank James Bonanno for the excellent work he did in writing the simulator. Lastly I would like to thank my lab mates, Vida Kianzad, Nitin Chandra-choodan, Sumit Lohani and Neal Kumar Bambha for helping me at various stages of my thesis.

## TABLE OF CONTENTS

LIST OF FIGURES	iv
Chapter 1. Introduction	1
1.1. Synchronous Dataflow Graphs.....	3
1.2. Overview .....	5
Chapter 2. Scheduling Dataflow Graphs	6
2.1. Execution Time Estimates.....	7
2.2. Throughput Computation .....	8
2.3. Scheduling Strategies .....	9
2.4. Fully-static Scheduling.....	10
2.5. Self-timed Strategy.....	11
2.6. Ordered Transaction Strategy.....	13
Chapter 3. Notation and Terminology	16
Chapter 4. Previous Work	22
Chapter 5. Comparison of Self-timed and Ordered Transaction Strategies	28
5.1. Improved Throughput.....	29
5.2. Better Predictability.....	30
Chapter 6. Finding Optimal Transaction Orders	34
6.1. The Non-iterative Ordered Transaction Ordering Problem.....	34
6.2. NP-completeness of Ordered Transaction Problem .....	40
6.3. The Transaction Partial Order Heuristic.....	43
6.4. Branch and Bound Strategy.....	51
6.5. Genetic Algorithm for Transaction Scheduling .....	52
6.6. Dynamic Reordering .....	54
6.7. Simulator .....	55
6.8. Results .....	58
Chapter 7. Software Ordered Transactions	62
7.1. Transaction Order Removal Heuristic.....	72
7.2. Genetic Algorithm for Software Ordered Transactions .....	76
7.3. Results .....	77
Chapter 8. Conclusions and Future Work	81



## LIST OF FIGURES

Figure 1. An example of dataflow graph .....	4
Figure 2. Fully-static schedule on five processors. (a) HSDFG “G”. (b) Static schedule. (c) Fully-static execution.....	10
Figure 3. Self-timed schedule.....	11
Figure 4. Schedule evolution when transaction order in Example 4 is enforced.....	13
Figure 5. An example of application graph and an associated self-timed schedule. The numbers on edges (6,8) and (6,9) denote non-zero delays.....	17
Figure 6. IPC graph constructed from application graph of Figure 5.....	18
Figure 7. Example of OT graph.....	19
Figure 8. An IPC graph with feedforward edge: (a) original graph, (b) imposing bounded buffers.....	23
Figure 9. $x_2$ is an example of a redundant synchronization edge.....	24
Figure 10. An example of resynchronization.....	25
Figure 11. Gantt chart for ST schedule in Example 8.....	29
Figure 12. Gantt chart for OT schedule in Example 8.....	30
Figure 13. IPC graph for Example 9.....	30
Figure 14. Gantt chart for ST schedule when $\text{exec}(1) = 21$ .....	31
Figure 15. Gantt chart for OT schedule when $\text{exec}(1) = 19$ .....	31
Figure 16. NIPC graph corresponding to IPC graph in Figure 6.....	35
Figure 17. NOT graph constructed in Example 11.....	39
Figure 18. Constructed OT graph in Figure 12.....	42
Figure 19. TPO graph in Example 13.....	44
Figure 20. Function to choose next communication actor in transaction order.....	46
Figure 21. Function to initialize data structures called by TPO function.....	46
Figure 22. Pseudocode for TPO heuristic.....	47
Figure 23. OT graph obtained by applying TPO heuristic in Example 14.....	48
Figure 24. Pseudocode for Branch and Bound strategy.....	50
Figure 25. Pseudocode for GA approach to transaction ordering.....	52
Figure 26. Pseudocode for Dynamic Transaction Reordering.....	54
Figure 27. NSTO graph constructed in Example 17.....	66
Figure 28. Constructed STO graph in Example 14.....	70
Figure 29. Pseudocode for Transaction Order Removal algorithm.....	72
Figure 30. Pseudocode for GA approach to software ordered transactions.....	75
Figure 31. Comparison without synchronization costs for benchmark qmf4.....	75
Figure 32. Comparison with synchronization costs for benchmark qmf4.....	76

Figure 33. Comparison without synchronization costs for application video_coder benchmark.....	77
Figure 34. Comparison with synchronization costs for video_coder benchmark.....	77

## Chapter 1. Introduction

In this thesis, we shall consider techniques to reduce communication costs in statically-scheduled multiprocessors for iterative dataflow specifications. The hardware platform that we consider is a bus-based architecture in which a group of processors communicate by means of a single shared bus. 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.

It provides a cheap and efficient platform to perform tasks which ordinarily a single processor could not handle. This kind of architecture is particularly well suited for embedded applications in DSP (Digital Signal Processing) that have a simple control structure and are computationally intensive. Many of these applications in signal and image processing have real-time performance requirements.

The programmable processor that we speak about is a general one. It can consist of an integrated circuit such as a RISC or a Digital Signal Processor. The processors sharing the bus need not be homogeneous but can consist of different types of heterogeneous programmable elements coupled together. This is typically the case in embedded applications where a variety of different tasks have to be performed each having different requirements. These applications have extensive use in the telecommunications industry where programmability is paramount. Examples are home gateway systems that require recognizing different audio/video formats

and doing compression/decompression of data streams. In these systems, cost plays a deciding factor in choosing the hardware involved.

Mapping the application on this kind of an architecture is a hardware/software co-design problem and is not trivial. Since in embedded applications, the designer has to do a one-time job of programming the system, it is necessary to develop a design methodology that will give the designer tools to program the system in an efficient manner.

These applications are different from general-purpose application in the sense that significant amount of information is known at compile-time and much of this information can be processed at compile-time to streamline the design methodology. Communication costs have added importance in these kinds of architectures since the only mode of computation is the shared bus which if not used efficiently, may significantly decrease the performance of such embedded multiprocessor systems. Our focus in this thesis will be to examine some of the techniques that have been researched in this context, and to propose new methodologies to further the design techniques.

Our objective is to reduce the overall IPC (inter-processor communication) 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).

In the area of digital signal processing (DSP), dataflow is widely recognized as a natural model for specifying DSP applications. In dataflow, a program is represented as a directed graph, called a *dataflow graph*, in which vertices, called *actors*, represent computations and edges represent FIFO channels, (also called *buffers*). These channels queue data values, in the form of *tokens*, which are passed from the output of one actor to the input of another. When an actor is executed (fired), it consumes a certain number of tokens from its inputs, and produces a certain number of tokens at its outputs.

### **1.1. Synchronous dataflow graphs**

In application-specific systems, it is easier to identify and exploit parallelism than the general-purpose computation domain; the challenge is to meet the stringent requirements of the system. Our study of multiprocessor implementations in this thesis is in the context of Synchronous Dataflow (SDF) specifications [21]. In SDF, 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. The arcs in the SDF graph may contain initial tokens, which we also refer to as *delays*. Arcs with delays can be interpreted as data dependencies across iterations of the graph.

**Example 1:** Figure 1 shows an example of a dataflow graph. Numbers beside edges indicate non-zero delays. There exists a delay of 1 between actor C and actor D.

The model that we will use is an extension of the SDF model called the (Homogeneous Synchronous Dataflow) HSDF model which 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. It has been shown that every SDF graph can be converted to an equivalent HSDF graph [33] that has the same behavior and functionality as the original SDF graph although the number of actors in the HSDF graph may be exponential to the number of actors in the corresponding SDF graph.

Dataflow models are a very useful specification mechanism for signal processing systems since they capture the intuitive expressivity of block diagrams, flow charts and signal flow descriptions. SDF models have proven to be very popular since they are able to satisfy the goal of both simulation of the algorithm at the functional or behavioral level, and for synthesis from such a high level specification to a software or hardware description. Many extensions have been made to the basic

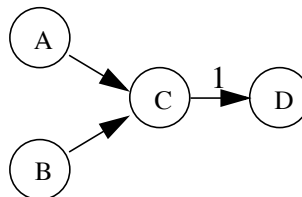


Figure 1. An example of a dataflow graph.

SDF model to allow control constructs so that data dependent control can be expressed in such controls [19, 11, 5].

## **1.2. Overview**

This section gives an overview of how the rest of the thesis is organized. In Chapter 2, we will discuss some of the scheduling strategies for actors in these multiprocessor DSP systems. We will describe the fully-static, self-timed and the ordered transaction strategies. In Chapter 3, we will lay down the formal notation and terminology that will be followed in the rest of the thesis. In Chapter 4, we will discuss some of the previous work that has been studied in the context of ordered transactions. In Chapter 5, we will compare self-timed schedules to ordered transaction schedules and demonstrate some of the advantages of the ordered transaction strategy. In Chapter 6, we will show that finding the optimal transaction order is an intractable problem and suggest efficient algorithms to overcome this. In Chapter 7, we will discuss the software ordered transaction method, a new methodology that employs software techniques for transaction ordering to improve the throughput of self-timed schedules. Finally, in Chapter 8 we will discuss some of the conclusions we have arrived at and directions for future research.

## Chapter 2. Scheduling dataflow graphs

In this thesis, we assume that we are given a schedule for the associated application graph. This schedule can be obtained using a multitude of methods available [12, 18, 20, 21]. Most of the research on scheduling HSDFG's has focussed on minimizing the schedule makespan, which is the time required to execute all actors in the HSDFG once. One of the most fundamental classical approaches is *list scheduling* in which a priority list of actors is constructed; a global time clock is maintained; and each actor is eventually mapped on some processor such that the time intervals for two distinct actors assigned to the same processor do not overlap [16]. One of the most widely used algorithms and popular list scheduling techniques is the HLFET (highest level first with estimated times) [17] in which the priority list is constructed by sorting the actors in decreasing order of their *levels*. The level of an actor in an acyclic graph is defined as the length of the longest path to the actor. The length of a path is taken to be the sum of the execution times of the actors on the path. Other more advanced techniques include Dynamic Level Scheduling by Sih [32], where the priority list is continuously updated while constructing the schedule; the Linear Clustering Algorithm proposed by Kim and Browne, which operates by incrementally constructing groupings, called clusters, of actors that are to be executed on the same processor; and Pipelined scheduling [10] algorithms, which attempt to effi-

ciently partition a task graph into stages, assign groups of processors into pipeline stages, and construct schedules for each pipeline stage.

## 2.1. Execution time estimates

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 thesis, 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.

A very simple model for actors with variable execution times is to assign to each actor an execution time that is a random variable (r.v.) with a discrete probability distribution function (p.d.f); successive invocations of each actor are assumed statistically independent, execution times of different actors are assumed independent, and the statistics of the random execution times are assumed to be time invariant. Thus, for example, an actor  $A$  could have execution time  $t_1$  with probability  $p$  and execution time  $t_2$  with probability  $(1 - p)$ . Although this model is too simple to capture many real scenarios, it does model an actor such as a data-dependent conditional branch. Dataflow graphs where actors have random execution times

have been studied by Olsder [27] who shows that the behavior of such a system can be described by a discrete-time Markov chain. The problem with such an exact analysis, however, is the very large state space that can result for the Markov chain. The upper bound on the size is exponential in the number of vertices. Therefore this approach has limited use in determining effects of varying execution times.

## 2.2. Throughput computation

The metric of interest to us 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. 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. In acyclic graphs, the iteration period is computed by calculating the makespan (longest path) from the source actors to the sink actors in the schedule. In cyclic dataflow graphs, the average iteration period is computed by calculating the Maximum Cycle Mean (MCM) of the graph, which can be computed using efficient polynomial-time algorithms [13] The MCM is the minimum achievable iteration period. We shall later define in more formal terms what the MCM is. However, for graphs, where communication costs are not negligible, there is no known method to compute the iteration period using a polynomial-time algorithm. The only known way to determine  $T$  is to simulate the shared-bus system for a large period of time and calculate the average. When contention is resolved in a deterministic manner, Bambha [3] has shown that the period can be computed by determining the

*period graph.*

### 2.3. Scheduling strategies

Scheduling can be divided into three steps [22] — 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 [22]. We shall restrict ourselves to the case when the first two steps are carried out beforehand using any of the techniques mentioned above

In relation to the scheduling taxonomy of Lee and Ha [22], there are three general strategies with which we are primarily concerned with in our analysis. These are the *fully-static (FS)* strategy, the *self-timed (ST)* strategy and the *ordered transaction (OT)* strategy. Let  $T_{FS}$ ,  $T_{OT}$ , and  $T_{ST}$  denote the average iteration periods of the fully-static, ordered transaction and self-timed schedules respectively.

### 2.4. Fully-static scheduling

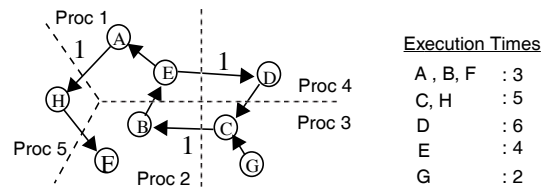
In fully-static scheduling, all the three scheduling steps mentioned above are computed at compile-time.

**Example 2:** Figure 2a shows an example application graph. and the execution time estimates are mentioned on the side. Figure 2b shows the static schedule when the application graph is scheduled and Figure 2c demonstrates how the fully-static

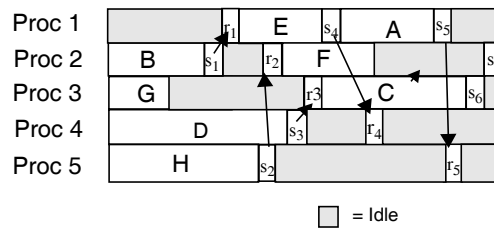
schedule looks when a processor executes it. The horizontal axis denotes the time axis and the vertical axis the processors. We see that the iteration period obtained when IPC costs are ignored is equal to 11.

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.

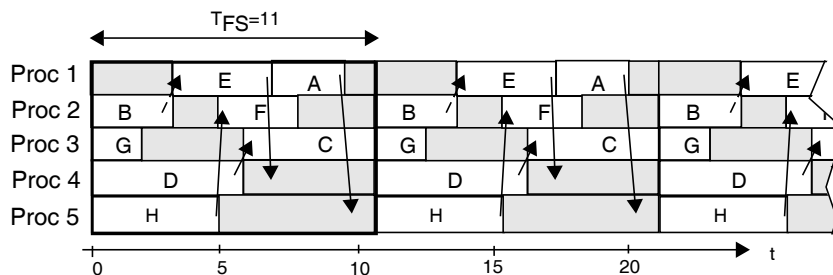
The FS strategy does not work well in cases when there is a high degree of variabil-



(a) HSDFG "G"



(b) Static schedule



(c) Fully-static execution

Figure 2. Fully-static schedule on five processors. s and r denote send and receive respectively.

ity in the execution time of actors such as when there are cache misses or data dependent loops. However it has significantly lower communication costs since all the inter-processor communication is determined beforehand and no run-time synchronization is needed to prevent buffer underflow and overflow.

## 2.5. Self-timed strategy

On the other hand, in the self-timed schedule, 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. Conceptually, the processor sending data writes data into a FIFO buffer, and blocks when the buffer is full; the receiver on the other hand blocks when the buffer it reads from is empty. Thus, the flow control is performed at run-time.

**Example 3:** Figure 3 shows the self-timed schedule when the application graph of Figure 2a is carried out in a self-timed manner. Two iterations of the schedule are

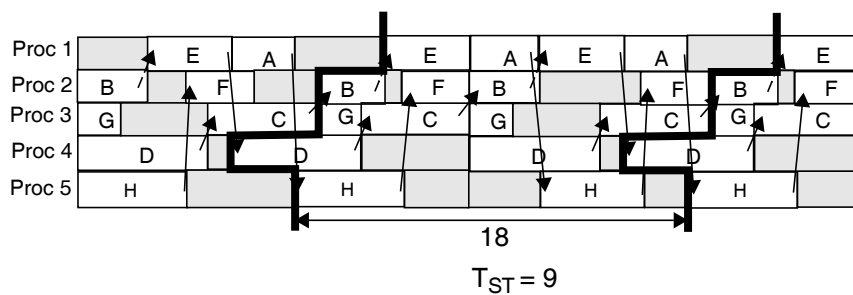


Figure 3. Self-timed schedule.

carried out in 18 time units when synchronization costs are ignored, which gives an iteration period of 9. When the shown self-timed schedule settles down, it spans a period repeating period of 18 time units. It has been shown that eventually any ST schedule will settle down into a periodic repeating pattern provided that contention of shared resources is resolved deterministically [3]. This repeating pattern can be exponential in the number of delays that are present in the critical paths of the schedule [33]. The schedule of the self-timed period is similar to the firing times of timed marked graphs [1] that have an asymptotic period. The settling-down time is termed as the transient and the *transient* itself can be exponential in size. 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 ST method provides more flexibility and can withstand variability in execution times of actors since it performs run-time synchronization but it has higher IPC costs to guarantee correct execution order. The self-timed method also simplifies the job of the scheduler since it does not have to perform a detailed timing analysis to find exacting firing times.

## **2.6. Ordered transaction strategy**

The *ordered transaction (OT)* method [20, 34] 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. For a shared-bus implementation, this translates into determining the sequence of shared memory accesses at compile-time and enforcing this pre-determined order at run-time. This strategy, therefore, involves no run-time arbitration. When a processor obtains the bus, it performs the necessary shared memory transaction, and releases the bus; the bus is then granted to the next processor in the ordered list. The linear ordering imposed is called the *transaction order* of the associated multiprocessor implementation.

**Example 4:** Figure 4 shows the schedule evolution when the transaction order  $r_1, s_2, r_2, s_3, r_3, s_4, r_4, s_5, r_5, s_6$ , is enforced on the shared bus at run-time. The corresponding iteration period is equal to 10.

When IPC costs are ignored, we observe that the ST method performs better than the OT method, which in turn performs better than the FS method. This is because the ST schedule imposes the least number of constraints while the FS method is very rigid and the OT method lies in between. We will later show that when IPC costs are non-negligible, which is frequently the case, then the OT sched-

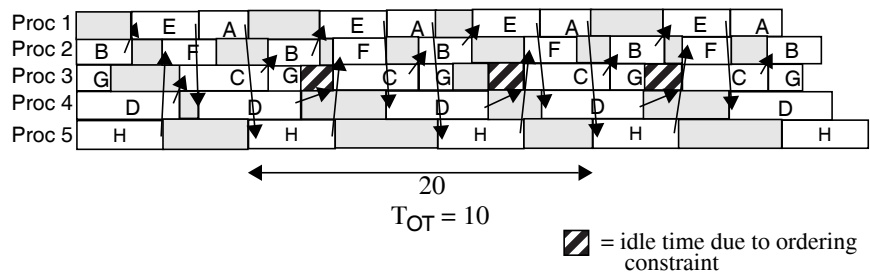


Figure 4. Schedule evolution when the transaction order in Example 4 is enforced.

ule might perform better than the ST method if the transaction order is chosen carefully. We will also elaborate on some of the other useful properties of the Ordered Transaction method such as better predictability and easier compile-time analysis.

The FS and OT strategies have significantly lower overall IPC cost than the ST method since all of the sequencing decisions associated with communication are made at compile time. Architectural support for ordered transaction and related implementation styles is not new and have been developed in the OMA [33] and Raw architectures [38]. A Raw architecture replaces a superscalar bus architecture with a switched interconnect. A Raw chip consists of an array of a number of tiles. Each tile is identical to all the others and contains memory units, function units, and a switch that controls the wires connecting the tile with adjacent ones. The switch is integrated directly into the processor pipeline to support single cycle message injection and receive operations.

Using control logic, it is possible for the compiler to program the switches on all the tiles and change the switch schedule when a different program is loaded. In the ordered transaction method, the shared resource is the bus while in the Raw architecture, the shared resource are the wires which interconnect the tiles on the chip.

## Chapter 3. Notation and terminology

In this chapter, we shall give a brief description of the formal terms and definitions used in the rest of the thesis.

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

In an HSDF specification  $(V, E)$ , with each actor  $v \in V$ , 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  $k$ th 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

$$start(v_j, k) \geq start(v_i, k - delay(v_i, v_j)) + exec(v_i), \quad (1)$$

where  $start(x, k) \in Z^+$  denotes the starting time of the  $k$ th 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  $(e_1, e_2, \dots, e_n)$ , where each  $e_i$  is in  $E$ , and  $snk(e_i) = src(e_{i+1})$ , for  $i = 1, 2, \dots, (n - 1)$ . We say that the path  $(e_1, e_2, \dots, e_n)$  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  $p = (e_1, e_2, \dots, e_n)$ , the *path delay* of  $p$ , denoted  $Delay(p)$ , is given by

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

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

By a subgraph of  $(V, E)$ , we mean the directed graph formed by any  $V' \subseteq V$  together with the set of edges  $\{e \in E \mid src(e), snk(e) \in V'\}$ . We denote the subgraph associated with the vertex-subset  $V'$  by  $subgraph(V')$ .

We say that  $(V, E)$  is *strongly connected* if for each pair of distinct vertices  $x, y$ , there is a path directed from  $x$  to  $y$  and there is a path directed from  $y$  to  $x$ . We say that a subset  $V' \subseteq V$  is strongly connected if  $subgraph(V')$  is strongly connected. A *strongly connected component (SCC)* of  $(V, E)$  is a strongly connected subset  $V' \subseteq V$  such that no strongly connected subset of  $V$  properly contains  $V'$ . If  $V'$  is an SCC, then when there is no ambiguity, we may also say that  $subgraph(V')$  is an SCC. If  $C_1$  and  $C_2$  are distinct SCCs in  $(V, E)$ , we say that  $C_1$  is a predecessor SCC of  $C_2$  if there is an edge directed from some vertex in  $C_1$  to some vertex in  $C_2$ ;  $C_1$  is a successor SCC of  $C_2$  if  $C_2$  is a predecessor SCC of  $C_1$ . A SCC is a source SCC if it has no predecessor SCC; and a SCC is a sink SCC if it has no successor SCC. An edge  $e$  is a *feedforward* edge of  $(V, E)$  if it is not contained in an SCC, or equivalently, if it is not contained in a cycle; an edge that is contained in at least one cycle is called a *feedback* edge.

The evolution of a self-timed implementation can be modeled by Sriram's

*IPC graph* model [35]. 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 5, and the corresponding IPC graph is illustrated in Figure 6.

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 [30], also

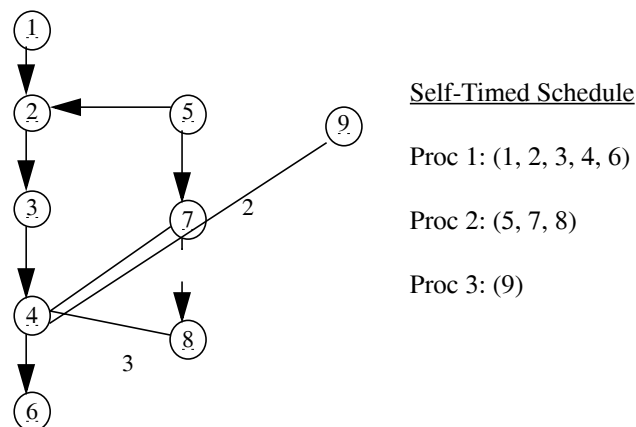


Figure 5. An example of an application graph and an associated self-timed schedule. The numbers on the edges  $(6, 8)$  and  $(6, 9)$  denote nonzero delays.

known as the *timed marked graph* model in Petri net theory [29], 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

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

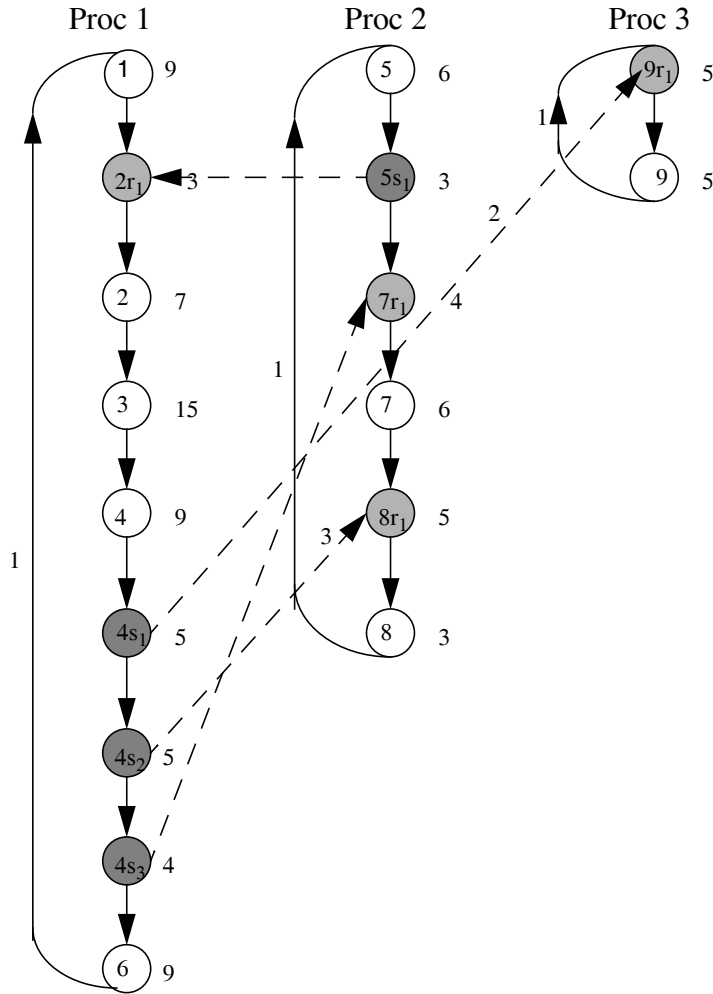


Figure 6. IPC graph constructed from application graph of Figure 5. Numbers besides communication actors denote communication costs.

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 [35]. Given an ordering  $O = \{o_1, o_2, \dots, o_p\}$  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  $V_{OT} = V_{ipc}$ ,  $E_{OT} = E_{ipc} \cup E_O$ , where

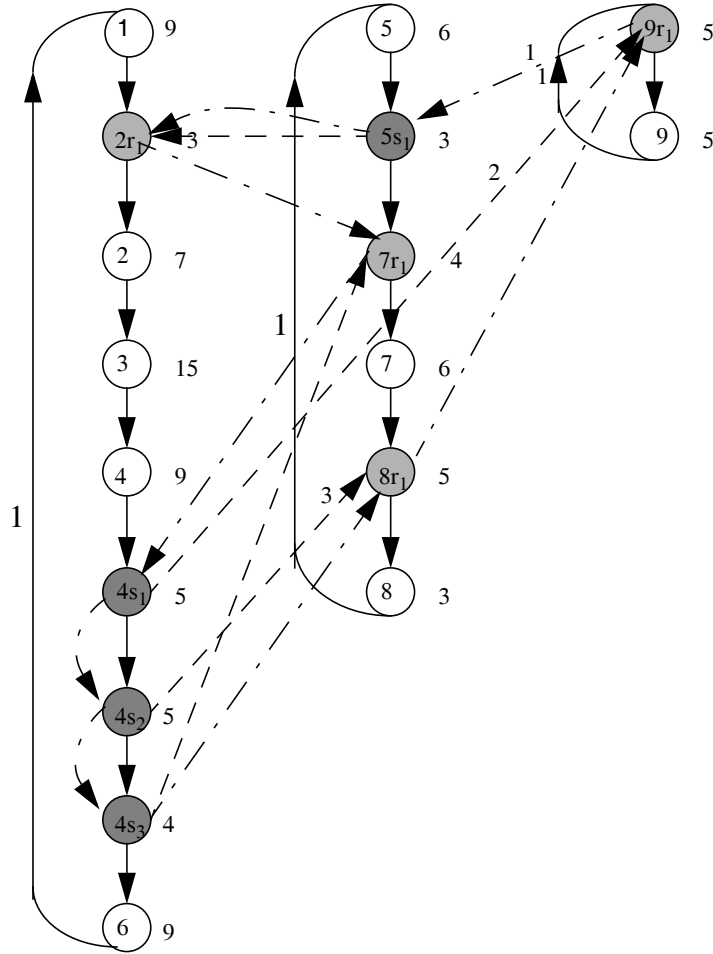


Figure 7. Example of OT graph for ordering  $(5s_1, 2r_1, 7r_1, 4s_1, 4s_2, 4s_3, 8r_1, 9r_1)$ .

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

$delay(o_i, o_{i+1}) = 0$  for  $1 \leq i < p$ , 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. An example of an OT graph is shown in Figure 7.

Note that IPC edges represent both buffer activity and synchronization constraints. Before each buffer is read or written to, it must be checked for buffer underflow or overflow. The synchronization constraints ensure that the buffer is not empty or full. However, the ordered transaction edges only represent synchronization and do not imply buffer activity. Similarly when we talk about removing or adding edges, we refer to only the synchronization edges which can be removed or replaced.

## Chapter 4. Previous work

In [34, 35], Sriram and Lee discuss some of the advantages and disadvantages of the OT strategy compared to the ST strategy in the context of zero IPC costs — in particular, lower synchronization and arbitration costs for the IPC mechanism at the expense of some run-time flexibility. Let  $T_{FS}$ ,  $T_{ST}$  and  $T_{OT}$  denote the iteration periods of the fully-timed, self-time and ordered transaction strategies respectively. They also develop a method to compute an optimum transaction order for the ordered transaction strategy when a fully-static schedule is given beforehand. In this approach, they modify a given fully-static schedule so that the resulting fully-static schedule has  $T_{FS}$  equal to  $\lceil T_{ST} \rceil$ , and then to derive the transaction order from that modified schedule. 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 for the given FS schedule is always optimum.

The authors also describe the hardware support necessary to enforce the

transaction order at run-time. Furthermore they argue that even though the OT schedule is more sensitive to variations in execution times variations than St schedules, the difference in the performance of the ST and OT strategies is minimal and that if the variation times do vary significantly then even the ST approach is not practical.

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.

Numerous approaches have been proposed for incorporating IPC costs into the assignment and ordering steps of scheduling (e.g., [4, 32]). The techniques that we propose in the first half of the thesis 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. We do not know of any work which have studied ordered transactions in the context of non-negligible IPC costs.

Several mechanisms have been proposed for reducing synchronization accesses in self-timed schedules [7, 8, 33]. Before we discuss some of these mechanisms, it is important to understand exactly why these synchronizations are needed. In dataflow semantics, IPC edges between actors represent infinite buffers. An IPC

edge indicates that before the actual communication of data is carried out, the corresponding buffer is checked for overflow or underflow. However every feedback edge in the IPC graph only has a finite number of tokens at any instance of time [33] and can be bounded. However, feedforward edges that do not belong to any SCC have no such bound and for practical implementations, a bound has to be made that constrains the source actor from running ahead of the sink actor and that prevents exhausting the memory capacity available for shared buffers.

**Example 5:** Figure 8 shows an example of a dataflow graph with a feedforward edge and the effect on the graph when an artificial bound of  $m$  is made on the edge.

In [33], two basic synchronization protocols are introduced for feedforward and feedback edges. For feedforward edges, a synchronization protocol called unbounded buffer synchronization (UBS) is adopted which guarantees that an invo-

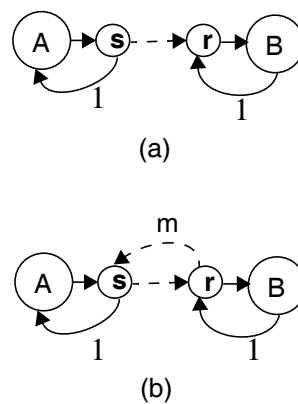


Figure 8. An IPC graph with a feedforward edge: (a) original graph, (b) imposing bounded buffers.

cation of the sink actor never attempts to read data from an empty buffer and an invocation of the source actor never attempts to write data into the buffer when the number of tokens in the buffer is more than some pre-specified limit. The synchronization protocol for feedback edges is termed bounded buffer synchronization (BBS) and it only has to ensure that the source actor does not try to read data from a empty buffer. Clearly, using the UBS protocol is more expensive than using the BBS protocol as more run-time checking is required.

An effective method for reducing synchronization costs is the detection and removal of *redundant* synchronization edges, which are synchronization edges whose respective synchronization functions are subsumed by other synchronization edges, and thus need not be implemented explicitly.

**Example 6:** Figure 9 displays an example of a graph with a redundant synchronization edge. Before executing actor  $D$ , the processor 1 does not need to synchronize with processor 2 because, due to synchronization edge  $x_1$ , the corresponding invocation of  $F$  is guaranteed to complete before each invocation of  $D$  is begun.

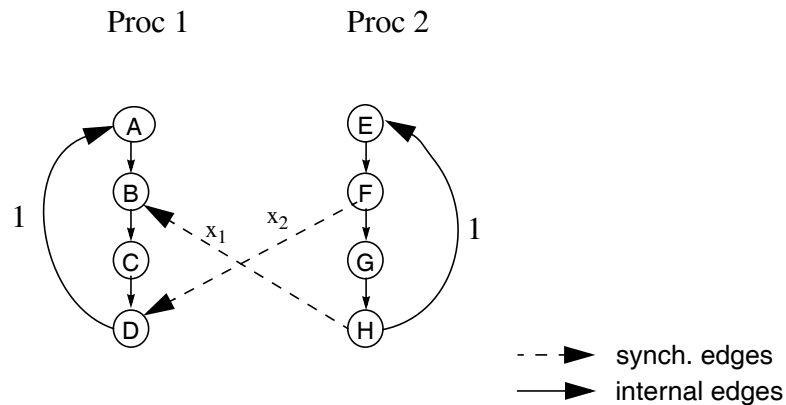


Figure 9.  $x_2$  is an example of a redundant synchronization edge.

An efficient polynomial time algorithm has also been proposed in [33] which performs systematic removal of all redundant synchronization edges. A second method for synchronization optimization is to examine the utility of adding additional synchronization edges to convert a synchronization graph that is not strongly connected into a strongly connected graph. Such a conversion allows all synchronization edges to be implemented with the BBS protocol, thus, reducing the synchronization cost. In [33], an algorithm has been proposed such that the extra synchronization accesses required for performing such a conversion are always (at least) compensated by the number of synchronization accesses that are saved by the more expensive UBS protocol instantiations that are converted to BBS instantiations.

A third mechanism, which is termed *resynchronization* [33], involves inserting new (extraneous) synchronization edges in such a way that the number of origi-

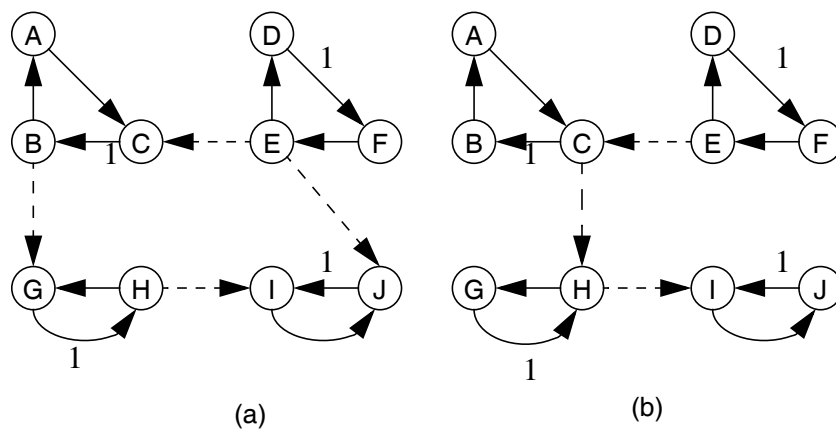


Figure 10. An example of resynchronization.

nal synchronization edges that become redundant exceeds the number of new edges added.

**Example 7:** Figure 10 displays an example where a synchronization edge between actors C and H can be added enabling the edge between actors B and G and the edge between actors E and J to become redundant. The resynchronization is only carried out for feedforward edges since if resynchronization is carried out for feedback edges, it may result in increasing the MCM of the associated IPC graph. Existing methods for resynchronization operate in the context of reducing synchronization costs without increasing the MCM so that the estimated throughput of the corresponding system does not degrade.

Note that all these methods proposed have been studied in the context of negligible communication costs. However, in the presence of non-zero IPC costs, it is possible to add additional edges that increase the MCM of the IPC graph but increase the throughput at the same time. This is possible because when we take IPC costs into account, shared-bus conflicts come into the picture and the exact throughput is determined by simulating the graph and not by measuring the MCM. In the second half of the thesis, we will propose a method of adding additional synchronization edges which may or may not be feedforward edges to increase the throughput.

## Chapter 5. Comparison of self-timed and ordered transaction strategies

In general, when IPC costs are negligible,  $T_{FS} \geq T_{OT} \geq T_{ST}$  [35] where  $T_{FS}$ ,  $T_{ST}$  and  $T_{OT}$  denote the iteration periods of the fully-timed, self-time and ordered transaction strategies respectively. This is because the ST method has the least 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 settles into periodic patterns of iterations. These patterns 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 non-zero. Depending on the bandwidth of the shared 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 ordered transaction graph since there will be no bus contention when a linear order is imposed on the communication actors [34].

## 5.1. Improved throughput

The relation  $T_{FS} \geq T_{OT} \geq T_{ST}$  is also no longer valid in the presence of non-zero IPC costs. 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 8:** To illustrate how an ST schedule might perform worse than an OT schedule, consider the IPC graph of Figure 6. Dashed edges represent inter-proces-

sor data dependencies. Numbers beside actors show their execution times, numbers beside edges indicate nonzero delays,  $x_s_y$  denotes the  $y$ th send actor of computation actor  $x$ , and  $x_r_y$  denotes the  $y$ th receive actor of  $x$ . Figure 11 shows the periodic pattern that the ST schedule eventually settles down into. 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 (Figure 12], 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 it requires it. 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.

## 5.2. Better predictability

**Example 9:** 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-

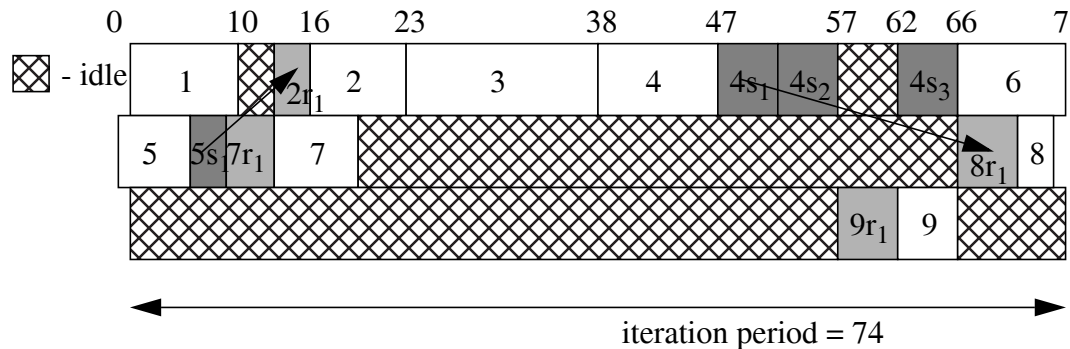


Figure 11. Gantt Chart for ST schedule in Example 8.

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. .

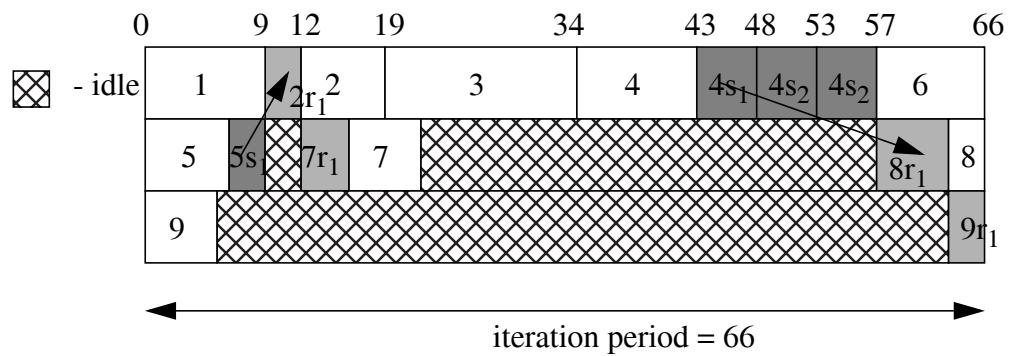


Figure 12. Gantt Chart for OT schedule in Example 8.

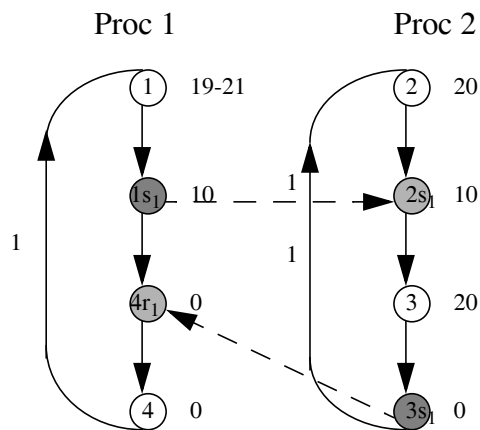


Figure 13. IPC graph for Example 10.

**Example 10:** Consider the IPC graph of Figure 13, and suppose that Actor 1 has a worst-case execution time of 21, and a best case execution time of 19. Figure 14 shows the ST schedule that results when actor 1 has an execution time of 21. An iteration 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 Figure 15.

**Example 11:** 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 perfor-

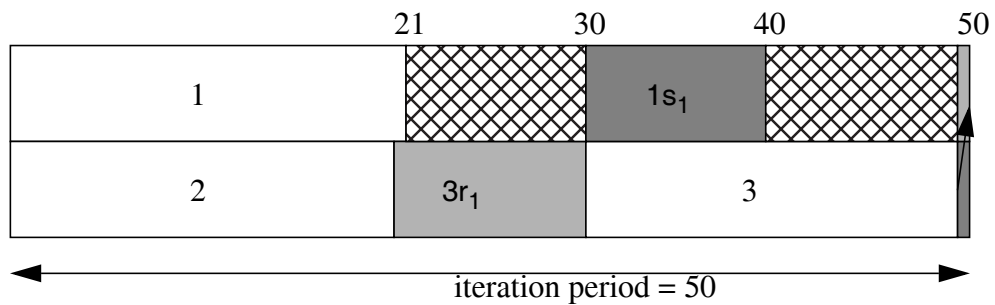


Figure 14. Gantt Chart for ST schedule when  $exec(1)=21$ .

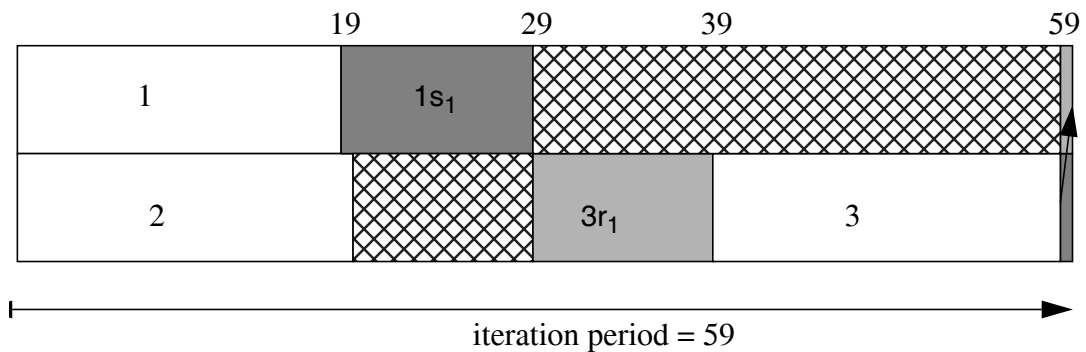


Figure 15. Gantt Chart for ST schedule with  $exec(1)=19$ .

mance for ordered transaction implementations [34,35] and the MCM can only increase or remain the same when the execution time of an actor is increased.

## Chapter 6. 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 Chapter 4, it has been shown that this problem is tractable when IPC costs are ignored. In this chapter, we show that when IPC costs are considered, the transaction ordering problem becomes NP-complete.

### 6.1. The non-iterative ordered transaction problem

We show this by first showing that determining an optimal transaction order for non-iterative implementations, 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  $E_{nipc} = \{e | (e \in E \text{ and } (delay(e) = 0))\}$ .

**Definition 2:** Given an NIPC graph  $G_{nipc} = (V, E_{nipc})$ , and an ordering  $O$  of 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  $V_{NOT} = V$ ,  $E_{NOT} = (E_{nipc} \cup E_O) - \{(o_p, o_1)\}$ , 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 non-negative. However, given an IPC graph, finding a transaction order that minimizes the makespan of the associated NOT graph is intractable.

**Example 12:** Figure 16 shows the NIPC graph for the corresponding IPC graph in Figure 6.

**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  $O = \{o_1, o_2, \dots, o_n\}$  such that  $G_{NOT} = \Pi(G_{nipc}, O)$  has a makespan that is less than or equal to  $k$ ?

To show that the non-iterative 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 [15]. 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)  $l(t) \in \mathfrak{N}$ , a release time  $r(t) \in \mathfrak{N}$ , and a deadline  $d(t) \in \mathfrak{N}$ , 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*)  $\sigma : T \rightarrow \mathbb{R}$ , 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 non-iterative 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 such as the Bellman-Ford algorithm.

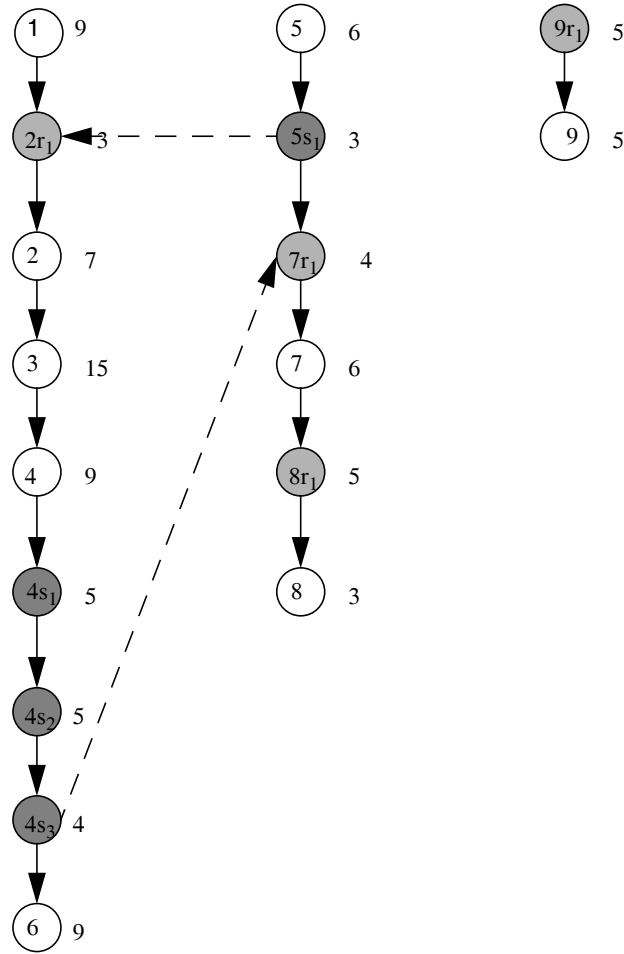


Figure 16. NIPC graph corresponding to IPC graph in Figure 6.

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

For each  $t_i \in T$ ,

i) Instantiate a send actor  $u_i$  when  $i$  is odd, or a receive actor  $u_i$  when  $i$  is even with  $exec(u_i) = l(t_i)$  and  $proc(u_i) = i$ .

ii) Instantiate a computation actor  $m_i$  with  $exec(m_i) = r(t_i)$  and  $proc(m_i) = i$ .

iii) Instantiate a computation actor  $n_i$  with  $exec(n_i) = k - d(t_i)$  and  $proc(n_i) = i$ .

iv) 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 1** There exists a transaction order  $O$  for  $G_{NOT} = \Pi(G_{nipc}, O)$  that results in a makespan that is less than or equal to  $k$ , if and only if 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 STRD 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 when they complete execution. Similarly, the execution time of the computation actors that follow the communication actors are chosen to be  $k - d(t_i)$  so that the corresponding communication actors will have to 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 non-iterative 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.*

Note that in Theorem 1, we have simplified the problem greatly by assuming the inter-processor edges to have unit delays. This removes the inter-dependencies that are imposed by these edges (within a given iteration period), but even with this simplification, the problem remains NP-complete.

**Example 13:** 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 non-iterative 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 Figure 17. 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.

## 6.2. NP-completeness of ordered transaction problem

As demonstrated by the Theorem 2 below, we can extend the proof of Theorem 1 to show that the transaction ordering problem is NP-complete in the *iterative*

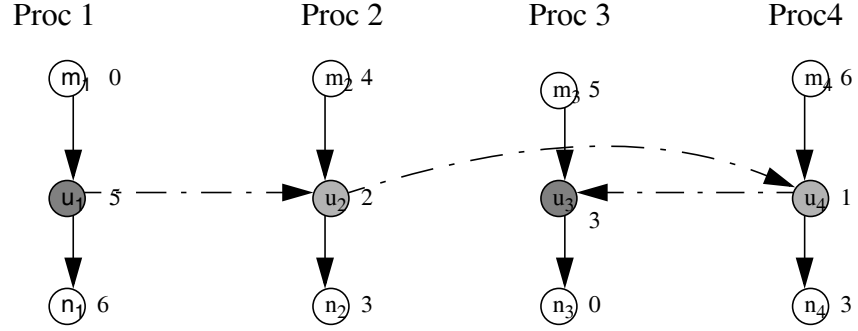


Figure 17. NOT Graph constructed in Example 13.

context as well as the non-iterative 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  $G_{OT} = \Gamma(G_{ipc}, O)$  satisfies  $MCM(G_{OT}) \leq k$ ?

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

*Proof:* 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.

Now suppose we are given an instance of the SRTD problem  $(T, r, l, d)$  with  $T = \{t_1, t_2 \dots t_p\}$ . We construct an IPC graph  $G_{ipc}$  from this instance by carrying out the following steps. All edges instantiated are delayless unless otherwise specified.  $k$  is equal to the maximum deadline of the tasks in the given instance of

the STRD problem.

For each  $t_i \in T$ ,

i) Instantiate a send actor  $u_i$  when  $i$  is odd, or a receive actor  $u_i$  when  $i$  is even with  $exec(u_i) = l(t_i)$  and  $proc(u_i) = i$ .

ii) Instantiate a computation actor  $m_i$  with  $exec(m_i) = r(t_i)$  and  $proc(m_i) = i$ .

iii) Instantiate a computation actor  $n_i$  with  $exec(n_i) = k - d(t_i)$  and  $proc(n_i) = i$ .

iv) Instantiate an edge  $(m_i, u_i)$  and another edge  $(u_i, n_i)$ .

v) Instantiate a send actor  $s_i$  with  $exec(s_i) = 0$  and  $proc(s_i) = i$ .

vi) Instantiate a receive actor  $r_i$  with  $exec(r_i) = 0$  and  $proc(r_i) = i$ .

vii) Instantiate a computation actor  $d_i$  with  $exec(d_i) = 0$  and  $proc(d_i) = i$ .

viii) Instantiate an edge  $(n_i, s_i)$ , an edge  $(s_i, r_i)$ , and another edge  $(r_i, d_i)$ .

ix) Instantiate another receive actor  $q_i$  with  $exec(q_i) = 0$  and  $proc(q_i) = p + 1$  (recall that  $p = |T|$ ).

x) Instantiate another send actor  $w_i$  with  $exec(w_i) = 0$  and  $proc(w_i) = p + 1$ .

xi) Instantiate an (interprocessor) edge  $(s_i, q_i)$  and another edge  $(w_i, r_i)$ .

After completing all the above, join all  $q_i$ s with edges in a linear chain, instantiate a computation actor  $h$  with  $exec(h) = 0$  and  $proc(h) = p + 1$ , instantiate edges  $(q_p, h)$  and  $(h, w_1)$  and again join all  $w_i$ s with edges in a linear chain.

Finally for each of the  $p + 1$  processors, add an edge with a delay of unity from the last actor on the processor to the first actor.

We again assume without loss of generality that there are an even number of tasks in  $T$ . Each send actor  $u_i$  is connected to the receive actor  $u_{i+1}$  with an inter-processor edge of unit delay. Note that in the OT graph  $\Gamma(G_{ipc}, O)$ , these inter-processor edges become *redundant* (in the sense of synchronization redundancy, as discussed in Chapter 4, because of the ordered transaction edges added due to  $O$ : since the ordered transaction edges are connected by a cycle of delay unity, the constraints imposed by  $\{(u_i, u_{i+1})\}$  are automatically met by the ordered transaction edges.

This graph effectively represents a blocked schedule for an iterative graph when the execution time of the actors which have been instantiated after step  $v$  have an execution time much less than the execution times of the other actors, and the MCM of the constructed graph represents the longest path or the schedule length of the graph. Note that each of the longest paths in the non-iterative graph will correspond to a cycle in the iterative case where the cycle mean of the cycle is equal to the longest path (since the denominator of the associated quotient in (3) is unity). Similarly as in the non-iterative case, it is possible to find a one-processor schedule of the STRD instance that satisfies the constraints if we can determine a transaction order whose enforcement will guarantee that the MCM of the corresponding OT graph is less than or equal to  $k$ . This is true because the communication actors that have non-zero IPC cost in the bus schedule of the OT problem correspond to the

tasks in the one-processor schedule of the STRD problem.

Hence, we can conclude that the (iterative) transaction ordering problem is NP-complete. *Q.E.D.*

**Example 14:** Consider again the SRTD instance of Example 14. Figure 18 shows the corresponding ordered transaction graph that results when the ordering  $(u_1, u_2, u_4, u_3)$  is imposed. Removing the OT edges

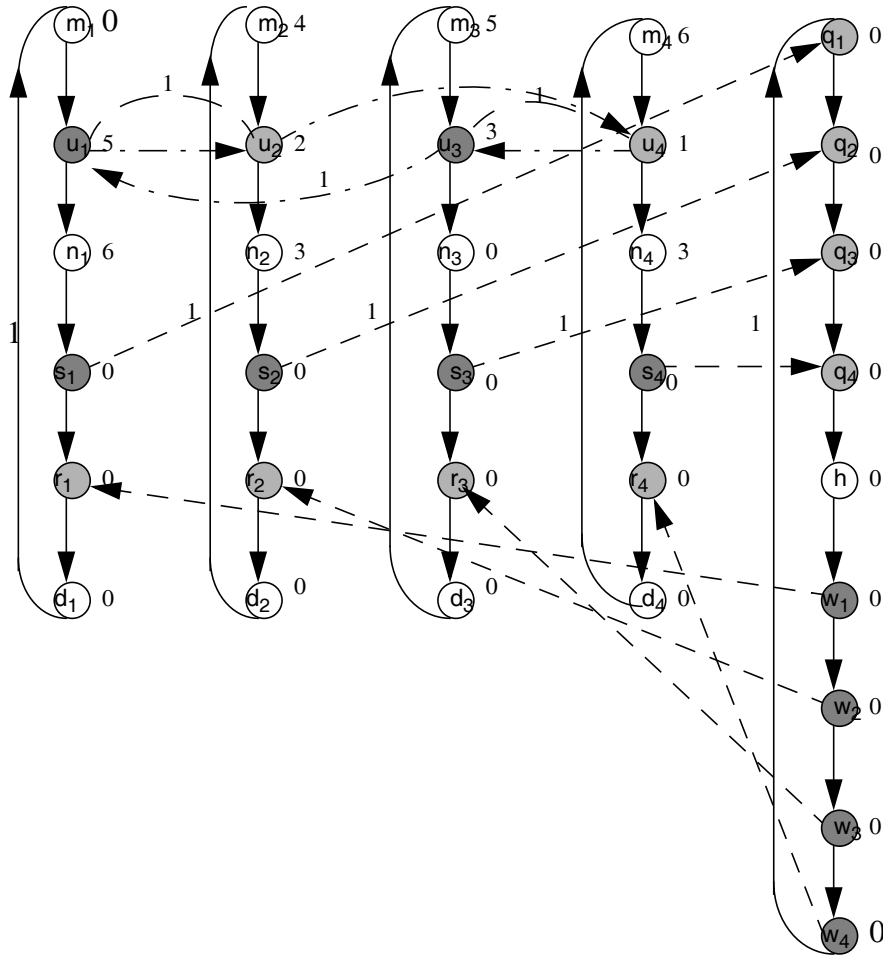


Figure 18. Constructed OT graph in Example 14.

$(u_1, u_2), (u_2, u_4), (u_4, u_3), (u_3, u_1)$  gives the constructed IPC graph. Note that the edges  $(u_1, u_2)$  and  $(u_3, u_4)$  introduced during construction are redundant in the OT graph due to the paths  $((u_1, u_2))$  and  $((u_3, u_1), (u_1, u_2), (u_2, u_4))$ , respectively, that are imposed by the linear order and have delays of one or less.

### 6.3. 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 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, 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 15:** The transaction partial order graph computed from the IPC graph of Figure 6 is illustrated in Figure 19. 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

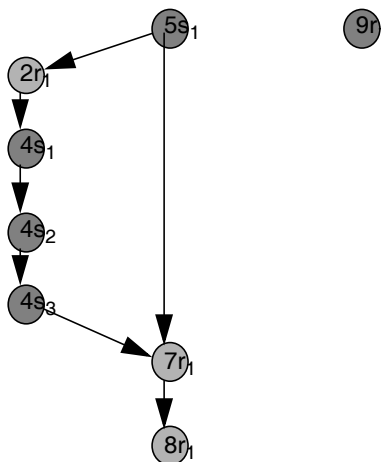


Figure 19. TPO Graph in Example 15.

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 the Inter-Processor Communication graph ( $G_{ipc}$ ), 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 scheduled into a linear ordering. A pseudocode specification of the TPO heuristic is given in Figures 20-22.

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 16:** When we apply the TPO heuristic to the IPC graph of Figure 6, the schedule we obtain is illustrated by the Gantt chart of Figure 12. The corresponding OT graph is illustrated in Figure 23.

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

**Function** Choose-Communication-Actor

**Input** an IPC graph  $G = (V, E)$ , a TPO graph  $G_{TPO}$  and a list of actors *ReadyList*

**Output** a communication actor  $v$

```
for  $x \in ReadyList$ 
    For  $y \in ReadyList$ 
        if  $x \neq y$ 
             $e = G.addedge(x, y)$ 
             $temp.addedge(e)$ 
        end if
    end for
     $criteria[x] = MCM(G)$ 
end for
For  $e \in temp$ 
     $G.delete(e)$ 
end for
 $v = \min\{criteria(x)\}$ 
return  $v$ 
```

Figure 20. Function to choose the next communication actor in the transaction order.

**Function** Initialize

**Input** an IPC graph  $G$

compute  $G_{TPO}$  from  $G$

```
For  $v \in G$ 
     $mark[v] = FALSE$ 
    if  $indegree(v) == 0$ 
         $ReadyList.append(v)$ 
    end if
end for
```

Figure 21. Function to initialize data structures called by the TPO function.

**Function** TPO-heuristic

**Input** an IPC graph  $G = (V, E)$

**Output** a linear list of communication actors *LinearList*

```
Initialize(G)
complete = FALSE
first = TRUE
while not (complete)
     $v = \text{choose-communication-actor}(G, G_{TPO}, \text{ReadyList})$ 
    mark[v] = TRUE
    LinearList.append(v)
    If not (first)
        G.addedge( $w, v$ )
         $w = v$ 
    end if
    first = FALSE
    For  $u \in \{(v, u) \in E\}$ 
        flag = TRUE
        For  $s \in \{(s, u) \in E\}$ 
            If mark[s] = FALSE
                flag = FALSE
            end if
        end for
        If flag == TRUE
            ReadyList.append( $u$ )
        end if
    end for
    If (ReadyList.empty) == TRUE
        complete = TRUE
    end if
end while
return LinearList
end Function
```

Figure 22. Pseudocode for TPO heuristic.



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  $G_{TPO}$  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  $O(P^2|V|^2|E_{OT}|)$ .

#### **6.4. 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 throughput 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  $G_{TPO}$ . A candidate vertex is chosen and deleted from  $G_{TPO}$  and the ready list is updated. In successive iterations, an edge is drawn in  $G_{ipc}$  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. A sketch

of the algorithm is given in Figure 24.

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 in a reasonable amount of time for complex graphs.

```

Function: BranchAndBound;
Input: an IPC graph  $G = (V, E)$ 
          a Transaction Partial Order Graph  $G_{TPO}$ 
          a linear list of actors ReadyList
          a start node  $v$ 
          a linear list of actors CurrList

UpdateReadyList;
If ReadyList
     $e = G.AddEdge(v, first\_actor\_curr\_list)$ 
     $M = max\_cycle\_mean(G)$ 
    If ( $M < best\_cycle\_mean$ )
         $LinearList = CurrList$ 
         $BestCycleMean = M$ 
    end if
     $G.delete(e)$ 
else
    for  $w \in ReadyList$ 
         $e = G.AddEdge(v, w)$ 
         $M = MaxCycleMean(G)$ 
        if ( $M < BestCycleMean$ )
             $CurrList.append(w)$ 
             $BranchAndBound(G, G_{TPO}, ReadyList, w, CurrList)$ 
             $CurrList.remove(w)$ 
        end if
         $G.delete(e)$ 
    end for
end if

```

Figure 24. Pseudocode for Branch and Bound strategy.

## 6.5. 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 [23], 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 [24]. 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 [24]. 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.

A pseudocode sketch of the GA is shown in Figure 25. For details on the underlying GA concepts (e.g., tournament selection), we refer the reader to [2]. 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.

## 6.6. Dynamic reordering

Once we obtain a transaction order (e.g., using the TPO heuristic or the GA approach defined in Section 6.5, 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

### Function TransOrderingGA

**Input** an IPC graph  $G = (V, E)$

**Output** a linear list of communication actors *LinearList*

compute  $G_{TPO}$  from  $G$

convert  $G_{TPO}$  to boolean matrix  $M_{TPO}$

generate initial population  $M$  by randomly completing  $M_{TPO}$

**for**  $j = 1$  to *NoIterations*

**for**  $i = 1$  to *PopulationSize*

$P_i = \text{mutate}^*(M_i)$

$R_i = \text{recombine}(P_{i-1}, P_i)$

$R_i.\text{FitnessValue} = \text{evaluate}(R_i, G)$

**end for**

$M = \text{tournament\_selection}(R, M)$

**end for**

Figure 25. Pseudocode for our GA approach to transaction ordering.

OBDD's (Ordered Binary Decision Diagrams) [25]. 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, which is represented as a Directed Acyclic Graph (DAG); its size may vary from linear to exponential. Dynamic ordering 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 [31]. 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 (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. A pseudo code sketch is given in Figure 26.

## **6.7. 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 [25]. The input to the simulator is a schedule. It specifies the number of processors, actor assignment and ordering on each processor, execution times and their respective probabilities for each actor, and synchronization and IPC constraints

(edges) between actors. The input also includes the number of clock cycles to simulate, and memory access time. The output is either a terse or verbose list of states describing the system at each step in the simulation. This output can be analyzed to determine the period of execution of the schedule, and therefore the system throughput. It can also be used to determine what percentage of the simulation time was used for memory access, processor idling and busy waiting.

```

Function Dynamic-Transaction-Reordering
Input an IPC graph  $G = (V, E)$ , a TPO graph  $G_{TPO}$ 
        and a list of actors  $O = \{o_1, o_2, \dots, o_p\}$ 
Output a linear list of communication actors LinearList

BestM = MaximumCycleMean( $\Gamma(G, O)$ )
TmpO = O
For  $o_i \in O$ 
     $j = 1; O = TmpO$ 
    while  $o_i \neq successor(o_{i-j})$  in  $G_{TPO}$ 
        swap( $o_i, o_{i-j}$ ) in  $O$ 
         $M = MaximumCycleMean \Gamma(G, O)$ 
        if ( $M < BestM$ )
             $BestM = M$ 
             $LinearList = O$ 
        end if
    end while
     $j = 1; O = TmpO$ 
    while  $o_i \neq predecessor(o_{i+j})$  in  $G_{TPO}$ 
        swap( $o_i, o_{i+j}$ ) in  $O$ 
         $M = MaximumCycleMean \Gamma(G, O)$ 
        if ( $M < BestM$ )
             $BestM = M$ 
             $LinearList = O$ 
        end if
    end while
end for
return LinearList

```

Figure 26. Pseudo code for Dynamic Transaction Reordering.

The simulated system is the 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 for a specified amount of time called the *back-off* time, and then tries again.

This is an event-based simulator. It operates by scheduling events such as the completion of an actor's execution, bus acquisition, and shared-memory accesses. The simulator implements both UBS and BBS protocols. In the BBS protocol (used with feedback synchronization edges), the protocol requires that 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  $snk(e)$  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  $src(e)$  can execute.

In the UBS protocol (used with feedforward 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  $src(e)$  executes, the shared count is repeatedly examined until it is found to be less than the feedforward buffer size, at which point the IPC operation can proceed. The count is incremented and  $snk(e)$  can execute. Before  $snk(e)$  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.

## 6.8. 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 6.3-6.6 were implemented in C/C++ using the *LEDA* [25] framework for fundamental graph-theoretic data structures and algorithms. The benchmarks are standard DSP applications that have been scheduled using the classic *HLFET* algorithm [17].

The IPC graphs are fairly complicated, ranging from between 10-800 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 [24]; *karp10* is a music synthesis application based on the Karplus Strong algorithm in 10 voices; the *video coder* and *decoder* are benchmarks described in [37]; *cddat* is a CD to digital audio tape sample rate converter taken

from the Ptolemy environment for heterogeneous systems [12]; *multirate1* and *multirate2* are tree shaped multirate filter banks; and *qmf4* is a 4 channel multi-resolution QMF filter bank for signal compression.

The iteration period for the ST schedule is obtained using the simulator described in Section 6.7. For these results, a fixed execution time has been chosen. In the next chapter, we will discuss the effects on the iteration period when we consider probabilistic execution times. 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 6.3-6.6, and  $T_{STi}$  denotes the average iteration period of the corresponding ST schedule with a synchronization cost of  $i$ . 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 by setting them uniformly to zero, which gives us a very optimistic view of the performance under ST execution. The iteration period using the OT strategy is approximately 5% less than the iteration period using the ST strategy with zero synchronization cost, and about 24% less with a synchronization cost (bus access time) of 2.

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 and  $T_C$  indicates the iteration period when DTR is used to refine the transaction order obtained using TPO and the solu-

tion is incorporated into the initial population of the GA. From the table we can conclude that all the heuristics work fairly well compared to random generation of transaction orders. The TPO heuristic for which the results have been shown is the modified version that inserts the delay beforehand. This “delay pre-insertion” consistently gives us a slight improvement over the original TPO algorithm. 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.

Table 1. Comparison of ST and OT schedules.

Application	$T_{ST0}$	$T_{ST1}$	$T_{ST2}$	$T_{ST3}$	$T_{OT}$
fft1	250	333	408	487	245
fft2	290	364	430	507	300
fft3	250	335	408	484	245
karp10	313	337	361	395	308
video coder	147	155	162	171	145
video decoder	33	37	40	44	33
cddat	4349	4733	5110	5532	4022
multirate1	1313	1529	1754	1986	1312
multirate2	318	406	480	567	316
qmf4	146	158	168	181	140

When we use the transaction ordering obtained by the TPO heuristic com-

Table 2. Comparison of algorithms.

Applica- tion	$T_{\text{random}}$	$T_{\text{BFB}}$	$T_{\text{TPO}}$	$T_{\text{GA}}$	$T_{\text{TPO+DTR}}$	$T_{\text{C}}$	$T_{\text{BB}}$
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

bined 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 since the complexity of the BB method is exponential, we are unable to achieve results for larger benchmarks. The results are shown in Table 2.

## Chapter 7. Software ordered transactions

To simultaneously leverage the advantages of the self-timed schedule (allowing successive iterations to be overlapped and allowing more flexibility for variations in actor execution times), and the ordered transaction schedule (efficient compile-time analysis of constraints), we would ideally like to find a smaller set of constraints instead of a complete linear ordering that would be enforced at run-time. However, the transaction controller hardware associated with these kind of constraints would be too complicated since it would have to run multiple threads to find out which communication operations have occurred and which are due. This kind of hardware would invariably increase the overall cost of the system making it unattractive for cost-effective embedded applications.

The OT edges can also be viewed as synchronization edges by themselves when they are implemented in software instead of hardware, the only difference being that there would be a communication cost associated with each of them. The resynchronization strategy [6] that we mentioned earlier tries to improve the performance of the system by adding synchronization edges and removing others that become redundant in the process. However, resynchronization edges that increased the MCM were not added since they may adversely affect the throughput of the system. Specifically, only feedforward edges were added to the IPC graph to avoid the overhead and complications of evaluating the performance impact of modifications

within strongly connected components. Since feedforward edges do not lie on the critical cycle of the graph, the MCM of the graph cannot increase and thus, the estimated throughput cannot be degraded [33].

However, when we study the problem more deeply in context of non-zero IPC costs, it is possible that the throughput of the system improves even when we add edges on the critical cycles of the IPC graph since these may or may not lie on the critical cycle of the self-timed simulation that includes the bus accesses. Moreover, it is possible that when we add feedback edges, several other synchronizations become redundant, thereby, improving the throughput further.

We propose a new strategy termed *software ordered transactions*, where we determine a set of synchronization constraints that may be either feedback or feedforward edges that are enforced at run-time using shared memory similar to resynchronization. The major difference being that the algorithms that determine these constraints do not differentiate between feedback and feedforward edges and the basic goal is to improve overall throughput rather than reduce the number of synchronizations.

**Definition 6:** Suppose that  $G_{ipc} = (V, E)$  is an IPC graph, then a *software transaction order* is a set of synchronization edges  $E_D = \{e_1, e_2, \dots, e_p\}$  such that each  $e_i$  is an edge between two communication actors of  $G_{ipc}$ . We impose no more restrictions. Note, that  $e_i$  may be an edge that already belongs to  $E$ .

It is only meaningful to add synchronization edges that do not result in deadlock [33] (i.e, cycles that have zero delay). The number of possible software transac-

tion orders is exponential in the number of actors in the graph. In this thesis, we shall restrict ourselves to the case where the software transaction order is a subset of the transaction order studied earlier. This software transaction will be termed as a *software ordered transaction*. We have already shown that the OT method works better than the self-timed schedule with accurate execution-time estimates even when synchronization costs are negligible. By carefully choosing a subset of the transaction ordering edges and implementing them in software instead of hardware, we attempt to find a schedule that will work better than the self-timed schedule and that does not require the hardware support of the OT method. This is especially useful in a case where implementing a hardware microcontroller is not feasible.

**Definition 7:** Suppose that  $G_{ipc} = (V, E)$  is an IPC graph. Then a software ordered transaction is a set of edges  $E_F = \{e_1, e_2, \dots, e_p\}$  such that each  $e_i \in E_O - \{o_p, o_1\}$  where  $E_O = \{o_1, o_2, \dots, o_p\}$  is some valid transaction order for  $G_{ipc}$ . Note, that  $e_i$  may be an edge that already belongs to  $E$ .

When we implement a software ordered transaction as a set of synchronizations, deadlock is not possible since we are restricting the edges to a subset of a valid OT transaction order and the transaction order is chosen taking the relevant IPC constraints into account.

We shall first show that finding an optimal software ordered transaction is NP-hard when synchronization cost is zero.

**Definition 8:** Given an NIPC graph  $G_{nipc} = (V, E_{nipc})$ , and a software ordered

transaction  $E_F$ , the corresponding *non-iterative software transaction ordered (NSTO) graph*  $G_{NSTO} = \sigma(G_{nipc}, E_F)$  is defined as  $G_{NSTO} = (V_{NSTO}, E_{NSTO})$ , where  $V_{NSTO} = V$ ,  $E_{NSTO} = (E_{nipc} \cup E_F)$ .

When a multiprocessor system implements  $G_{NSTO}$  in a self-timed manner, we obtain a *software ordered transaction (SOT) schedule*. Note that when  $E_F$  is null, the SOT schedule effectively becomes a self-timed schedule.

When implementing a graph in a self-timed manner, we make the following assumptions

i) Accesses made to the bus by the processors are queued. That is if a processor submits a communication operation  $i$  before another processor submits a communication operation  $j$  to the bus, the communication operation  $i$  is performed before communication operation  $j$ .

ii) In case two processors  $P_i$  and  $P_j$  submit communication operations at the same instant of time, the communication operation of the lower numbered processor  $\min(P_i, P_j)$  is performed first. Alternatively, any other policy can be defined as long as the decision is not arbitrary, but based on fixed deterministic criteria that is known at run-time.

**Definition 9:** The *non-iterative software 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 software ordered transaction  $E_F$  such that  $G_{NSTO} = \sigma(G_{nipc}, E_F)$  has a makespan that is less than or equal to  $k$ ?

As with the non-iterative transaction ordering problem and the iterative transaction ordering problem studied in Chapter 6, we show that non-iterative software 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 [15].

**Theorem 3:** The *non-iterative software transaction ordering* problem is NP-complete.

**Claim 2** When a software ordered transaction  $E_F$  is implemented on a non-iterative graph in a self-timed manner using the assumptions specified above, this results in a unique transaction order  $E_O$  that can be determined in polynomial time.

*Proof:* The unique transaction order can easily be found in polynomial-time by simulating the graph in a self-timed manner for one iteration. Since all decisions regarding scheduling of nodes is fixed from the above assumptions, there can be no node that is arbitrarily scheduled by the shared bus. This problem is therefore clearly in NP since we can verify in polynomial time whether the longest path length (in terms of cumulative execution time) of the non-iterative software transaction ordered 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  $T = \{t_1, t_2 \dots t_p\}$ . We construct an NIPC graph  $G_{nipc}$  from this instance by carrying out the following steps. Here, all edges instantiated are delayless unless otherwise specified.  $k$  is equal to at least the maximum deadline of the tasks in the given instance of the STRD problem.

For each  $t_i \in T$ , perform the following steps:

i) Instantiate a send actor  $u_i$  when  $i$  is odd, or a receive actor  $u_i$  when  $i$  is even with  $exec(u_i) = l(t_i)$  and  $proc(u_i) = i$ .

ii) Instantiate a computation actor  $m_i$  with  $exec(m_i) = r(t_i)$  and  $proc(m_i) = i$ .

iii) Instantiate a computation actor  $n_i$  with  $exec(n_i) = k - d(t_i)$  and  $proc(n_i) = i$ .

iv) 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 3** There exists a valid software ordered transaction  $E_F$  for

$G_{NSTO} = \sigma(G_{nipc}, E_F)$  that has a makespan that is less than or equal to  $k$ , if and only if there exists a valid SRTD schedule for the given instance of the SRTD problem.

The reasoning behind the claim is that a software ordered transaction when

implemented on a non-iterative graph in a self-timed manner using the assumptions specified above, results in a unique transaction order  $E_O$  as explained above (Claim 2)

**Example 17:** This reasoning is illustrated by means of the example in Figure 27.

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 non-iterative software 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 NSTO graph is illustrated in Figure 17. Dash-dot edges indicate SOT edges. Removing the dash-dot edges that represent the transaction order edges gives us the NIPC graph constructed from above. In the example,  $E_F = \{(u_4, u_3)\}$ . When this STO graph is implemented in a self-timed manner, this results in the uniquely transaction order  $(u_1, u_2, u_4, u_3)$ .

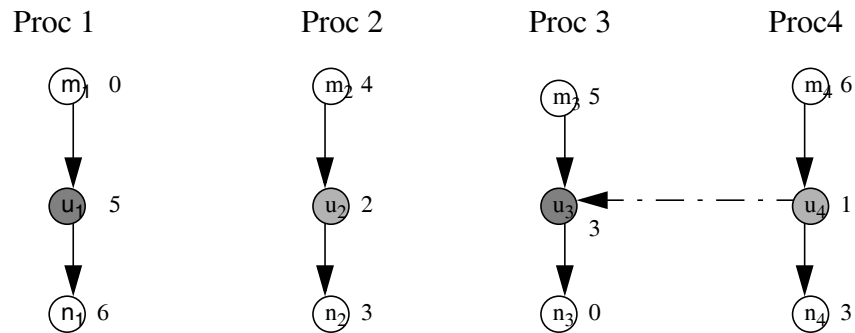


Figure 27. NSTO graph constructed in Example 17.

If we can obtain the unique transaction order that gives us a schedule length less than or equal to  $k$ , we can find a valid STRD schedule for the given instance of the STRD problem since the tasks in the STRD schedule correspond exactly to the communication actors on the bus. Therefore, from Theorem 1, the non-iterative software transaction ordering problem is NP-complete.

*Q.E.D.*

**Definition 10:** Given an IPC graph  $G_{ipc} = (V, E_{ipc})$ , and a software ordered transaction  $E_F$ , the corresponding *software transaction order (STO) graph*

$G_{STO} = \sigma(G_{ipc}, E_F)$  is defined as  $G_{STO} = (V_{STO}, E_{STO})$ , where  $V_{STO} = V$ ,  $E_{STO} = (E_{ipc} \cup E_F)$ .

**Definition 11:** The *iterative software transaction ordering problem* (also called the *software transaction ordering problem*) is defined as follows. Given an IPC graph  $G_{ipc}$  and a positive integer  $k$ , does there exist a software transaction order  $E_F$  such that for  $G_{STO} = \sigma(G_{ipc}, E_F)$ , the iteration period under the self-timed execution is less than or equal to  $k$ ?

The iteration period of the software transaction order graph can only be determined by simulating it in a self-timed manner and based on the present state of knowledge on self-timed systems since the iteration period and the transient can be exponential as explained earlier. It remains an open question whether or not the iterative software transaction ordering problem is in NP. As shown in the following theorem, we can however, establish that the iterative software transaction ordering

problem is NP-hard.

**Theorem 4:** The iterative software transaction ordering problem is NP-hard.

To establish NP-hardness, we again derive a reduction from the SRTD problem, and we modify the graph construction from the proof of Theorem 3 so that the schedule is blocked, that is, the iteration period remains the same for successive iterations.

Now suppose we are given an instance of the SRTD problem  $(T, r, l, d)$  with  $T = \{t_1, t_2 \dots t_p\}$ . We construct an IPC graph  $G_{ipc}$  from this instance by carrying out the following steps. All edges instantiated are delayless unless otherwise specified.  $k$  is equal to the maximum deadline of the tasks in the given instance of the STRD problem.

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

i) Instantiate a send actor  $u_i$  when  $i$  is odd, or a receive actor  $u_i$  when  $i$  is even with  $exec(u_i) = l(t_i)$  and  $proc(u_i) = i$ .

ii) Instantiate a computation actor  $m_i$  with  $exec(m_i) = r(t_i)$  and  $proc(m_i) = i$ .

iii) Instantiate a computation actor  $n_i$  with  $exec(n_i) = k - d(t_i)$  and  $proc(n_i) = i$ .

iv) Instantiate an edge  $(m_i, u_i)$  and another edge  $(u_i, n_i)$ .

v) Instantiate a send actor  $s_i$  with  $exec(s_i) = 0$  and  $proc(s_i) = i$ .

vi) Instantiate a receive actor  $r_i$  with  $exec(r_i) = 0$  and  $proc(r_i) = i$ .

vii) Instantiate a computation actor  $d_i$  with  $exec(d_i) = 0$  and

$proc(d_i) = i$ .

viii) Instantiate an edge  $(n_i, s_i)$ , an edge  $(s_i, r_i)$ , and another edge  $(r_i, d_i)$ .

ix) Instantiate another receive actor  $q_i$  with  $exec(q_i) = 0$  and

$proc(q_i) = p + 1$  (recall that  $p = |T|$ ).

x) Instantiate another send actor  $w_i$  with  $exec(w_i) = 0$  and

$proc(w_i) = p + 1$ .

xi) Instantiate an (interprocessor) edge  $(s_i, q_i)$  and another edge  $(w_i, r_i)$ .

After completing all the above, join all  $q_i$ s with edges in a linear chain, instantiate a computation actor  $h$  with  $exec(h) = 0$  and  $proc(h) = p + 1$ , instantiate edges  $(q_p, h)$  and  $(h, w_1)$  and again join all  $w_i$ s with edges in a linear chain. Finally for each of the  $p + 1$  processors, add an edge with a delay of unity from the last actor on the processor to the first actor.

We again assume without loss of generality that there are an even number of tasks in  $T$ . Each send actor  $u_i$  is connected to the receive actor  $u_{i+1}$  with an interprocessor edge of unit delay. Note that in the STO graph  $\sigma(G_{ipc}, E_F)$ , these interprocessor edges become *redundant* since the schedule is blocked.

Note that each of the successive iterations is going to have the same iteration period. Similarly, as in the non-iterative case, it is possible to find a unique transaction order for any software ordered transaction by simulating the graph for one iteration. From the unique transaction order  $E_O$ , the one-processor schedule of the STRD instance that satisfies the constraints can be determined. If we can find a transaction order whose enforcement will guarantee that the iteration period of the

corresponding STO graph is less than or equal to  $k$ , we can effectively solve the SRTD problem.

Hence, from Theorem 2 we can conclude that the (iterative) software transaction ordering problem is NP-hard.

*Q.E.D.*

**Example 18:** Consider again the SRTD instance of Example 14. Figure 18 shows

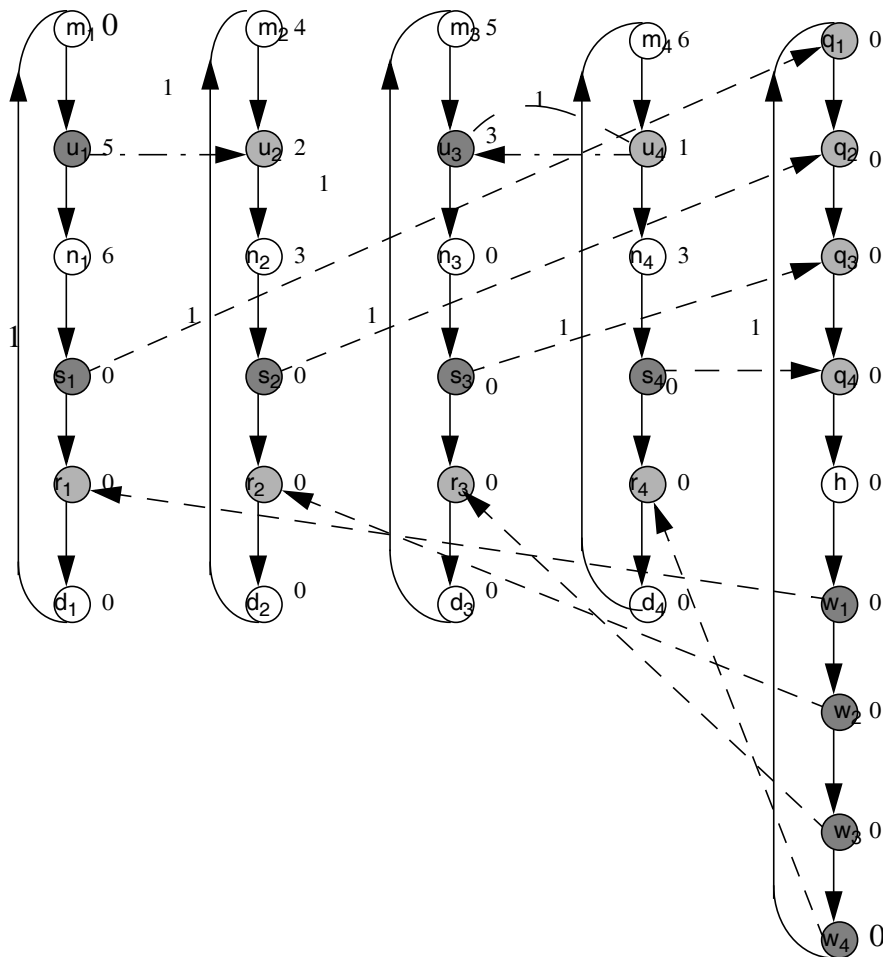


Figure 28. Constructed STO graph in Example 14.

the corresponding STO graph that results when the software transaction order  $E_F = \{(u_4, u_3)\}$  is imposed. The unique transaction order determined from the software transaction order is  $(u_1, u_2), (u_2, u_4), (u_4, u_3), (u_3, u_1)$ .

## 7.1. Transaction order removal (TOR) heuristic

All transaction order edges of the OT graph do not contribute in decreasing the iteration period of the graph. Only some of the edges are responsible for adding constraints that would not have been followed by the self-timed schedule. Most of the edges are, therefore, redundant in the sense of synchronization redundancy. That is, the synchronization constraints imposed by these edges are automatically met by the other edges in the system. Other edges of the transaction order might be actually increasing the iteration period by removing the flexibility of the self-timed schedule in places where the flexibility is beneficial. Therefore, our algorithm for finding software ordered transactions, which we call the *Transaction Order Removal (TOR)* algorithm, tries to carefully prune the redundant edges of a given transaction order since they add additional synchronization cost, and also to remove other edges that might increase the iteration period. It tries to retain only those edges that may actually increase the throughput through strategic serialization of communication operations.

The TOR algorithm takes as input an IPC graph and a transaction order. The algorithm initially adds all the edges in the transaction order to the graph. After that,

Function TransactionOrderRemoval;  
 Input : an IPC graph  $G = (V, E)$   
           a transaction order  $E_O$   
 Output : a set of synchronization edges  $E_F$

```

 $E_F = E_O - \{(o_p, o_1)\}$ 
for i = 1 to p
    keep[i] = true
endfor
outer_best_iter =  $\infty$ 
for i = 1 to p
    best_iter =  $\infty$ 
    for  $e_i \in E_F$ 
        keep[i] = false
         $G_{STO} = \text{add } E_F \text{ to } G_{ipc}$ 
        iter = call_simulation( $G_{STO}$ )
        if (iter < best_iter)
            best_iter = iter
            index = i
        endif
        keep[i] = true
    endfor
    if (best_iter < outer_best_iter)
         $E_F = E_F - e_{index}$ 
        outer_best_iter = best_iter
    endif
endfor
iter = call_simulation( $G_{ipc}$ )
if (iter <= outer_best_iter)
    return(null)
else
    return( $E_F$ )
endif

```

Figure 29. Pseudocode for Transaction Order Removal algorithm.

in each successive iteration, it removes one edge at a time, calculates the iteration period by simulating the graph in a self-timed manner and adds the edge back. The edge that corresponds to the best iteration period found using this process is removed permanently from the list and the entire process is repeated. A sketch of the algorithm is given in Figure 29.

As explained above, our aim is to remove transaction order edges from the OT graph that are not significant and the best way to measure that is by looking at the iteration period of the corresponding ST schedule. The algorithm works in a greedy fashion by removing edges that decrease the iteration period the most and continues in this fashion until it finds that removing any edge increases the iteration period. The best iteration period is finally compared to the iteration period of the self-timed schedule and if it is less than the best iteration period found, the algorithm returns the null set indicating that no edges should be added. In this way, we are assured that it never gives us a result that is worse than the self-time schedule. The complexity of the algorithm  $O(|E_o|^3T)$  where  $T$  is the complexity of carrying out the self-timed simulation. This is dependent on a number of factors such as the number of edges and actors in the STO graph and also depends on number of synchronization accesses and bus conflicts. Note that before the self-timed simulation is called, the RemoveRedundantEdges algorithm [33] is called to remove any synchronization redundancies as a result of the extra OT edges or edges inherent in the IPC graph. Since  $T$  is in the order of a few seconds for even graphs of small complexity, the algorithm can take considerable time to run. Thus, it is more suited to the final

implementation phase of embedded system development rather than early-stage prototyping

## **7.2. Genetic algorithm for software ordered transactions**

The search space for finding a suitable software ordered transaction is much larger than that for the OT problem. Also, since the iteration period is measured using simulation techniques, it is not possible in general to compute the optimal solution. To further exploit any increased tolerance in compile-time, we have extended the trans-ordering genetic algorithm to search for solutions in the software ordered transaction problem. We call this extended genetic algorithm technique the software transordering GA

As before, transaction orders are encoded using the matrix based formulation described earlier and are also accompanied by a bit vector to indicate whether the edges are present in the software ordered transaction or not. The evaluation criteria is changed to calling the simulation instead of calculating the MCM. Therefore, the time taken to perform the software ordered transaction GA is significantly larger than that of the transordering GA. Unlike the TOR algorithm, the GA does not take a transaction order in its input. It simultaneously tries to determine a transaction order and extract a subset of the transaction order to implement in software. A brief sketch of the software transordering GA is given in Figure 30.

## **7.3. Results**

The simulator is used to measure how effective the software ordered transac-

tion is and to gauge the performance of the SOT and the GA algorithms. Figure 31 shows the results for application benchmark *qmf4* when synchronization cost is negligible. Simulations have been carried out with probabilistic execution times. The

**Function** SoftwareTransactionOrderingGA

**Input** an IPC graph  $G = (V, E)$

**Output** a linear list of communication actors *LinearList*

compute  $G_{TPO}$  from  $G$

convert  $G_{TPO}$  to boolean matrix  $M_{TPO}$

generate initial population  $M$  by randomly completing  $M_{TPO}$

initialize boolean vector  $V$  to 1

**For**  $j = 1$  to *NoIterations*

**For**  $i = 1$  to *PopulationSize*

$P_i = \text{mutate}^*(M_i, V_i)$

$R_i = \text{recombine}(P_{i-1}, P_i)$

$R_i.\text{FitnessValue} = \text{evaluate}(R_i, G)$

**end for**

$(M, V) = \text{tournament\_selection}(R, (M, V))$

**end for**

Figure 30. Pseudocode for GA approach to software ordered transactions.

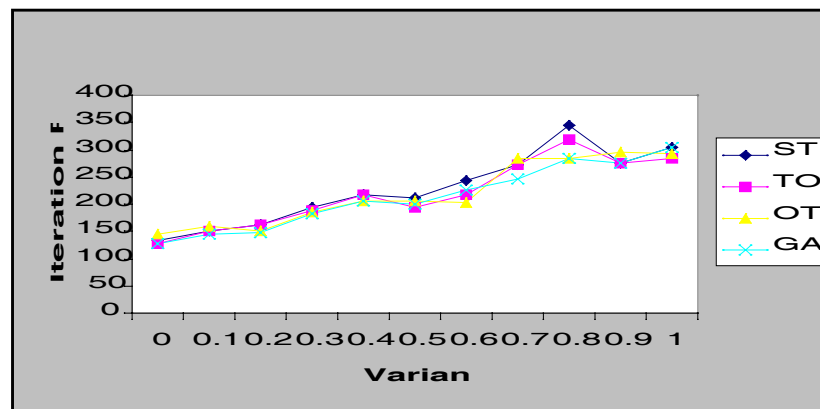


Figure 31. Comparison without synchronization costs for benchmark *qmf4*.

graph shows the iteration periods when the variance is increased. The upward trend does not indicate that the performance degradability results directly from higher variances. This is because the average execution time of the actors increases with increased variance in the model we have implemented. From Figure 31, it is apparent that the TOR algorithm works better than the self-timed schedule in most cases. When the variance is low, we observe that the OT schedule is quite good but degrades as variance is increased. Overall the software transaction ordered GA works the best at the cost of longer simulation time. Figure 32 shows the graph when synchronization cost is non-negligible. Note that the OT curve remains the same since it is not affected by synchronization cost. However the other curves move upwards indicating higher iteration periods. We also observe (not indicated in graph) that the number of synchronization edges in the software ordered transaction is usually very small (between 1-4 synchronization edges). This demonstrates that the actual difference between the performance of the ST and the OT schedules is usually

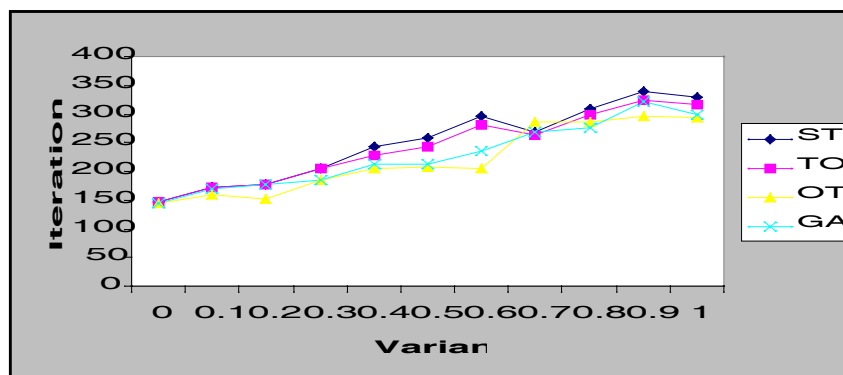


Figure 32. Comparison with synchronization costs for benchmark qmf4.

due to only a few key constraints. Figure 33 and 34 show the trends when the simulation is carried out for the *video\_coder* benchmark which is less complex compared to *qmf4*. The TOR algorithm performs much better for this benchmark. In almost all the cases, it performs better. Also the OT strategy performs much worse for higher variances in execution times of the actors. When the GA and the TOR algorithm work better than the ST method, the software ordered transaction has exactly 1 synchronization edge ( in addition to the synchronization constraints already present in

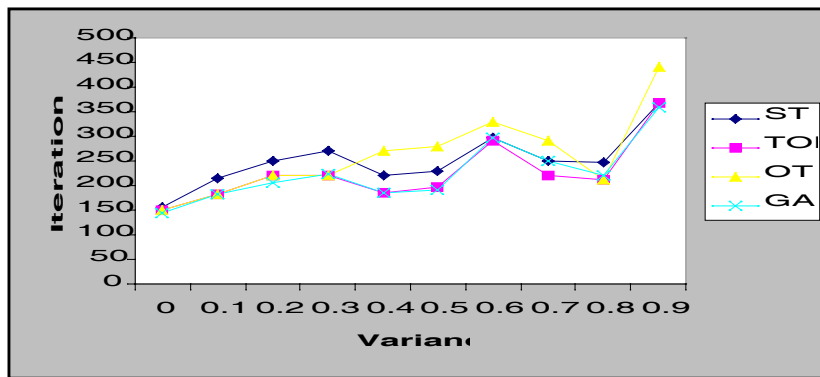


Figure 33. Comparison without synchronization costs for the *video\_coder* benchmark.

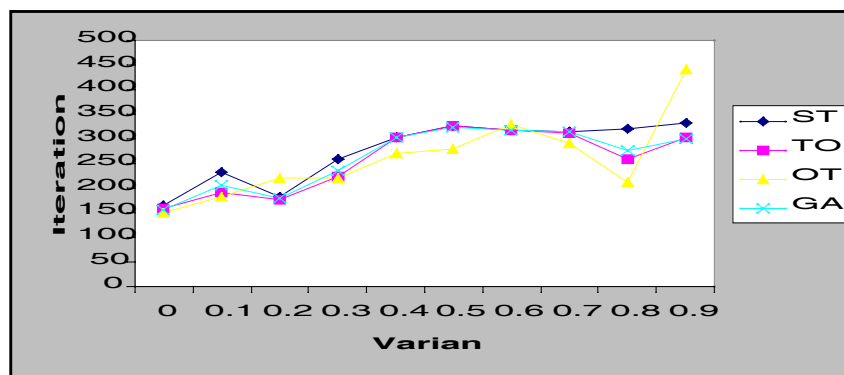


Figure 34. Comparison with synchronization costs for the *video\_coder* benchmark.

the self-timed schedule). This shows that one crucial synchronization edge when added can make a difference of 15% in the throughput. The OT schedule works comparably in the non-zero synchronization cost cases compared to the software ordered transaction strategy.

## Chapter 8. Conclusions and future work

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 under both iterative and non-iterative execution. Furthermore, 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.

In the second half of the thesis, we have proposed a method to integrate the self-timed strategy and the ordered transaction strategy. By adding a relatively small number of synchronization edges, we have shown that it is possible to improve the throughput of a self-timed schedule. We have shown that even under negligible synchronization costs, finding an optimal subset of synchronizations to implement (with the goal of maximizing throughput) is intractable. Again, this intractability holds under both iterative and non-iterative executions. We have also shown effec-

tive methods to compute software ordered transactions and have shown experimentally that they are effective even when there is variation in the execution times of actors.

When synchronization costs are high, the ordered transaction strategy can be used to improve performance and predictability. On the other hand, if synchronization costs are low or extra hardware cannot be incorporated, the software ordered transaction is an attractive approach. Useful directions for further work include integrating transaction ordering considerations into the scheduling process, and the exploration of strategies that find synchronization edges that may or may not be a subset of a transaction order for e.g. when a synchronization set consists of 2 edges incident on the same communication actor. This kind of a set cannot be a software ordered transaction since it is not a subset of any transaction order.

## REFERENCES

- [1] F. Baccelli, G. Cohen, G. J. Olsder, and J. Quadrat, *Synchronization and Linearity*, John Wiley & Sons, Inc., 1992.
- [2] 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.
- [3] N. K. Bambha and S. S. Bhattacharyya, "A joint power/performance optimization technique for multiprocessor systems using a period graph construct", *Proceedings of the International Symposium on Systems Synthesis*, pages 91-97, Madrid, Spain, September 2000.
- [4] S. Banerjee, T. Hamada, P. M. Chau and R. D. Fellman, "Macro Pipelining Based Scheduling on High Performance Heterogeneous Multiprocessor Systems," *IEEE Transactions on Signal Processing*, Vol. 43, No. 6, pp 1468-1484, June, 1995.
- [5] B. Bhattacharya and S. S. Bhattacharyya, "Parameterized dataflow modeling of DSP systems", *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, pages 1948-1951, Istanbul, Turkey, June 2000.
- [6] S. Bhattacharyya, S. Sriram, and E. A. Lee, "Self-timed resynchronization: A post-optimization for static multiprocessor schedules", *Proceedings of the International Parallel Processing Symposium*, pages 199-205, April 1996, Honolulu, Hawaii.
- [7] 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.
- [8] 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.
- [9] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, *Software Synthesis from Dataflow Graphs*, Kluwer Academic Publishers, 1996.
- [10] S. H. Bokhari, "Partitioning problems in parallel, pipelined, and distributed computing", *IEEE Transactions on Computers*, pp. 48-57, January, 1988.

- [11] J. T. Buck, *Scheduling Dynamic Dataflow Graphs with Bounded Memory using the Token Flow model*, Ph.D. Thesis, Department of Computer Engineering and Computer Sciences, University of California at Berkeley, September, 1993.
- [12] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems," *International Journal of Computer Simulation*, Vol. 4, pp. 155-182, Jan. 1994.
- [13] A. Dasdan and R. K. Gupta, "Faster Maximum and Minimum Mean Cycle Algorithms for System Performance Analysis", *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 17(10), Oct. 1998.
- [14] B. R. Fox and M. B. McMahon, "Genetic Operators for Sequencing Problems", G. Rawlins, "*Foundations of Genetic Algorithms*, Morgan Kaufman Publishers Inc., 1991.
- [15] 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.
- [16] R. L. Graham, "Bounds on multiprocessing timing anomalies", *SIAM Journal of Applied Math*, 17(2):416-429, March 1969.
- [17] T. C. Hu, "Parallel sequencing and assembly line problems," *Operations Research*, Vol. 9, No. 6, pp. 841-848, Nov. 1961.
- [18] S. Y. Kung and P. S. Lewis and S. C. Lo, "Performance Analysis and Optimization of VLSI Dataflow Array," *Journal of Parallel and Distributed Computing*, pp. 592-618, 1987.
- [19] E. A. Lee, "Consistency in dataflow graphs", *IEEE Transactions on Parallel and Distributed Systems*, April, 1991.
- [20] E. A. Lee and J. C. Bier, "Architectures for Statically Scheduled Dataflow," *Journal of Parallel and Distributed Computing*, Vol. 10, pp. 333-348, Dec. 1990.
- [21] E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Dataflow Programs for Digital Signal Processing," *IEEE Trans. on Computers*, Vol. C-36, No. 2, Feb. 1982.
- [22] E. A. Lee and S. Ha, "Scheduling strategies for Multiprocessor real-time DSP," *Proceedings of the Globecom Conference*, Dallas, Texas, pp. 1279-1283, Nov. 1989.
- [23] P. Marwedel and G. Goossens, *Code Generation for Embedded Processors*, Kluwer Academic Publishers, 1995.

- [24] C. L. McCreary, A. A. Kahn, J. J. Thompson and M. E. McArdle, "A Comparison of Heuristics for Scheduling DAGS on Multiprocessors," *International Parallel Processing Symposium*, 1994.
- [25] K. Mehlhorn and Stefan Naher, *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 199G.
- [26] De. Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 1994.
- [27] G. J. Olsder, *Systolic Array Processors; Contributions by Speakers at the International Conference on Systolic Arrays*, Prentice-Hall, 1989.
- [28] 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.
- [29] J. L. Peterson, *Petri Net Theory and Modeling of Systems*, Prentice-Hall Inc., Englewoods Cliffs, New Jersey, 1981.
- [30] R. Reiter, "Scheduling Parallel Computations," *Journal of the Association for Computing Machinery*, Vol. 15, No. 4, pp. 590-599, Oct. 1968.
- [31] R. Rudell, "Dynamic Variable Ordering for Ordered Binary Decision Diagrams," *IEEE Transactions on Computer Aided Design*, 1993.
- [32] G. C. Sih, "Multiprocessor Scheduling to account for Interprocessor Communication," Ph.D. Thesis, Memorandum No. UCB/ERL M91/29, Electronics Research Laboratory, University of California at Berkeley, April, 1991.
- [33] S. Sriram and S. S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*, Marcel-Dekker, Inc., 2000.
- [34] 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.
- [35] S. Sriram, "Minimizing Communication and Synchronization Overhead in Multiprocessors for Digital Signal Processing," Ph.D. Thesis, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, 1995.

[36] S. Sriram and E. A. Lee, "Design and implementation of an ordered memory access architecture", *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, pages 345-348, April 1993.

[37] J. Teich, and T. Blickle, "System-Level Synthesis using Evolutionary Algorithms", *J. Design Automation of Embedded Systems*, vol. 3, no. 1, pp. 23-58, Kluwer Academic Publishers, Jan. 1998.

[38] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe and A. Agarwal, "Baring it all to Software: Raw Machines", *IEEE Computer*, September 1997, pp. 86-93.