

Copyright © 1995, by the author(s).  
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

**MINIMIZING COMMUNICATION AND  
SYNCHRONIZATION OVERHEAD IN  
MULTIPROCESSORS FOR DIGITAL  
SIGNAL PROCESSING**

by

Sundararajan Sriram

Memorandum No. UCB/ERL M95/90

7 November 1995

*COVER PAGE*

**MINIMIZING COMMUNICATION AND  
SYNCHRONIZATION OVERHEAD IN  
MULTIPROCESSORS FOR DIGITAL  
SIGNAL PROCESSING**

Copyright © 1995

by

**Sundararajan Sriram**

Memorandum No. UCB/ERL M95/90

7 November 1995

**ELECTRONICS RESEARCH LABORATORY**

College of Engineering  
University of California, Berkeley  
94720

## **Abstract**

# **MINIMIZING COMMUNICATION AND SYNCHRONIZATION OVERHEAD IN MULTIPROCESSORS FOR DIGITAL SIGNAL PROCESSING**

**by**

**Sundararajan Sriram**

**Doctor of Philosophy in Electrical Engineering**

**Professor Edward A. Lee, Chair**

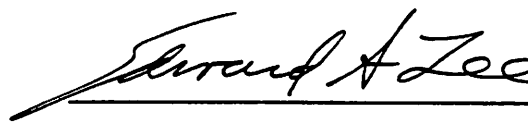
This thesis is concerned with embedded systems for Digital Signal Processing (DSP) that consist of multiple programmable digital signal processors augmented with custom VLSI components; we will refer to such systems by the term “multiprocessor.” The dataflow model of computation has been widely used for providing a formal methodology for specifying computations and mapping them to such multiprocessor systems.

In this thesis, we focus on DSP algorithms that can be specified as Synchronous Data Flow graphs and its extensions. Such algorithms can be efficiently scheduled onto multiple processing elements (a processor could be either programmable or a custom VLSI component) at compile time - computations in the graph are assigned to processors at compile time and the execution order of tasks assigned to each processor is also determined at compile time.

In such a compile-time (static) scheduling strategy, it is possible to predict

the run time inter-processor communication (IPC) pattern. We present two techniques that make use of this compile-time determined communication pattern, for minimizing IPC and synchronization overhead in the parallel implementation. The first technique is aimed at eliminating arbitration and synchronization costs when using shared memory for IPC. We call this the Ordered Transactions strategy; the idea is to determine the order in which processors require access to shared resources and to enforce this order at run time. Enforcing such an order eliminates contention for shared resources and the need for explicit synchronization. We describe the design and hardware implementation details of a prototype multiprocessor board that was built as a proof-of-concept for the ordered transactions strategy.

The second technique we present in this thesis consists of efficient algorithms for minimizing synchronization costs in statically scheduled multiprocessors. These include procedures for detecting and eliminating redundant synchronization points in the schedule and systematically adding certain synchronization points with a view towards reducing the overall synchronization cost.

 Nov. 7, 1995

**Edward A. Lee, Thesis Committee Chairman**

# Table of Contents

<b>1</b>	<b>INTRODUCTION.....</b>	<b>1</b>
1.1	The Synchronous Dataflow model.....	7
1.1.1	Background.....	7
1.1.2	Utility of dataflow for DSP .....	11
1.2	Parallel scheduling .....	13
1.2.1	Fully-static schedules .....	15
1.2.2	Self-timed schedules.....	19
1.2.3	Execution time estimates and static schedules .....	21
1.3	Application-specific parallel architectures.....	24
1.3.1	Dataflow DSP architectures .....	24
1.3.2	Systolic and wavefront arrays .....	25
1.3.3	Multiprocessor DSP architectures .....	26
1.4	Thesis overview: our approach and contributions .....	27
<b>2</b>	<b>TERMINOLOGY AND NOTATIONS .....</b>	<b>33</b>
2.1	HSDF graphs and associated graph theoretic notation .....	33
2.2	Schedule notation.....	35
<b>3</b>	<b>THE ORDERED TRANSACTION STRATEGY.....</b>	<b>39</b>
3.1	The Ordered Transactions strategy .....	39
3.2	Shared bus architecture .....	42
3.2.1	Using the OT approach.....	46
3.3	Design of an Ordered Memory Access multiprocessor .....	47
3.3.1	High level design description .....	48
3.3.2	A modified design .....	49
3.4	Design details of a prototype .....	52
3.4.1	Top level design .....	53
3.4.2	Transaction order controller .....	55
3.4.2.1	Processor bus arbitration signals.....	55
3.4.2.2	A simple implementation .....	57

3.4.2.3. Presetable counter .....	58
3.4.3 Host interface.....	60
3.4.4 Processing element .....	61
3.4.5 Xilinx circuitry .....	62
3.4.5.1. I/O interface .....	64
3.4.6 Shared memory.....	65
3.4.7 Connecting multiple boards.....	65
3.5 Hardware and software implementation .....	66
3.5.1 Board design.....	66
3.5.2 Software interface.....	69
3.6 Ordered I/O and parameter control .....	71
3.7 Application examples.....	73
3.7.1 Music synthesis .....	73
3.7.2 QMF filter bank.....	75
3.7.3 1024 point complex FFT .....	76
3.8 Summary .....	78
<b>4 AN ANALYSIS OF THE OT STRATEGY .....</b>	<b>79</b>
4.1 Inter-processor Communication graph ( $G_{ipc}$ ).....	82
4.2 Execution time estimates .....	88
4.3 Ordering constraints viewed as edges added to $G_{ipc}$ .....	89
4.4 Periodicity .....	90
4.5 Optimal order .....	92
4.6 Effects of changes in execution times.....	96
4.6.1 Deterministic case .....	97
4.6.2 Modeling run time variations in execution times .....	99
4.6.3 Implications for the OT schedule .....	104
4.7 Summary .....	106
<b>5 MINIMIZING SYNCHRONIZATION COSTS IN SELF-TIMED SCHEDULES .....</b>	<b>107</b>

5.1	Related work .....	108
5.2	Analysis of self-timed execution.....	112
5.2.1	Estimated throughput.....	114
5.3	Strongly connected components and buffer size bounds .....	114
5.4	Synchronization model .....	116
5.4.1	Synchronization protocols .....	116
5.4.2	The synchronization graph $G_s$ .....	118
5.5	Formal problem statement .....	122
5.6	Removing redundant synchronizations .....	124
5.6.1	The independence of redundant synchronizations .....	125
5.6.2	Removing redundant synchronizations .....	126
5.6.3	Comparison with Shaffer's approach .....	128
5.6.4	An example.....	129
5.7	Making the synchronization graph strongly connected .....	131
5.7.1	Adding edges to the synchronization graph .....	133
5.7.2	Insertion of delays .....	137
5.8	Computing buffer bounds from $G_s$ and $G_{ipc}$ .....	141
5.9	Resynchronization.....	142
5.10	Summary .....	144
<b>6</b>	<b>EXTENSIONS.....</b>	<b>147</b>
6.1	The Boolean Dataflow model .....	147
6.1.1	Scheduling .....	148
6.2	Parallel implementation on shared memory machines .....	152
6.2.1	General strategy.....	152
6.2.2	Implementation on the OMA.....	155
6.2.3	Improved mechanism .....	157
6.2.4	Generating the annotated bus access list .....	161
6.3	Data-dependent iteration .....	164
6.4	Summary .....	165



<b>7</b>	<b>CONCLUSIONS AND FUTURE DIRECTIONS.....</b>	<b>166</b>
<b>8</b>	<b>REFERENCES.....</b>	<b>170</b>

# List of Figures

Figure 1.1.	Fully static schedule .....	16
Figure 1.2.	Fully-static schedule on five processors .....	17
Figure 1.3.	Steps in a self-timed scheduling strategy .....	20
Figure 3.1.	One possible transaction order derived from the fully-static schedule .....	41
Figure 3.2.	Block diagram of the OMA prototype .....	49
Figure 3.3.	Modified design.....	50
Figure 3.4.	Details of the “TA” line mechanism (only one processor is shown) . .....	51
Figure 3.5.	Top-level schematic of the OMA prototype.....	54
Figure 3.6.	Using processor bus arbitration signals for controlling bus access.	56
Figure 3.7.	Ordered Transaction Controller implementation .....	58
Figure 3.8.	Presetable counter implementation .....	59
Figure 3.9.	Host interface .....	61
Figure 3.10.	Processing element.....	62
Figure 3.11.	Xilinx configuration at run time .....	64
Figure 3.12.	Connecting multiple boards .....	67
Figure 3.13.	Schematics hierarchy of the four processor OMA architecture .....	68
Figure 3.14.	OMA prototype board photograph.....	69
Figure 3.15.	Steps required for downloading code ( <i>tcl</i> script <i>omaDoAll</i> ).....	70
Figure 3.16.	Hierarchical specification of the Karplus-Strong algorithm in 28 voices.....	74
Figure 3.17.	Four processor schedule for the Karplus-Strong algorithm in 28 voices. Three processors are assigned 8 voices each, the fourth (Proc 1) is assigned 4 voices along with the noise source. ....	75
Figure 3.18.	(a) Hierarchical block diagram for a 15 band analysis and synthesis filter bank. (b) Schedule on four processors (using Sih’s DL heuristic [Sih90]).....	77

Figure 3.19.	Schedule for the FFT example. ....	78
Figure 4.1.	Fully-static schedule on five processors .....	80
Figure 4.2.	Self-timed schedule .....	81
Figure 4.3.	Schedule evolution when the transaction order of Fig. 3.1 is enforced .....	81
Figure 4.4.	The IPC graph for the schedule in Fig. 4.1. ....	83
Figure 4.5.	Transaction ordering constraints .....	89
Figure 4.6.	Modified schedule $S'$ .....	95
Figure 4.7.	$G_{ipc}$ , actor $C$ has execution time $t_C$ , constant over all invocations of $C$ .....	97
Figure 4.8.	$T_{ST}(t_C)$ .....	98
Figure 4.9.	$G_{ipc}$ with transaction ordering constraints represented as dashed lines .....	105
Figure 4.10.	$T_{ST}(t_C)$ and $T_{OT}(t_C)$ .....	105
Figure 5.1.	(a) An HSDFG (b) A three-processor self-timed schedule for (a). (c) An illustration of execution under the placement of barriers. ....	110
Figure 5.2.	Self-timed execution .....	113
Figure 5.3.	An IPC graph with a feedforward edge: (a) original graph (b) impos- ing bounded buffers. ....	115
Figure 5.4.	$x_2$ is an example of a redundant synchronization edge. ....	124
Figure 5.5.	An algorithm that optimally removes redundant synchronization edges .....	127
Figure 5.6.	(a) A multi-resolution QMF filter bank used to illustrate the benefits of removing redundant synchronizations. (b) The precedence graph for (a). (c) A self-timed, two-processor, parallel schedule for (a). (d) The initial synchronization graph for (c) .....	130
Figure 5.7.	The synchronization graph of Fig. 5.6(d) after all redundant synchro- nization edges are removed. ....	132
Figure 5.8.	An algorithm for converting a synchronization graph that is not	

	strongly connected into a strongly connected graph. ....	133
Figure 5.9.	An illustration of a possible solution obtained by algorithm Convert-to-SC-graph. ....	134
Figure 5.10.	The synchronization graph, after redundant synchronization edges are removed, induced by a four-processor schedule of a music synthesizer based on the Karplus-Strong algorithm. ....	136
Figure 5.11.	A possible solution obtained by applying Convert-to-SC-graph to the example of Figure 5.10.....	137
Figure 5.13.	An example used to illustrate a solution obtained by algorithm DetermineDelays.....	138
Figure 5.12.	An algorithm for determining the delays on the edges introduced by algorithm Convert-to-SC-graph. ....	139
Figure 5.14.	An example of resynchronization. ....	143
Figure 5.15.	The complete synchronization optimization algorithm.....	145
Figure 6.1.	BDF actors SWITCH and SELECT .....	148
Figure 6.2.	(a) Conditional (if-then-else) dataflow graph. The branch outcome is determined at run time by actor B. (b) Graph representing data-dependent iteration. The termination condition for the loop is determined by actor D. ....	149
Figure 6.3.	Acyclic precedence graphs corresponding to the if-then-else graph of Fig. 6.2. (a) corresponds to the TRUE assignment of the control token, (b) to the FALSE assignment. ....	150
Figure 6.4.	Quasi-static schedule for a conditional construct (adapted from [Lee88b]).....	152
Figure 6.5.	Programs on three processors for the quasi-static schedule of Fig. 6.4.....	153
Figure 6.6.	Transaction order corresponding to the TRUE and FALSE branches . ....	155
Figure 6.7.	Bus access list that is stored in the schedule RAM for the quasi-static schedule of Fig. 6.6. Loading operation of the schedule counter con-	

	ditioned on value of $c$ is also shown.....	157
Figure 6.8.	Conditional constructs in parallel paths .....	158
Figure 6.9.	A bus access mechanism that selectively “masks” bus grants based on values of control tokens that are evaluated at run time .....	159
Figure 6.10.	Bus access lists and the annotated list corresponding to Fig. 6.6..	161
Figure 6.11.	Quasi-static schedule for the data-dependent iteration graph of Fig. 6.2(b). .....	164
Figure 6.12.	A possible access order list corresponding to the quasi-static schedule of Fig. 6.11. ....	165
Figure 7.1.	An example of how execution time guarantees can be used to reduce buffer size bounds. ....	168

## ACKNOWLEDGEMENTS

I have always considered it a privilege to have had the opportunity of pursuing my Ph.D. at Berkeley. The time I have spent here has been very fruitful, and I have found the interaction with the exceptionally distinguished faculty and the smart set of colleagues extremely enriching. Although I will not be able to acknowledge all the people who have directly or indirectly helped me during the course of my Ph. D., I wish to mention some of the people who have influenced me most during my years as a graduate student.

First and foremost, I wish to thank Professor Edward Lee, my research advisor, for his valuable support and guidance, and for having been a constant source of inspiration for this work. I really admire Professor Lee's dedication to his research; I have learned a lot from his approach of conducting research.

I also thank Professors Pravin Varaiya and Henry Helson for serving on my thesis committee. I thank Professor Messerschmitt for his advice; I have learned from him, both in the classroom as well as through his insightful and humorous "when I was at Bell Labs ..." stories at our Friday afternoon post-seminar get-togethers. I have also greatly enjoyed attending classes and discussions with Professors Avidah Zakhor, Jean Walrand, John Wawrzynek, and Robert Brayton.

During the course of my Ph. D. research I have had the opportunity to work closely with several fellow graduate students. In particular I would like to mention Shuvra Bhattacharyya, in collaboration with whom some of the work in this thesis was done, and Praveen Murthy. Praveen and Shuvra are also close friends and I have immensely enjoyed my interactions with them, both technical as well as non-technical (such as music, photography, tennis, etc.).

I want to thank Phil Lapsley, who helped me with the DSP lab hardware when I first joined the DSP group; Soonhoi Ha, who helped me with various aspects of the scheduling implementation in Ptolemy; and Mani Srivastava, who helped me a great deal with printed circuit board layout tools, and provided me with several useful tips that helped me design and prototype the 4 processor OMA board.

I should mention Mary Stewart and Carol Sitea for helping me with reimbursements and other bureaucratic paperwork, Christopher Hylands for patiently answering my system related queries, and Heather Levien for cheerfully helping me with the mass of graduate division related paperwork, deadlines, formalities to be completed, etc.

I have enjoyed many useful discussions with some of some of my friends and colleagues, in particular Alan Kamas (I have to mention his infectious sense of humor), Louis Yun, Wan-teh Chan, Rick Han, William Li, Tom Parks, Jose Pino, Brian Evans, Mike Williamson, Bilung Lee and Asawaree Kalavade, who have made my (innumerable) hours in Cory Hall much more fun than what would have been otherwise. I will miss the corridor/elevator discussions (on topics ranging from the weather to Hindu philosophy) with Sriram Krishnan (the *other* Sriram), Jagesh Sanghavi, Rajeev Murgai, Shankar Narayanaswami, SKI, Angela Chuang, Premal Buch; and so will I miss the discussions, reminiscences and retelling of old tales with the sizable gang of graduate students in Berkeley and Stanford with whom I share my alma mater (IIT Kanpur) — Vigyan, Adnan, Kumud, Sunil, Amit Narayan, Geetanjali, Sanjay, Vineet, Ramesh, to name a few.

While at Berkeley, I have met several people who have since become good friends: Juergen Teich, Raghuram Devarakonda, Amit Lal, Amit Marathe, Ramesh Gopalan, Datta Godbole, Satyajit Patwardhan, Aparna Pandey, Amar Kapadia. I

thank them all for their excellent company; I have learned a lot from their talents and experiences as well.

I also wish to thank my long time friends Anurag, Ashish, Akshay, Anil, Kumud, Nitin, RD, Sanjiv — our occasional get-togethers and telephone chats have always provided a welcome relief from the tedium that grad school sometimes tends to become.

Of course, the Berkeley experience in general — the beautiful campus with great views of the San Francisco bay and the Golden Gate, the excellent library system, the cafe's and the restaurants, the CD shops and the used book stores, student groups and cacophonous drummers on Sproul plaza, the Hateman and the Naked Guy — has left me with indelible memories, and a wealth of interesting stories to tell, and has also helped keep my efforts towards a Ph. D. in perspective.

Finally, I wish to thank my parents for all their support and belief in me, and my sister, who has a knack for boosting my morale during rough times. I dedicate this thesis to them.



# 1

---

## INTRODUCTION

---

The focus of this thesis is the exploration of architectures and design methodologies for application-specific parallel systems for embedded applications in digital signal processing (DSP). The hardware model we consider consists of multiple programmable processors (possibly heterogeneous) and multiple application-specific hardware elements. Such a heterogeneous architecture is found in a number of embedded applications today: cellular radios, image processing boards, music/sound cards, robot control applications, etc. In this thesis we develop systematic techniques aimed at reducing inter-processor communication and synchronization costs in such multiprocessors that are designed to be application-specific. The techniques presented in this thesis apply to DSP algorithms that involve simple control structure; the precise domain of applicability of these techniques will be formally stated shortly.

Applications in signal processing and image processing require large computing power and have real-time performance requirements. The computing engines in such applications tend to be embedded as opposed to general-purpose. Custom VLSI implementations are usually preferred in such high throughput applications. However, custom approaches have the well known problems of long design cycles (the advances in high-level VLSI synthesis notwithstanding) and low flexibility in the final implementation. Programmable solutions are attractive

in both these respects: the programmable core needs to be verified for correctness only once, and design changes can be made late in the design cycle by modifying the software program. Although verifying the embedded software to be run on a programmable part is also a hard problem, in most situations changes late in the design cycle (and indeed even after the system design is completed) are much easier and cheaper to make in the case of software than in the case of hardware.

Special processors are available today that employ an architecture and an instruction set tailored towards signal processing. Such software programmable integrated circuits are called “Digital Signal Processors” (DSP chips or DSPs for short). The special features that these processors employ are discussed by Lee in [Lee88a]. However, a single processor — even DSPs — often cannot deliver the performance requirement of some applications. In these cases, use of multiple processors is an attractive solution, where both the hardware and the software make use of the application-specific nature of the task to be performed.

Over the past few years several companies have been offering boards consisting of multiple DSP chips. More recently, semiconductor companies are offering chips that integrate multiple CPUs on a single die: Texas Instruments (the TMS320C80 multi-DSP), Star Semiconductors (SPROC chip), Adaptive Solutions (CNAPS processor), etc. Multiple processor DSPs are becoming popular because of variety of reasons. First, VLSI technology today enables one to “stamp” 4-5 standard DSPs onto a single die; this trend is only going to continue in the coming years. Such an approach is expected to become increasingly attractive because it reduces the testing time for the increasingly complex VLSI systems of the future. Second, since such a device is programmable, tooling and testing costs of building an ASIC (application-specific integrated circuit) for each different application are saved by using such a device for many different applications, a situation that is going to be increasingly important in the future with up to a tenfold improvement in integration. Third, although there has been reluctance in adopting automatic compilers for embedded DSP processors, such parallel DSP products make the use of automated tools feasible; with a large number of processors per chip, one can

afford to give up some processing power to the inefficiencies in the automatic tools. In addition new techniques are being researched to make the process of automatically mapping a design onto multiple processors more efficient — this thesis is also an attempt in that direction. This situation is analogous to how logic designers have embraced automatic logic synthesis tools in recent years — logic synthesis tools and VLSI technology have improved to the point that the chip area saved by manual design over automated design is not worth the extra design time involved: one can afford to “waste” a few gates, just as one can afford to waste processor cycles to compilation inefficiencies in a multiprocessor DSP.

Finally, there are embedded applications that are becoming increasingly important for which programmability is in fact indispensable; set-top boxes capable of recognizing a variety of audio/video formats and compression standards, multimedia workstations that are required to run a variety of different multimedia software products, programmable audio/video codecs, etc.

The generalization of such a multiprocessor chip is one that has a collection of programmable processors as well as custom hardware on a single chip. Mapping applications onto such an architecture is then a hardware/software code-sign problem. The problems of inter-processor communication and synchronization are identical to the homogeneous multiprocessor case. In this thesis when we refer to a “multiprocessor” we will imply a heterogeneous architecture that may be comprised of different types of programmable processors and may include custom hardware elements too. All the techniques we present here apply to such a general system architecture.

Why study application-specific parallel processing in the first place instead of applying the ideas in general purpose parallel systems to the specific application? The reason is that general purpose parallel computation deals with a user-programmable computing device. Computation in embedded applications, however, is usually one-time programmed by the designer of that embedded system (a digital cellular radio handset for example) and is not meant to be programmable by the end user. The computation in embedded systems is specialized (the computa-

tion in a cellular radio handset involves specific DSP functions such as speech compression, channel equalization, modulation, etc.). Furthermore, embedded applications face very different constraints compared to general purpose computation: non-recurring design costs, power consumption, and real-time performance requirements are a few examples. Thus it is important to study techniques that are application-specific, and that make use of the special characteristics of the applications they target, in order to optimize for the particular metrics that are important for that specific application. These techniques adopt a design methodology that tailors the hardware and software implementation to the particular application. Some examples of such embedded computing systems are in robot controllers [Sriv92] and real-time speech recognition systems [Stolz91]; in consumer electronics such as future high-definition television sets, compact disk players, electronic music synthesizers and digital audio systems; and in communication systems such as digital cellular phones and base stations, compression systems for video-phones and video-conferencing, etc.

The idea of using multiple processing units to execute one program has been present from the time of the very first electronic computer in the nineteen forties. Parallel computation has since been the topic of active research in computer science. Whereas parallelism within a single processor has been successfully exploited (instruction-level parallelism), the problem of partitioning a single user program onto multiple such processors is yet to be satisfactorily solved. Instruction-level parallelism includes techniques such as pipelining (employed in traditional RISC processors), vectorization, VLIW (very large instruction word), superscalar — these techniques are discussed in detail by Patterson and Hennessy in [Patt90]. Architectures that employ multiple CPUs to achieve task-level parallelism fall into the shared memory, message passing, or dataflow paradigms. The Stanford DASH multiprocessor [Len92] is a shared memory machine whereas the Thinking Machines CM-5 falls into the message passing category. The MIT Monsoon machine [Pap90] is an example of a dataflow architecture.

Although the hardware for the design of such multiple processor machines

— the memory, interconnect network, IO, etc. — has received much attention, software for such machines has not been able to keep up with the hardware development. Efficient partitioning of a general program (written in C say) across a given set of processors arranged in a particular configuration is still an open problem. Detecting parallelism, the overspecified sequencing in popular imperative languages like C, managing overhead due to communication and synchronization between processors, and the requirement of dynamic load balancing for some programs (an added source of overhead) makes the partitioning problem for a general program hard.

If we turn away from general purpose computation to application-specific domains, however, parallelism is easier to identify and exploit. For example, one of the more extensively studied family of such application-specific parallel processors is the systolic array architecture [Kung88][Quin84][Rao85]; this architecture consists of regularly arranged arrays of processors that communicate locally, onto which a certain class of applications, specified in a mathematical form, can be systematically mapped. We discuss systolic arrays further in section 1.3.2.

The necessary elements in the study of application-specific computer architectures are: 1) a clearly defined set of problems that can be solved using the particular application-specific approach, 2) a formal mechanism for specification of these applications, and 3) a systematic approach for designing hardware from such a specification.

In this thesis, the applications we focus on are those that can be described by **Synchronous Dataflow Graphs (SDF)** [Lee87] and its extensions; we will discuss this model in detail shortly. SDF in its pure form can only represent applications that have no decision making at the task level. Extensions of SDF (such as the *Boolean dataflow* (BDF) model [Lee91][Buck93]) allow control constructs, so that data-dependent control flow can be expressed in such models. These models are significantly more powerful in terms of expressivity, but they give up some of the useful analytical properties that the SDF model has. For instance, Buck shows that it is possible to simulate any Turing machine in the BDF model [Buck93]. The

BDF model can therefore compute all Turing computable functions, whereas this is not possible in the case of the SDF model. We discuss the Boolean dataflow model further in Chapter 6.

In exchange for the limited expressivity of an SDF representation, we can efficiently check conditions such as whether a given SDF graph deadlocks, and whether it can be implemented using a finite amount of memory. No such general procedures can be devised for checking the corresponding conditions (deadlock behaviour and bounded memory usage) for a computation model that can simulate any given Turing machine. This is because the problems of determining if any given Turing machine halts (the halting problem), and determining whether it will use less than a given amount of memory (or tape) are *undecidable* [Lew81]; that is, no general algorithm exists to solve these problems in finite time.

In this thesis we will first focus on techniques that apply to SDF applications, and we will propose extensions to these techniques for applications that can be specified essentially as SDF, but augmented with a limited number of control constructs (and hence fall into the BDF model). SDF has proven to be a useful model for representing a significant class of DSP algorithms; several DSP tools have been designed based on the SDF and closely related models. Examples of commercial tools based on SDF are the Signal Processing Worksystem (SPW), developed by Comdisco Systems (now the Alta group of Cadence Design Systems) [Pow92][Barr91]; and COSSAP, developed by Cadis in collaboration with Meyr's group at Aachen University [Ritz92]. Tools developed at various universities that use SDF and related models include Ptolemy [Pin95a], the Warp compiler [Prin92], DESCARTES [Ritz92], GRAPE [Lauw90], and the Graph Compiler [Veig90].

The SDF model is popular because it has certain analytical properties that are useful in practice; we will discuss these properties and how they arise in the following section. The property most relevant for this thesis is that it is possible to effectively exploit parallelism in an algorithm specified in SDF by scheduling computations in the SDF graph onto multiple processors at compile or design time

rather than at run time. Given such a schedule that is determined at compile time, we can extract information from it with a view towards optimizing the final implementation. The main contribution of this thesis is to present techniques for minimizing synchronization and inter-processor communication overhead in statically (i.e. compile time) scheduled multiprocessors where the program is derived from a dataflow graph specification. The strategy is to model run time execution of such a multiprocessor to determine how processors communicate and synchronize, and then to use this information to optimize the final implementation.

## 1.1 The Synchronous Dataflow model

### 1.1.1 Background

Dataflow is a well-known programming model in which a program is represented as a directed graph, where the vertices (or actors) represent computation and edges (or arcs) represent FIFO (first-in-first-out) queues that direct data values from the output of one computation to the input of another. Edges thus represent data precedences between computations. Actors consume data (or tokens) from their inputs, perform computation on them (fire), and produce certain number of tokens on their outputs.

Programs written in high-level functional languages such as pure LISP, and in dataflow languages such as Id and Lucid can be directly converted into dataflow graph representations; such a conversion is possible because these languages are designed to be *free of side-effects*, i.e. programs in these languages are not allowed to contain global variables or data structures, and functions in these languages cannot modify their arguments [Ack82]. Also, since it is possible to simulate any Turing machine in one of these languages, questions such as deadlock (or, equivalently, terminating behaviour) and determining maximum buffer sizes required to implement edges in the dataflow graph become undecidable. Several models based on dataflow with restricted semantics have been proposed; these models give up the descriptive power of general dataflow in exchange for proper-

ties that facilitate formal reasoning about programs specified in these models, and are useful in practise, leading to simpler implementation of the specified computation in hardware or software.

One such restricted model (and in fact one of the earliest graph based computation models) is the **computation graph** of Karp and Miller [Karp66]. In their seminal paper Karp and Miller establish that their computation graph model is *determinate*, i.e. the sequence of tokens produced on the edges of a given computation graph are unique, and do not depend on the order that the actors in the graph fire, as long as all data dependencies are respected by the firing order. The authors also provide an algorithm that, based on topological and algebraic properties of the graph, determines whether the computation specified by a given computation graph will eventually terminate. Because of the latter property, computation graphs clearly cannot simulate all Turing machines, and hence are not as expressive as a general dataflow language like Lucid or pure LISP. Computation graphs provide some of the theoretical foundations for the SDF model.

Another model of computation relevant to dataflow is the **Petri net model** [Peter81][Mur89]. A Petri net consists of a set of *transitions*, which are analogous to actors in dataflow, and a set of *places* that are analogous to arcs. Each transition has a certain number of input places and output places connected to it. Places may contain one or more *tokens*. A Petri net has the following semantics: a transition *fires* when all its input places have one or more tokens and, upon firing, it produces a certain number of tokens on each of its output places.

A large number of different kinds of Petri net models have been proposed in the literature for modeling different types of systems. Some of these Petri net models have the same expressive power as Turing machines: for example if transitions are allowed to possess “inhibit” inputs (if a place corresponding to such an input to a transition contains a token, then that transition is not allowed to fire) then a Petri net can simulate any Turing machine (pp. 201 in [Peter81]). Others (depending on topological restrictions imposed on how places and transitions can be interconnected) are equivalent to finite state machines, and yet others are simi-



lar to SDF graphs. Some extended Petri net models allow a notion of time, to model execution times of computations. There is also a body of work on stochastic extensions of timed Petri nets that are useful for modeling uncertainties in computation times. We will touch upon some of these Petri net models again in Chapter 4. Finally, there are Petri nets that distinguish between different classes of tokens in the specification (*colored* Petrinets), so that tokens can have information associated with them. We refer to [Peter81] [Mur89] for details on the extensive variety of Petri nets that have been proposed over the years.

The particular restricted dataflow model we are mainly concerned with in this thesis is the SDF — Synchronous Data Flow — model proposed by Lee and Messerschmitt [Lee87]. The SDF model poses restrictions on the firing of actors: the number of tokens produced (consumed) by an actor on each output (input) edge is a fixed number that is known at compile time. The arcs in an 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; this concept will be formalized in the following chapter. In an actual implementation, arcs represent buffers in physical memory.

DSP applications typically represent computations on an indefinitely long data sequence; therefore the SDF graphs we are interested in for the purpose of signal processing must execute in a nonterminating fashion. Consequently, we must be able to obtain periodic schedules for SDF representations, which can then be run as infinite loops using a finite amount of physical memory. Unbounded buffers imply a sample rate inconsistency, and deadlock implies that all actors in the graph cannot be iterated indefinitely. Thus for our purposes, correctly constructed SDF graphs are those that can be scheduled periodically using a finite amount of memory. The main advantage of imposing restrictions on the SDF model (over a general dataflow model) lies precisely in the ability to determine whether or not an arbitrary SDF graph has a periodic schedule that neither deadlocks nor requires unbounded buffer sizes [Lee87]. The buffer sizes required to implement arcs in SDF graphs can be determined at compile time (recall that this

is not possible for a general dataflow model); consequently, buffers can be allocated statically, and run time overhead associated with dynamic memory allocation is avoided. The existence of a periodic schedule that can be inferred at compile time implies that a correctly constructed SDF graph entails no run time scheduling overhead.

An SDF graph in which every actor consumes and produces only one token from each of its inputs and outputs is called a **homogeneous SDF graph (HSDFG)**. An HSDF graph actor fires when it has one or more tokens on all its input edges; it consumes one token from each input edge when it fires, and produces one token on all its output edges when it completes execution. A general (*multirate*) SDF graph can always be converted into an HSDF graph [Lee86]; this transformation may result in an exponential increase in the number of actors in the final HSDF graph (see [Pin95b] for an example of an SDF graph in which this blowup occurs). Such a transformation, however, appears to be necessary when constructing periodic multiprocessor schedules from multirate SDF graphs. There is some recent work on reducing the complexity of the HSDFG that results from transforming a given SDF graph by applying graph clustering techniques to that SDF graph [Pin95b]. Since we are concerned with multiprocessor schedules in this thesis, we assume we start with an application represented as a homogeneous SDF graph henceforth, unless we state otherwise. This of course results in no loss of generality because a multirate graph is converted into a homogeneous graph for the purposes of multiprocessor scheduling anyway. In Chapter 6 we discuss how the ideas that apply to HSDF graphs can be extended to graphs containing actors that display data-dependent behaviour (i.e. *dynamic* actors).

We note that an HSDFG is very similar to a **marked graph** in the context of Petri nets [Peter81]; transitions in the marked graph correspond to actors in the HSDFG, places correspond to edges, and initial tokens (or initial marking) of the marked graph correspond to initial tokens (or delays) in HSDFGs. We will represent delays using bullets (•) on the edges of the HSDFG; we indicate more than one delay on an edge by a number alongside the bullet, as in Fig. 1.1(a).

SDF should not be confused with **synchronous languages** [Hal93][Ben91] (e.g. LUSTRE, SIGNAL, and ESTEREL), which have very different semantics from SDF. Synchronous languages have been proposed for formally specifying and modeling reactive systems, i.e. systems that constantly react to stimuli from a given physical environment. Signal processing systems fall into the reactive category, and so do control and monitoring systems, communication protocols, man-machine interfaces, etc. In these languages variables are possibly infinite sequences of data of a certain type. Associated with each such sequence is a conceptual (and sometimes explicit) notion of a *clock signal*. In LUSTRE, each variable is explicitly associated with a clock, which determines the instants at which the value of that variable is defined. SIGNAL and ESTEREL do not have an explicit notion of a clock. The clock signal in LUSTRE is a sequence of Boolean values, and a variable in a LUSTRE program assumes its  $n$ th value when its corresponding clock takes its  $n$ th TRUE value. Thus we may relate one variable with another by means of their clocks. In ESTEREL, on the other hand, clock ticks are implicitly defined in terms of instants when the reactive system corresponding to an ESTEREL program receives (and reacts to) external events. All computations in synchronous language are defined with respect to these clocks.

In contrast, the term “synchronous” in the SDF context refers to the fact that SDF actors produce and consume fixed number of tokens, and these numbers are known at compile time. This allows us to obtain periodic schedules for SDF graphs such that the average rates of firing of actors are fixed relative to one another. We will not be concerned with synchronous languages in this thesis, although these languages have a close and interesting relationship with dataflow models used for specification of signal processing algorithms [Lee95].

### **1.1.2 Utility of dataflow for DSP**

As mentioned before, dataflow models such as SDF (and other closely related models) have proven to be useful for specifying applications in signal processing and communications, with the goal of both simulation of the algorithm at

the functional or behavioural level, and for synthesis from such a high level specification to a software description (e.g. a C program) or a hardware description (e.g. VHDL) or a combination thereof. The descriptions thus generated can then be compiled down to the final implementation, e.g. an embedded processor, or an ASIC.

One of the reasons for the popularity of such dataflow based models is that they provide a formalism for block-diagram based visual programming, which is a very intuitive specification mechanism for DSP; the expressivity of the SDF model sufficiently encompasses a significant class of DSP applications, including multi-rate applications that involve upsampling and downsampling operations. An equally important reason for employing dataflow is that such a specification exposes parallelism in the program. It is well known that imperative programming styles such as C and FORTRAN tend to over-specify the control structure of a given computation, and compilation of such specifications onto parallel architectures is known to be a hard problem. Dataflow on the other hand imposes minimal data-dependency constraints in the specification, potentially enabling a compiler to detect parallelism. The same argument holds for hardware synthesis, where it is important to be able to exploit concurrency.

The SDF model has also proven useful for compiling DSP applications on single processors. Programmable digital signal processing chips tend to have special instructions such as a single cycle multiply-accumulate (for filtering functions), modulo addressing (for managing delay lines), bit-reversed addressing (for FFT computation); DSP chips also contain built in parallel functional units that are controlled from fields in the instruction (such as parallel moves from memory to registers combined with an ALU operation). It is difficult for automatic compilers to optimally exploit these features; executable code generated by commercially available compilers today utilizes one and a half to two times the program memory that a corresponding hand optimized program requires, and results in two to three times higher execution time compared to hand-optimized code [Zivo95]. There has been some recent work on compilation techniques for embedded software target-

ted towards DSP processors and microcontrollers [Liao95]; it is still too early to determine the impact of these techniques on automatic compilation for large-scale DSP/control applications, however.

Block diagram languages based on models such as SDF have proven to be a bridge between automatic compilation and hand coding approaches; a library of reusable blocks in a particular programming language is hand coded, this library then constitutes the set of atomic SDF actors. Since the library blocks are reusable, one can afford to carefully optimize and fine tune them. The atomic blocks are fine to medium grain in size; an atomic actor in the SDF graph may implement anything from a filtering function to a two input addition operation. The final program is then automatically generated by concatenating code corresponding to the blocks in the program according to the sequence prescribed by a schedule. This approach is mature enough that there are commercial tools available today, for example the SPW and COSSAP tools mentioned earlier, that employ this technique. Powerful optimization techniques have been developed for generating sequential programs from SDF graphs that optimize for metrics such as memory usage [Bhat94].

Scheduling is a fundamental operation that must be performed in order to implement SDF graphs on both uniprocessor as well as multiprocessors. Uniprocessor scheduling simply refers to determining the sequence of execution of actors such that all precedence constraints are met and all the buffers between actors (corresponding to arcs) return to their initial states. We discuss the issues involved in multiprocessor scheduling next.

## 1.2 Parallel scheduling

We recall that in the execution of a dataflow graph, actors fire when sufficient number of tokens are present at their inputs. The task of scheduling such a graph onto multiple processing units therefore involves assigning actors in the HSDFG to processors (the **processor assignment** step), ordering execution of these actors on each processor (the **actor ordering** step), and determining when each

actor fires such that all data precedence constraints are met. Each of these three tasks may be performed either at run time (a dynamic strategy) or at compile time (static strategy). We restrict ourselves to **non-preemptive** schedules, i.e. schedules where an actor executing on a processor can not be interrupted in the middle of its execution to allow another task to be executed. This is because preemption entails a significant implementation overhead and is therefore of limited use in embedded, time-critical applications.

Lee and Ha [Lee89] propose a scheduling taxonomy based on which of the scheduling tasks are performed at compile time and which at run time; we use the same terminology in this thesis. To reduce run time computation costs it is advantageous to perform as many of the three scheduling tasks as possible at compile time, especially in the context of algorithms that have hard real-time constraints. Which of these can be effectively performed at compile time depends on the information available about the execution time of each actor in the HSDFG.

For example, dataflow computers first pioneered by Dennis [Denn80] perform the assignment step at compile time, but employ special hardware (the *token-match* unit) to determine, at runtime, when actors assigned to a particular processor are ready to fire. The runtime overhead of token-matching and dynamic scheduling (within each processor) is fairly severe, so much so that dataflow architectures have not been commercially viable; even with expensive hardware support for dynamic scheduling, performance of such computers has been unimpressive.

The performance metric of interest for evaluating schedules is the **average iteration period  $T$** : the average time it takes for all the actors in the graph to be executed once. Equivalently, we could use the throughput  $T^{-1}$  (i.e. the number of iterations of the graph executed per unit time) as a performance metric. Thus an optimal schedule is one that minimizes  $T$ .

In this thesis we focus on scheduling strategies that perform both processor assignment and actor ordering at compile time, because these strategies appear to be most useful for a significant class of real time DSP algorithms. Although

assignment and ordering performed at run time would in general lead to a more flexible implementation (because a dynamic strategy allows for run time variations in computation load and for operations that display data dependencies) the overhead involved in such a strategy is usually prohibitive and real-time performance guarantees are difficult to achieve. Lee and Ha [Lee89] define two scheduling strategies that perform the assignment and ordering steps at compile time: **fully-static** and **self-timed**. We use the same terminology in this thesis.

### 1.2.1 Fully-static schedules

In the fully-static (FS) strategy, the exact firing time of each actor is also determined at compile time. Such a scheduling style is used in the design of systolic array architectures [Kung88], for scheduling VLIW processors [Lam88], and in high-level VLSI synthesis of applications that consist only of operations with guaranteed worst-case execution times [DeMich94]. Under a fully static schedule, all processors run in lock step; the operation each processor performs on each clock cycle is predetermined at compile time and is enforced at run time either implicitly (by the program each processor executes, perhaps augmented with “nop”s or idle cycles for correct timing) or explicitly (by means of a program sequencer for example).

A fully-static schedule of a simple HSDFG  $G$  is illustrated in Fig. 1.1. The FS schedule is schematically represented as a Gantt chart that indicates the processors along the vertical axis, and time along the horizontal axis. The actors are represented as rectangles with horizontal length equal to the execution time of the actor. The left side of each actor in the Gantt chart corresponds to its starting time. The Gantt chart can be viewed as a processor-time plane; scheduling can then be viewed as a mechanism to tile this plane while minimizing total schedule length and idle time (“empty spaces” in the tiling process). Clearly, the FS strategy is viable only if actor execution time estimates are accurate and data-independent or if tight worst-case estimates are available for these execution times.

As shown in Fig. 1.1, two different types of FS schedules arise, depending

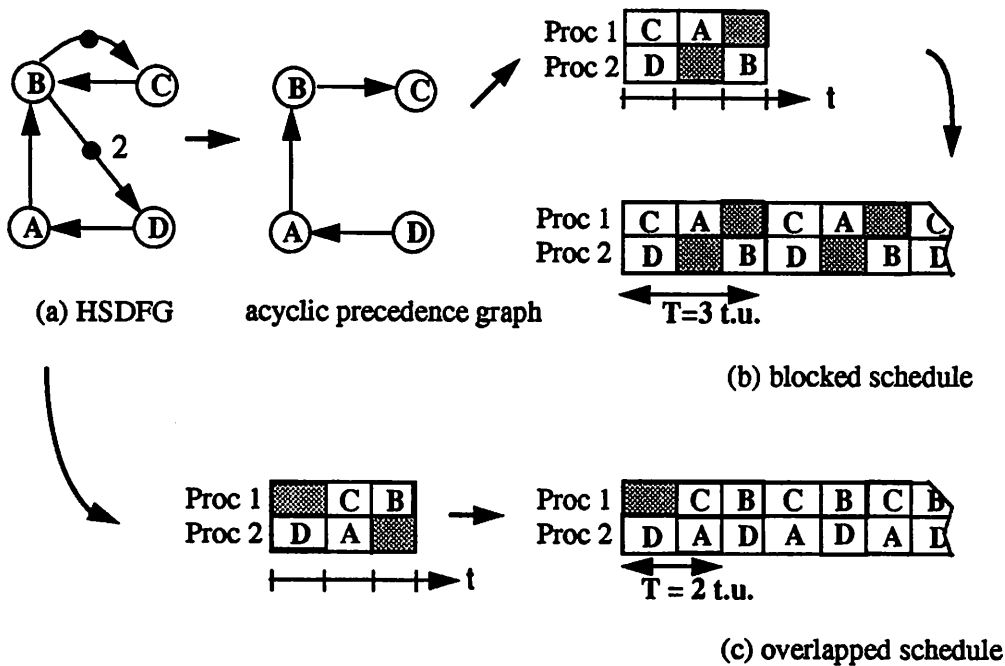


Figure 1.1. Fully static schedule

on how successive iterations of the HSDFG are treated. Execution times of all actors are assumed to be one time unit (t.u.) in this example. The FS schedule in Fig. 1.1(b) represents a **blocked schedule**: successive iterations of the HSDFG in a blocked schedule are treated separately so that each iteration is completed before the next one begins. A more elaborate blocked schedule on five processors is shown in Fig. 1.2. The HSDFG is scheduled as if it executes for only one iteration, i.e. inter-iteration dependencies are ignored; this schedule is then repeated to get an infinite periodic schedule for the HSDFG. The length of the blocked schedule determines the average iteration period  $T$ . The scheduling problem is then to obtain a schedule that minimizes  $T$  (which is also called the **makespan** of the schedule). A lower bound on  $T$  for a blocked schedule is simply the length of the **critical path** of the graph, which is the longest delay-free path in the graph.

Ignoring the inter-iteration dependencies when scheduling an HSDFG is equivalent to the classical multiprocessor scheduling problem for an Acyclic Precedence Graph (APG): the acyclic precedence graph is obtained from the given



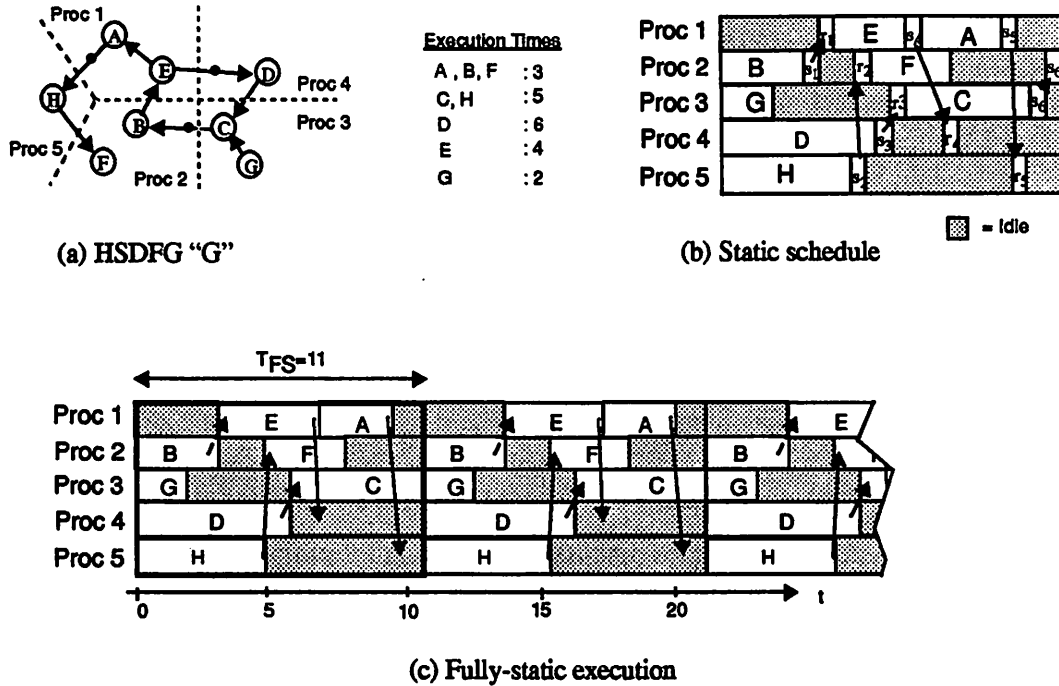


Figure 1.2. Fully-static schedule on five processors

HSDFG by eliminating all edges with delays on them (edges with delays represent dependencies across iterations) and replacing multiple edges that are directed between the same two vertices in the same direction with a single edge. This replacement is done because such multiple edges represent identical precedence constraints; these edges are taken into account individually during buffer assignment, however. Optimal multiprocessor scheduling of an acyclic graph is known to be NP-Hard [Garey79], and a number of heuristics have been proposed for this problem. One of the earliest, and still popular, solutions to this problem is *list scheduling*, first proposed by Hu [Hu61]. List scheduling is a greedy approach: whenever a task is ready to run, it is scheduled as soon as a processor is available to run it. Tasks are assigned priorities, and among the tasks that are ready to run at any instant, the task with the highest priority is executed first. Various researchers have proposed different priority mechanisms for list scheduling [Adam74], some of which use critical path based (CPM) methods [Ram72][Koh75][Blaz87] ([Blaz87] summarizes a large number of CPM based heuristics for scheduling).

The heuristics mentioned above ignore communication costs between processors, which is often inappropriate in actual multiprocessor implementations. An edge of the HSDFG that crosses processor boundaries after the processor assignment step represents interprocessor communication (IPC) (illustrated in Fig. 1.3(a)). These communication points are usually implemented using *send* and *receive* primitives that make use of the processor interconnect hardware. These primitives then have an execution cost associated with them that depends on the multiprocessor architecture and hardware being employed. Fully-static scheduling heuristics that take communication costs into account include [Sark89][Sih91][Prin91].

Computations in the HSDFG, however, are iterated essentially infinitely. The blocked scheduling strategies discussed thus far ignore this fact, and thus pay a penalty in the quality of the schedule they obtain. Two techniques that enable blocked schedules to exploit inter-iteration parallelism are **unfolding** and **retiming**. The unfolding strategy schedules  $J$  iterations of the HSDFG together, where  $J$  is called the **blocking factor**. Thus the schedule in Fig. 1.1(b) has  $J = 1$ . Unfolding often leads to improved blocked schedules (pp. 78-100 [Lee86], [Parhi91]), but it also implies a factor of  $J$  increase in program memory size and also in the size of the scheduling problem, which makes unfolding somewhat impractical.

Retiming involves manipulating delays in the HSDFG to reduce the critical path in the graph. This technique has been explored in the context of maximizing clock rates in synchronous digital circuits [Lei83], and has been proposed for improving blocked schedules for HSDFGs (“cutset transformations” in [Lee86], and [Hoang93]).

Fig. 1.1(c) illustrates an example of an **overlapped schedule**. Such a schedule is explicitly designed such that successive iterations in the HSDFG overlap. Obviously, overlapped schedules often achieve a lower iteration period than blocked schedules. In Fig. 1.1, for example, the iteration period for the blocked schedule is 3 units whereas it is 2 units for the overlapped schedule. One might

wonder whether overlapped schedules are fundamentally superior to blocked schedules with the unfolding and retiming operations allowed. This question is settled in the affirmative by Parhi and Messerschmitt [Parhi91]; the authors provide an example of an HSDFG for which no blocked schedule can be found, even allowing unfolding and retiming, that has a lower or equal iteration period than the overlapped schedule they propose.

Optimal resource constrained overlapped scheduling is of course NP-Hard, although a periodic overlapped schedule in the absence of processor constraints can be computed efficiently and optimally [Parhi91][Gasp92].

Overlapped scheduling heuristics have not been as extensively studied as blocked schedules. The main work in this area is by Lam [Lam88], and deGroot [deGroot92], who propose a modified list scheduling heuristic that explicitly constructs an overlapped schedule. Another work related to overlapped scheduling is the “cyclo-static scheduling” approach proposed by Schwartz. This approach attempts to optimally tile the processor-time plane to obtain the best possible schedule. The search involved in this process has a worst case complexity exponential in the size of the input graph, although it appears that the complexity is manageable in practice, at least for small examples [Schw85].

## 1.2.2 Self-timed schedules

The fully-static approach introduced in the previous section cannot be used when actors have variable execution times; the FS approach requires precise knowledge of actor execution times to guarantee sender-receiver synchronization. It is possible to use worst case execution times and still employ an FS strategy, but this requires tight worst case execution time estimates that may not be available to us. An obvious strategy for solving this problem is to introduce explicit synchronization whenever processors communicate. This leads to the self-timed scheduling (ST) strategy in the scheduling taxonomy of Lee and Ha [Lee89]. In this strategy we first obtain an FS schedule using the techniques discussed in section 1.2, making use of the execution time estimates. After computing the FS schedule (Fig. 1.3

(b)), we simply discard the timing information that is not required, and only retain the processor assignment and the ordering of actors on each processor as specified by the FS schedule (Fig. 1.3(c)). Each processor is assigned a sequential list of actors, some of which are *send* and *receive* actors, that it executes in an infinite loop. When a processor executes a communication actor, it synchronizes with the processor(s) it communicates with. Exactly when a processor executes each actor depends on when, at run time, all input data for that actor are available, unlike the fully-static case where no such run time check is needed. Conceptually, the processor sending data writes data into a FIFO buffer, and blocks when that buffer is full; the receiver on the other hand blocks when the buffer it reads from is empty. Thus flow control is performed at run time. The buffers may be implemented using shared memory, or using hardware FIFOs between processors. In a self-timed strategy, processors run sequential programs and communicate when they execute the communication primitives embedded in their programs, as shown schematically in Fig. 1.3 (c).

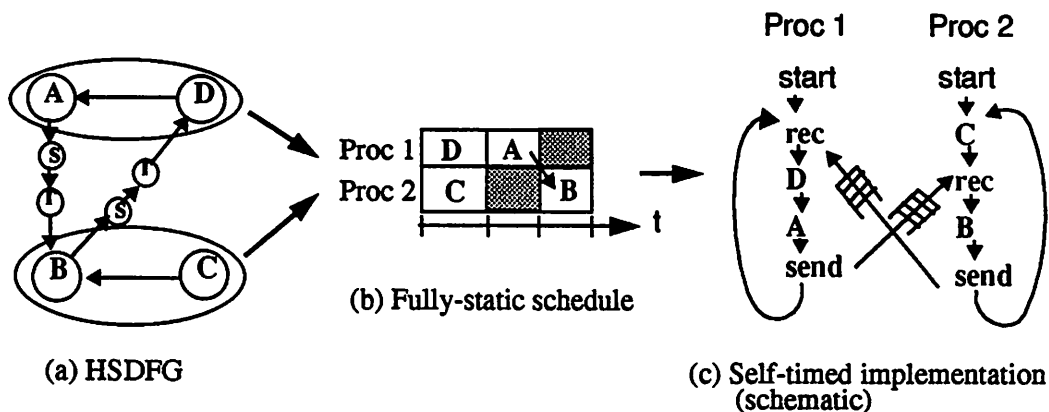


Figure 1.3. Steps in a self-timed scheduling strategy

An ST strategy is robust with respect to changes in execution times of actors, because sender-receiver synchronization is performed at run time. Such a strategy, however, implies higher IPC costs compared to the fully-static strategy because of the need for synchronization (e.g. using semaphore management). In addition the ST strategy faces arbitration costs: the FS schedule guarantees mutu-

ally exclusive access of shared communication resources, whereas shared resources need to be arbitrated at run time in the ST schedule. Consequently, whereas IPC in the FS schedule simply involves reading and writing from shared memory (no synchronization or arbitration needed), implying a cost of a few processor cycles for IPC, the ST strategy requires of the order of tens of processor cycles, unless special hardware is employed for run time flow control. We discuss in detail how this overhead arises in a shared bus multiprocessor configuration in Chapter 3.

Run time flow control allows variations in execution times of tasks; in addition, it also simplifies the compiler software, since the compiler no longer needs to perform detailed timing analysis and does not need to adjust the execution of processors relative to one another in order to ensure correct sender-receiver synchronization. Multiprocessor designs, such as the Warp array [Ann87][Lam88] and the 2-D MIMD (Multiple Instruction Multiple Data) array of [Ziss87], that could potentially use fully-static scheduling, still choose to implement such run time flow control (at the expense of additional hardware) for the resulting software simplicity. Lam presents an interesting discussion on the trade-off involved between hardware complexity and ease of compilation that ensues when we consider dynamic flow control implemented in hardware versus static flow control enforced by a compiler (pp. 50-68 of [Lam89]).

### **1.2.3 Execution time estimates and static schedules**

We assume we have reasonably good estimates of actor execution times available to us at compile time to enable us to exploit static scheduling techniques; however, these estimates need not be exact, and execution times of actors may even be data-dependent. Thus we allow actors that have different execution times from one iteration of the HSDFG to the next, as long as these variations are small or rare. This is typically the case when estimates are available for the task execution times, and actual execution times are close to the corresponding estimates with high probability, but deviations from the estimates of (effectively) arbitrary

magnitude occasionally occur due to phenomena such as cache misses, interrupts, user inputs or error handling. Consequently, tight worst-case execution time bounds cannot generally be determined for such operations; however, reasonably good execution time estimates can in fact be obtained for these operations, so that static assignment and ordering techniques are viable. For such applications self-timed scheduling is ideal, because the performance penalty due to lack of dynamic load balancing is overcome by the much smaller run time scheduling overhead involved when static assignment and ordering is employed.

The estimates for execution times of actors can be obtained by several different mechanisms. The most straightforward method is for the programmer to provide these estimates when he writes the library of primitive blocks. This strategy is used in the Ptolemy system, and is very effective for the assembly code libraries, in which the primitives are written in the assembly language of the target processor (Ptolemy currently supports the Motorola 56000 and 96000 processors). The programmer can provide a good estimate for blocks written in such a library by counting the number of processor cycles each instruction consumes, or by profiling the block on an instruction-set simulator.

It is more difficult to estimate execution times for blocks that contain control constructs such as data-dependent iterations and conditionals within their body, and when the target processor employs pipelining and caching. Also, it is difficult, if not impossible, for the programmer to provide reasonably accurate estimates of execution times for blocks written in a high-level language (as in the C code generation library in Ptolemy). The solution adopted in the GRAPE system [Lauw90] is to automatically estimate these execution times by compiling the block (if necessary) and running it by itself in a loop on an instruction-set simulator for the target processor. To take into account data-dependent execution behaviour, different input data sets can be provided for the block during simulation. Either the worst case or the average case execution time is used as the final estimate.

The estimation procedure employed by GRAPE is obviously time consum-

ing; in fact estimation turns out to be the most time consuming step in the GRAPE design flow. Analytical techniques can be used instead to reduce this estimation time; for example, Li and Malik [Li95] have proposed algorithms for estimating the execution time of embedded software. Their estimation technique, which forms a part of a tool called *cinderella*, consists of two components: 1) determining the sequence of instructions in the program that results in maximum execution time (program path analysis) and 2) modeling the target processor to determine how much time the worst case sequence determined in step 1 takes to execute (micro-architecture modeling). The target processor model also takes the effect of instruction pipelines and cache activity into account. The input to the tool is a generic C program with annotations that specify the loop bounds (i.e. the maximum number of iterations that a loop runs for). Although the problem is formulated as an integer linear program (ILP), the claim is that practical inputs to the tool can be efficiently analyzed using a standard ILP solver. The advantage of this approach, therefore, is the efficient manner in which estimates are obtained as compared to simulation.

It should be noted that the program path analysis component of the Li and Malik technique is in general an undecidable problem; therefore for these techniques to function, the programmer must ensure that his or her program does not contain pointer references, dynamic data structures, recursion, etc. and must provide bounds on all loops. Li and Malik's technique also depends on the accuracy of the processor model, although one can expect good models to eventually evolve for DSP chips and microcontrollers that are popular in the market.

The problem of estimating execution times of blocks is central for us to be able to effectively employ compile time design techniques. This problem is an important area of research in itself, and the strategies employed in Ptolemy and GRAPE, and those proposed by Li and Malik are useful techniques, and we expect better estimation techniques to be developed in the future.

## 1.3 Application-specific parallel architectures

There has been significant amount of research on general purpose high-performance parallel computers. These employ expensive and elaborate interconnect topologies, memory and Input/Output (I/O) structures. Such strategies are unsuitable for embedded DSP applications as we discussed earlier. In this section we discuss some application-specific parallel architectures that have been employed for signal processing, and contrast them to our approach.

### 1.3.1 Dataflow DSP architectures

There have been a few multiprocessors geared towards signal processing that are based on the dataflow architecture principles of Dennis [Denn80]. Notable among these are Hughes Data Flow Multiprocessor [Gau85], the Texas Instruments Data Flow Signal Processor [Grim84], and the AT&T Enhanced Modular Signal Processor [Bloch86]. The first two perform the processor assignment step at compile time (i.e. tasks are assigned to processors at compile time) and tasks assigned to a processor are scheduled on it dynamically; the AT&T EMPS performs even the assignment of tasks to processors at runtime.

Each one of these machines employs elaborate hardware to implement dynamic scheduling within processors, and employs expensive communication networks to route tokens generated by actors assigned to one processor to tasks on other processors that require these tokens. In most DSP applications, however, such dynamic scheduling is unnecessary since compile time predictability makes static scheduling techniques viable. Eliminating dynamic scheduling results in much simpler hardware without an undue performance penalty.

Another example of an application-specific dataflow architecture is the NEC  $\mu$ PD7281 [Chase84], which is a single chip processor geared towards image processing. Each chip contains one functional unit; multiple such chips can be connected together to execute programs in a pipelined fashion. The actors are statically assigned to each processor, and actors assigned to a given processor are



scheduled on it dynamically. The primitives that this chip supports, convolution, bit manipulations, accumulation, etc., are specifically designed for image processing applications.

### 1.3.2 Systolic and wavefront arrays

Systolic arrays consist of processors that are locally connected and may be arranged in different topologies: mesh, ring, torus, etc. The term “systolic” arises because all processors in such a machine run in lock-step, alternating between a computation step and a communication step. The model followed is usually SIMD (Single Instruction Multiple Data). Systolic arrays can execute a certain class of problems that can be specified as “Regular Iterative Algorithms (RIA)” [Rao85]; systematic techniques exist for mapping an algorithm specified in a RIA form onto dedicated processor arrays in an optimal fashion. Optimality includes metrics such as processor and communication link utilization, scalability with the problem size, achieving best possible speedup for a given number of processors, etc. Several numerical computation problems were found to fall into the RIA category: linear algebra, matrix operations, singular value decomposition, etc. (see [Kung88][Leigh92] for interesting systolic array implementations of a variety of different numerical problems). Only fairly regular computations can be specified in the RIA form; this makes the applicability of systolic arrays somewhat restrictive.

Wavefront arrays are similar to systolic arrays except that processors are not under the control of a global clock. Communication between processors is asynchronous or self-timed; handshake between processors ensures run time synchronization. Thus processors in a wavefront array can be complex and the arrays themselves can consist of a large number of processors without incurring the associated problems of clock skew and global synchronization. Again, similar to FS versus ST scheduling, the flexibility of wavefront arrays over systolic arrays comes at the cost of extra handshaking hardware.

The Warp project at Carnegie Mellon University [Anna87] is an example

of a programmable systolic array, as opposed to a dedicated array designed for one specific application. Processors are arranged in a linear array and communicate with their neighbors through FIFO queues. Programs are written for this computer in a language called W2 [Lam88]. The Warp project also led to the iWarp design [Bork88], which has a more elaborate inter-processor communication mechanism than the Warp machine. An iWarp node is a single VLSI component, composed of a computation engine and a communication engine; the latter consists of a crossbar and data routing mechanisms. The iWarp nodes can be connected in various single and two dimensional topologies, and point to point message-passing type communication is supported.

### **1.3.3 Multiprocessor DSP architectures**

In this section we discuss multiprocessors that make use of multiple off the shelf programmable DSP chips.

The SMART architecture [Koh90] is a reconfigurable bus design comprised of AT&T DSP32C processors, and custom VLSI components for routing data between processors. Clusters of processors may be connected onto a common bus, or may form a linear array with neighbor to neighbor communication. This allows the multiprocessor to be reconfigured depending on the communication requirement of the particular application being mapped onto it. Scheduling and code generation for this machine is done by the McDAS compiler [Hoang93].

The DSP3 multiprocessor [Shive92] was built at AT&T, and is comprised of DSP32C processors connected in a mesh configuration. The mesh interconnect is implemented using custom VLSI components for data routing. Each processor communicates with four of its adjacent neighbors through this router, which consists of input and output queues, and a crossbar that is configurable under program control. Data packets contain headers that indicate the ID of the destination processor.

The Ring Array Processor (RAP) system [Morg92] uses TI DSP320C30 processors connected in a ring topology. This system is designed specifically for

speech recognition applications based on artificial neural networks. The RAP system consists of several boards that are attached to a host workstation, and acts as a coprocessor for the host. The unidirectional pipelined ring topology employed for interprocessor communication was found to be ideal for the particular algorithms that were to be mapped to this machine. The ring structure is similar to the SMART array, except that no processor ID is included with the data, and processor reads and writes into the ring are scheduled in a fully-static fashion. The ring is used to broadcast data from one processor to all the others during one phase of the neural net algorithm, and is used to shift data from processor to processor in a pipelined fashion in the second phase.

The MUSIC system [Gunz92] uses Motorola DSP96000 processors, and has been designed for neural network simulations and scientific simulations. An “intelligent” communication network, implemented on Xilinx gate arrays, broadcasts data generated by any processing element (PE) to all the other PEs. The PEs are arranged in a ring topology. This kind of broadcast mechanism is suited to the applications the MUSIC system targets: the outputs from one layer of a multi-layer perceptron (a kind of neural net) is needed by all the “neurons” in the next layer, making broadcasting an ideal strategy when the different net layers are executed in a pipelined fashion. The molecular dynamics example the authors provide also benefits from this broadcast mechanism.

## **1.4 Thesis overview: our approach and contributions**

We argued that the self-timed scheduling strategy is suited towards parallel implementation for DSP. The multiprocessor architectures we discussed in the previous section support this argument: the dataflow architectures in section 1.3.1 use dynamic scheduling, but pay a high hardware cost, which makes them unsuited for embedded applications. In the case of the NEC dataflow chips, parallelism is mainly derived through pipelined execution. The dataflow model of execution that is implemented in hardware in this chip, although elegant, is of limited use for the

image processing applications that this part has been designed for; the order in which each processor executes instructions assigned to it can potentially be fixed at compile time without loss in parallelism.

Systolic arrays normally employ a fully-static strategy. As we discussed before, RIA specifications and the primarily SIMD approach used in systolic array mapping techniques restrict their domain of applicability. The approach taken by these techniques is to use a large number of very simple processors to perform the computation, whereas the approach we follow in this thesis is to use a small number of powerful processors. This enables us to handle algorithms specified as data-flow graphs where the actors are tasks with a potentially large granularity. The parallelism we employ is therefore at the task level (functional parallelism). Such a strategy gives up some of the optimality properties that systolic array mapping techniques guarantee in exchange for a larger application domain. Again, utilizing a number of pretested processor cores is economically more attractive than building a systolic array implementation from scratch.

Ideally we would like to exploit the strategy of partitioning data among different processors (data parallelism) that systolic techniques employ, along with task level parallelism. There has not been much work in this direction, although the work of Printz [Prin91], and the Multidimensional SDF model proposed by Lee in [Lee93], are two promising approaches for combining data and functional parallelism.

The multiple DSP machines we discussed in the last section all employ some form of self-timed scheduling. Clearly, general purpose parallel machines like the Thinking Machines CM-5 and Stanford Dash multiprocessor can also be programmed using the self-timed scheduling style, since these machines provide mechanisms for run time synchronization and flow control. These machines, however, do not attempt to make use of the fact that the interprocessor communication pattern in a self-timed implementation is fairly predictable. In this thesis we explore techniques that optimize the parallel implementation of a self-timed schedule by performing compile time analysis of the schedule to determine the pattern

# 2

---

## TERMINOLOGY AND NOTATIONS

---

In this chapter we introduce terminology and definitions used in the remainder of the thesis. We also formalize the scheduling concepts that were presented intuitively in the previous chapter.

### 2.1 HSDF graphs and associated graph theoretic notation

We represent an HSDFG by an ordered pair  $(V, E)$ , where  $V$  is the set of vertices (actors) and  $E$  is the set of edges. We refer to the source and sink vertices of a graph edge  $e$  by  $src(e)$  and  $snk(e)$ , and we denote the delay (or the number of initial tokens) on  $e$  by  $delay(e)$ . We say that  $e$  is an **output edge** of  $src(e)$ , and that  $e$  is an **input edge** of  $snk(e)$ . We will also use the notation  $(v_i, v_j)$ ,  $v_i, v_j \in V$ , for an edge directed from  $v_i$  to  $v_j$ .

A **path** in  $(V, E)$  is a finite, non-empty sequence  $(e_1, e_2, \dots, e_n)$ , where each  $e_i$  is a member of  $E$ , and  $snk(e_1) = src(e_2)$ ,  $snk(e_2) = src(e_3)$ , ...,  $snk(e_{n-1}) = src(e_n)$ . We say that the path  $p = (e_1, e_2, \dots, e_n)$  **contains** each  $e_i$  and each subsequence of  $(e_1, e_2, \dots, e_n)$ ;  $p$  is **directed from**  $src(e_1)$  to  $snk(e_n)$ ; and each member of  $\{src(e_1), src(e_2), \dots, src(e_n), snk(e_n)\}$  is **on**  $p$ . A path that is directed from a vertex to itself is called a **cycle**, and a **fundamen-**

**tal cycle** is a cycle of which no proper subsequence is a cycle.

If  $p = (e_1, e_2, \dots, e_n)$  is a path in an HSDFG, then we define the **path de-**

**lay** of  $p$ , denoted  $Delay(p)$ , by  $Delay(p) = \sum_{i=1}^n delay(e_i)$ . Since the delays on

all HSDFG edges are restricted to be non-negative, it is easily seen that between any two vertices  $x, y \in V$ , either there is no path directed from  $x$  to  $y$ , or there exists a (not necessarily unique) **minimum-delay path** between  $x$  and  $y$ . Given an HSDFG  $G$ , and vertices  $x, y$  in  $G$ , we define  $\rho_G(x, y)$  to be equal to the path delay of a minimum-delay path from  $x$  to  $y$  if there exist one or more paths from  $x$  to  $y$ , and equal to  $\infty$  if there is no path from  $x$  to  $y$ . If  $G$  is understood, then we may drop the subscript and simply write “ $\rho$ ” in place of “ $\rho_G$ ”.

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  $C_1$ . An SCC is a **source SCC** if it has no predecessor SCC; and an 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**.

Given two arbitrary sets  $S_1$  and  $S_2$ , we define the difference of these two sets by  $S_1 - S_2 = \{s \in S_1 \mid s \notin S_2\}$ , and we denote the number of elements in a finite set  $S$  by  $|S|$ . Also, if  $r$  is a real number, then we denote the smallest integer that is greater than or equal to  $r$  by  $\lceil r \rceil$ .

For elaboration on any of the graph-theoretic concepts presented in this section, we refer the reader to Cormen, Leiserson, and Rivest [Corm92].

## 2.2 Schedule notation

To model execution times of actors (and to perform static scheduling) we associate execution time  $t(v) \in Z^+$  (non-negative integer) with each actor  $v$  in the HSDFG;  $t(v)$  assigns execution time to each actor  $v$  (the actual execution time can be interpreted as  $t(v)$  cycles of a base clock). Inter-processor communication costs are represented by assigning execution times to the *send* and *receive* actors. The values  $t(v)$  may be set equal to execution time *estimates* when exact execution times are not available, in which case results of the computations that make use of these values (e.g. the iteration period  $T$ ) are compile time estimates.

Recall that actors in an HSDFG are executed essentially infinitely. Each firing of an actor is called an *invocation* of that actor. An *iteration* of the HSDFG corresponds to one invocation of every actor in the HSDFG. A schedule specifies processor assignment, actor ordering and firing times of actors, and these may be done at compile time or at run time depending on the scheduling strategy being employed. To specify firing times, we let the function  $start(v, k) \in Z^+$  represent the time at which the  $k$ th invocation of the actor  $v$  starts. Correspondingly, the function  $end(v, k) \in Z^+$  represents the time at which the  $k$ th execution of the actor  $v$  completes, at which point  $v$  produces data tokens at its output edges. Since we are interested in the  $k$ th execution of each actor for  $k = 0, 1, 2, 3, \dots$ , we set  $start(v, k) = 0$  and  $end(v, k) = 0$  for  $k < 0$  as the “initial conditions”. If the  $k$ th invocation of an actor  $v_j$  takes *exactly*  $t(v_j)$ , then we can claim:

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

Recall that a fully-static schedule specifies a processor assignment, actor ordering on each processor, and also the precise firing times of actors. We use the following notation for a fully-static schedule:

**Definition 2.1:** A fully-static schedule  $S$  (for  $P$  processors) specifies a triple:

$$S = \{ \sigma_p(v), \sigma_t(v), T_{FS} \},$$

where  $\sigma_p(v) \rightarrow [1, 2, \dots, P]$  is the processor assignment, and  $T_{FS}$  is the iteration period. An FS schedule specifies the firing times  $start(v, k)$  of all actors, and since we want a finite representation for an infinite schedule, an FS schedule is constrained to be periodic:

$$start(v, k) = \sigma_t(v) + kT_{FS},$$

$\sigma_t(v)$  is thus the starting time of the first execution of actor  $v$  (i.e.  $start(v, 0) = \sigma_t(v)$ ). Clearly, the throughput for such a schedule is  $T_{FS}^{-1}$ .

The  $\sigma_p(v)$  function and the  $\sigma_t(v)$  values are chosen so that all data precedence constraints and resource constraints are met. We define precedence constraints as follows:

**Definition 2.2:** An edge  $(v_j, v_i) \in E$  in an HSDFG  $(V, E)$  represents the (data) precedence constraint:

$$start(v_i, k) \geq end(v_j, k - delay((v_j, v_i))), \forall k \geq delay(v_j, v_i).$$

The above definition arises because each actor consumes one token from each of its input edges when it fires. Since there are already  $delay(e)$  tokens on each incoming edge  $e$  of actor  $v$ , another  $k - delay(e) - 1$  tokens must be produced on  $e$  before the  $k$ th execution of  $v$  can begin. Thus the actor  $src(e)$  must have completed its  $(k - delay(e) - 1)$ th execution before  $v$  can begin its  $k$ th execution. The “-1”s arise because we define  $start(v, k)$  for  $k \geq 0$  rather than  $k > 0$ .



This is done for notational convenience.

Any schedule that satisfies all the precedence constraints specified by edges in an HSDFG  $G$  is also called an **admissible schedule** for  $G$  [Reit68]. A **valid execution** of an HSDFG corresponds to a set of firing times  $\{start(v_p, k)\}$  that correspond to an admissible schedule, i.e. a valid execution respects all data precedences specified by the HSDFG.

For the purposes of the techniques presented in this thesis, we are only interested in the precedence relationships between actors in the HSDF graph. In a general HSDFG one or more pairs of vertices can have multiple edges connecting them in the same “direction.” Such a situation often arises when a multirate SDF graph is converted into a homogeneous HSDFG. Multiple edges between the same pair of vertices in the same direction are redundant as far as precedence relationships are concerned. Suppose there are multiple edges from vertex  $v_i$  to  $v_j$ , and amongst these edges the edge that has minimum delay has delay equal to  $d_{min}$ . Then, if we replace all these edges by a single edge with delay equal to  $d_{min}$ , it is easy to verify that this single edge maintains the precedence constraints for *all* the edges that were directed from  $v_i$  to  $v_j$ . Thus a general HSDF graph may be preprocessed into a form where the source and sink vertices uniquely identify an edge in the graph, and we may represent an edge  $e \in E$  by the ordered pair  $(src(e), snk(e))$ . The multiple edges are taken into account individually when buffers are assigned to the arcs in the graph.

As we discussed in section 1.2.1, in some cases it is advantageous to *unfold* a graph by a certain unfolding factor, say  $u$ , and schedule  $u$  iterations of the graph together in order to exploit inter-iteration parallelism more effectively. The unfolded graph contains  $u$  copies of each actor of the original graph. In this case  $\sigma_p$  and  $\sigma_t$  are defined for all the vertices of the *unfolded* graph (i.e.  $\sigma_p$  and  $\sigma_t$  are defined for  $u$  invocations of each actor);  $T_{FS}$  is the iteration period for the unfolded graph,

and the average iteration period for the original graph is then  $\frac{T_{FS}}{u}$ . In the remainder of this thesis, we assume we are dealing with the unfolded graph and we refer only to the iteration period and throughput of the unfolded graph, if unfolding is in fact employed, with the understanding that these quantities can be scaled by the unfolding factor to obtain the corresponding quantities for the original graph.

In a self-timed scheduling strategy, we determine a fully-static schedule,  $\{\sigma_p(v), \sigma_t(v), T_{FS}\}$ , using the execution time estimates, but we only retain the processor assignment  $\sigma_p$  and the ordering of actors on each processor as specified by  $\sigma_t$ , and discard the precise timing information specified in the fully-static schedule. Although we may start out with setting  $start(v, 0) = \sigma_t(v)$ , the subsequent  $start(v, k)$  values are determined at runtime based on availability of data at the input of each actor; the average iteration period of a self-timed schedule is represented by  $T_{ST}$ . We analyze the evolution of a self-timed schedule further in Chapter 4.

# 3

---

## THE ORDERED TRANSACTION STRATEGY

---

The self-timed scheduling strategy in Chapter 1 introduces synchronization checks when processors communicate; such checks permit variations in actor execution times, but they also imply run time synchronization and arbitration costs. In this chapter we present a hardware architecture approach called Ordered Transactions (OT) that alleviates some of these costs, and in doing so, trades off some of the run time flexibility afforded by the ST approach. The ordered transactions strategy was first proposed by Bier, Lee, and Sriram [Lee90][Bier90]. In this chapter we describe the idea behind the OT approach and then we discuss the design and hardware implementation of a shared-bus multiprocessor that makes use of this strategy to achieve a low-cost interprocessor communication using simple hardware. The software environment for this board is provided by the Ptolemy system developed at the University of California at Berkeley [Buck94][Ptol94].

### 3.1 The Ordered Transactions strategy

In the OT strategy we first obtain a fully-static schedule using the execution time estimates, but we discard the precise timing information specified in the fully-static schedule; as in the ST schedule we retain the processor assignment ( $\sigma_p$ ) and actor ordering on each processor as specified by  $\sigma_i$ ; in addition, we also

retain the order in which processors communicate with one another and we enforce this order at run time. We formalize the concept of transaction order below.

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

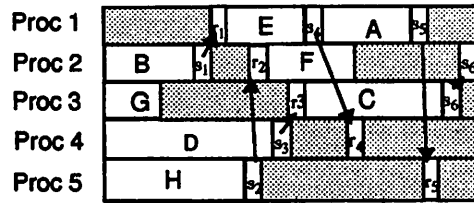
Thus the OT schedule is essentially an ST schedule with the added transaction order constraints specified by  $O$ .

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

$$\sigma_t(v_1) \leq \sigma_t(v_2) \leq \dots \leq \sigma_t(v_{2k-1}) \leq \sigma_t(v_{2k}).$$

The transaction order can therefore be determined by sorting the set of communication actors  $(S \cup R)$  according to their start times  $\sigma_t$ . Fig. 3.1 shows an example of how such an order could be derived from a given fully-static schedule. This FS schedule corresponds to the HSDFG and schedule illustrated in Chapter 1 (Fig. 1.2).

The transaction order is enforced at run time by a controller implemented in hardware. The main advantage of ordering inter-processor transactions is that it



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

Figure 3.1. One possible transaction order derived from the fully-static schedule

allows us to restrict access to communication resources statically, based on the communication pattern determined at compile time. Since communication resources are typically shared between processors, run time contention for these resources is eliminated by ordering processor accesses to them; this results in an efficient IPC mechanism at low hardware cost. We have built a prototype four processor DSP board, called the Ordered Memory Access (OMA) architecture, that demonstrates the ordered transactions concept. The OMA prototype board utilizes shared memory and a single shared bus for IPC — the sender writes data to a particular shared memory location that is allocated at compile time, and the receiver reads that location. In this multiprocessor, a very simple controller on the board enforces the pre-determined transaction order at run time, thus eliminating the need for run time bus arbitration or semaphore synchronization. This results in efficient IPC (comparable to the FS strategy) at relatively low hardware cost. As in the ST scenario, the OT strategy is tolerant of variations in execution times of actors, because the transaction order enforces correct sender-receiver synchronization; however, this strategy is more constrained than ST scheduling, which allows the order in which communication actors fire to vary at run time. The ordered transactions strategy, therefore, falls in between fully-static and self-timed strategies in that, like the ST strategy, it is tolerant of variations in execution times and, like the FS strategy, has low communication and synchronization costs. These performance issues will be discussed quantitatively in the following chapter; the rest

of this chapter describes the hardware and software implementation of the OMA prototype.

## 3.2 Shared bus architecture

The OMA architecture uses a single shared bus and shared memory for inter-processor communication. This kind of shared memory architecture is attractive for embedded multiprocessor implementations owing to its relative simplicity and low hardware cost and to the fact that it is moderately scalable — a fully interconnected processor topology, for example, would not only be much more expensive than a shared bus topology, but would also suffer from its limited scalability. Bus bandwidth limits scalability in shared bus multiprocessors, but for medium throughput applications (digital audio, music, etc.) and the size of the machine we are considering, a shared bus is ideal. We propose to solve the scalability problem by using multiple busses and hierarchy of busses, for which the ideas behind the OMA architecture directly apply. We refer to Lee and Bier [Lee90] for how the OMA concept is extended to such hierarchical bus structures.

From Fig. 1.3 we recall that the self-timed scheduling strategy falls naturally into a message passing paradigm that is implemented by the send and receive primitives inserted in the HSDFG. Accordingly, the shared memory in an architecture implementing such a scheduling strategy is used solely for message passing: the send primitive corresponds to writes to shared memory locations, and receive primitives correspond to reads from shared memory. Thus the shared memory is not used for storing shared data structures or for storing shared program code. In a self-timed strategy we can further ensure, at compile time, that each shared memory location is written by only one processor (one way of doing this is to simply assign distinct shared buffers to each of the send primitives, which is the scheme implemented in the Ptolemy environment); as a result, no atomic test-and-set instruction needs to be provided by the hardware.

Let us now consider the implementation of IPC in self-timed schedules on such a shared bus multiprocessor. The sender has to write into shared memory, which involves arbitration costs — it has to request access to the shared bus, and the access must be arbitrated by a bus arbiter. Once the sender obtains access to shared memory, it needs to perform a synchronization check on the shared memory location to ensure that the receiver has read data that was written in the previous iteration, to avoid overwriting previously written data. Such synchronization is typically implemented using a semaphore mechanism; the sender waits until a semaphore is reset before writing to a shared memory location, and upon writing that shared memory location, it sets that semaphore (the semaphore could be a bit in shared memory, one bit for each send operation in the parallel schedule). The receiver on the other hand busy waits until the semaphore is set before reading the shared memory location, and resets the semaphore after completing the read operation. It can easily be verified that this simple protocol guarantees correct sender-receiver synchronization, and, even though the semaphore bits have multiple writers, no atomic test-and-set operation is required of the hardware.

In summary the operations of the sender are: request bus, wait for arbitration, busy wait until semaphore is in the correct state, write the shared memory location if semaphore is in the correct state, and then release the bus. The corresponding operations for the receiver are: request bus, wait for arbitration, busy wait on semaphore, read the shared memory location if semaphore is in the correct state, and release the bus. The IPC costs are therefore due to bus arbitration time and due to semaphore checks. Such overhead consumes of the order of tens of instruction cycles if no special hardware support is employed for IPC. In addition, semaphore checks consume shared bus bandwidth.

An example of this is a four processor DSP56000 based shared bus system designed by Dolby labs for digital audio processing applications. In this machine, processors communicate through shared memory, and a central bus arbiter resolves bus request conflicts between processors. When a processor gets the bus it per-

forms a semaphore check, and continues with the shared memory transaction if the semaphore is in the correct state. It explicitly releases the bus after completing the shared memory transaction. A receive and a send together consume 30 instruction cycles, even if the semaphores are in their correct state and the processor gets the bus immediately upon request. This translates to 8% of the 380 instructions per processor in the example of Chapter 1, section 1.4, that considered processing samples of a high-quality audio signal at a sampling rate of 44 KHz on processors running on a 60ns clock. Such a high cost of communication forces the scheduler to insert as few interprocessor communication nodes as possible, which in turn limits the amount of parallelism that can be extracted from the algorithm.

One solution to this problem is to send more than one data sample when a processor gets access to the bus; the arbitration and synchronization costs are then amortized over several data samples. A scheme to “vectorize” data in this manner has been proposed by [Zivo94], where the authors use retiming [Lei91] to move delays in the HSDFG such that data can be moved in blocks, instead of one sample at a time. There are several problems with this strategy. First, retiming HSDFGs has to be done very carefully: moving delays across actors can change the initial state of the HSDFG causing undesirable transients in the algorithm implementation. This can potentially be solved by including preamble code to compute the value of the sample corresponding to the delay when that delay is moved across actors. This, however results in increased code size, and other associated code generation complications. Second, the work of Zivojinovic *et. al.* does not apply uniformly to all HSDFGs: if there are tight cycles in the graph that need to be partitioned among processors, the samples simply cannot be “vectorized” [Messer88]. Thus presence of a tight cycle precludes arbitrary blocking of data. Third, vectorizing samples leads to increased latency in the implementation; some signal processing tasks such as interactive speech are sensitive to delay, and hence the delay introduced due to blocking of data may be unacceptable. Finally, the problem of vectorizing data in HSDFGs into blocks, even with all the above limi-



tations, appear to be fundamentally hard; the algorithms proposed by Zivojinovic *et. al.* have exponential worst case run times. Code generated currently by the Ptolemy system does not support blocking (or vectorizing) of data for many of the above reasons.

Another possible solution is to use special hardware. One could provide a full interconnection network, thus obviating the need to go through shared memory. Semaphores could be implemented in hardware. One could use multiported memories. Needless to say, this solution is not favourable because of cost, especially when targeting embedded applications.

A general-purpose shared bus machine, the Sequent Balance [Patt90] for example, will typically use caches between the processor and the shared bus. Caches lead to increased shared memory bandwidth due to the averaging effect provided by block fetches and due to probabilistic memory access speedup due to cache hits. In signal processing and other real time applications, however, there is a stringent requirement for deterministic performance guarantee as opposed to probabilistic speedup. In fact, the unpredictability in task execution times introduced due to the use of caches may be a disadvantage for static scheduling techniques that utilize compile time estimates of task execution times to make scheduling decisions (we recall the discussion in section 1.2.3 on techniques for estimating task execution times). In addition, due to the deterministic nature of most signal processing problems (and also many scientific computation problems), shared data can be deterministically prefetched because information about when particular blocks of data are required by a particular processor can often be predicted by a compiler. This feature has been studied in [Mouss92], where the authors propose memory allocation schemes that exploit predictability in the memory access pattern in DSP algorithms; such a “smart allocation” scheme alleviates some of the memory bandwidth problems associated with high throughput applications.

Processors with caches can cache semaphores locally, so that busy waiting can be done local to the processor without having to access the shared bus, hence saving the bus bandwidth normally expended on semaphore checks. Such a procedure, however, requires special hardware (a snooping cache controller, for example) to maintain cache coherence; cost of such hardware usually makes it prohibitive in embedded scenarios.

Thus, for the embedded signal processing applications that we are focusing on, we argue that caches do not have a significant role to play, and we claim that the OT approach discussed previously provides a cost effective solution for minimizing IPC overhead in implementing self-timed schedules.

### **3.2.1 Using the OT approach**

The OT strategy, we recall, operates on the principle of determining (at compile time) the order in which processor communications occur, and enforcing that order at run time. For a shared bus implementation, this translates into determining the sequence of shared memory (or, equivalently, shared bus) accesses at compile time and enforcing this predetermined order at run time. This strategy, therefore, involves no run time arbitration; processors are simply granted the bus according to the pre-determined access order. When a processor obtains access to 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 task of maintaining ordered access to shared memory is done by a central *ordered transaction controller*. When the processors are downloaded with code, the controller too is loaded with the pre-determined access order list. At run time the controller simply grants bus access to processors according to this list, granting access to the next processor in the list when the current bus owner releases the bus. Such a mechanism is robust with respect to variations in execution times of the actors; the functionality of the system is unaffected by poor esti-

mates of these execution times, although the real-time performance obviously suffers.

If we are able to perform accurate compile time analysis, then each processor would obtain access to the shared bus whenever it needed it. No arbitration needs to be done since there is no contention for the bus. In addition, no semaphore synchronization needs to be performed, because the transaction ordering constraints respect data precedences in the algorithm; when a processor accesses a shared memory location and is correspondingly allowed access to it, the data accessed by that processor is certain to be valid. As a result, in the ideal scenario, a shared bus access takes no more than a single read or write cycle on the processor, and the overall cost of communicating one data sample is two or three instruction cycles.

The performance of this scheme depends on how accurately the execution times of the actors are known at compile time. If these compile time estimates are reasonably accurate, then an access order can be obtained such that a processor gains access to shared memory whenever it needs. Otherwise, a processor may have to idle until it gets a bus grant, or, even worse, a processor when granted the bus may not complete its transaction immediately, thus blocking all other processors from accessing the bus. This problem would not arise in normal arbitration schemes, because independent shared memory accesses would be dynamically reordered.

We will quantify these performance issues in the next chapter, where we show that when reasonably good estimates of actor execution times are available, forcing a run time access order does not in fact sacrifice performance significantly.

### **3.3 Design of an Ordered Memory Access multiproces-**

**sor**

### **3.3.1 High level design description**

We chose Motorola DSP96002 processors for the OMA prototype. Although the OMA architecture can be built around any programmable DSP that has built-in bus arbitration logic, the DSP96002 is particularly suited to our design because of its dual bus architecture and bus arbitration mechanism. In addition these processors are powerful DSPs with floating point capability [Moto89].

A high level block diagram of such a system is depicted in Fig. 3.2. Each DSP96002 is provided with a private memory that contains its program code; this local memory resides on one of the processor busses (the "A" bus). The alternate "B" bus of all processors are connected to the shared bus, and shared memory resides on the shared bus. The transaction controller grants access to processors using the bus grant (BG) lines on the processor. A processor attempts to perform a shared memory access when it executes a communication actor (either send or receive). If its BG line is asserted it performs the access, otherwise it stalls and waits for the assertion.

After a processor obtains access to the shared bus, it performs the shared memory operation and releases the bus. The transaction controller detects the release of the bus and steps through its ordered list, granting the bus to the next processor in its list.

The cost of transfer of one word of data between processors is 3 instruction cycles in the ideal case; two of these correspond to a shared memory write (by the sender) and a shared memory read (by the receiver), and an extra instruction cycle is expended in bus release by the sender and bus acquisition by the receiver.

Thus for the example of Chapter 1, less than 1% of the available 380 instructions per sample are required per transaction. This is of course in the ideal scenario where the sender and the receiver obtain access to the shared bus upon request. Such low overhead interprocessor communication is obtained with the

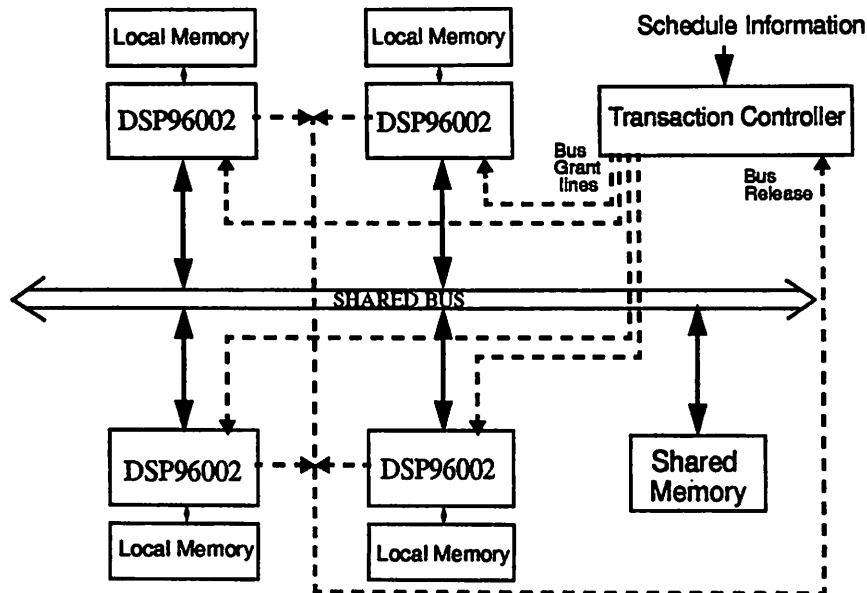


Figure 3.2. Block diagram of the OMA prototype

transaction controller providing the only additional hardware support. As described in a subsequent section, this controller can be implemented with very simple hardware.

### 3.3.2 A modified design

In the design proposed above, processor to processor communication occurs through a central shared memory; two transactions — one write and one read — must occur over the shared bus. This situation can be improved by distributing the shared memory among processors, as shown in Fig. 3.3, where each processor is assigned shared memory in the form of hardware FIFO buffers. Writes to each FIFO are accomplished through the shared bus; the sender simply writes to the FIFO of the processor to which it wants to send data by using the appropriate shared memory address.

Use of a FIFO implies that the receiver must know the exact order in which data is written into its input queue. This, however, is guaranteed by the ordered

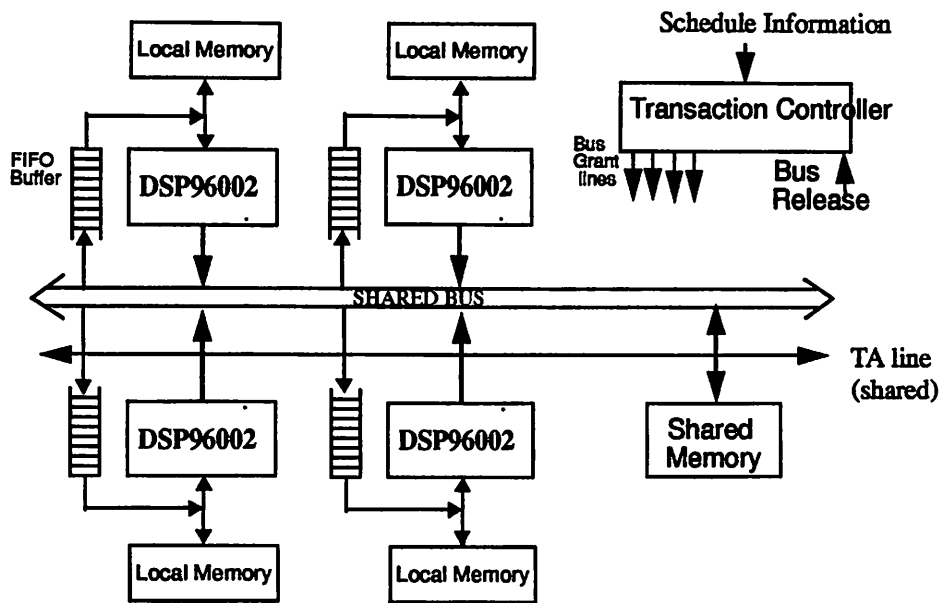


Figure 3.3. Modified design

transaction strategy. Thus replacing a RAM (random access memory) based shared memory with distributed FIFOs does not alter the functionality of the design. The sender need only block when the receiving queue is full, which can be accomplished in hardware by using the 'Transfer Acknowledge (TA)' signal on the DSP96002; a device can insert arbitrary number of wait states in the processor memory cycle by de-asserting the TA line. Whenever a particular FIFO is accessed, its 'Buffer Full' line is enabled onto the TA line of the processors (Fig. 3.4). Thus a full FIFO automatically blocks the processor trying to write into it, and no polling needs to be done by the sender. Receiver read is local to a processor, and does not consume shared bus bandwidth. The receiver can be made to either poll the FIFO empty line to check for an empty queue, or we one can use the same TA signal mechanism to block processor reads from an empty queue. The TA

mechanism will then use the local ("A") bus control signals ("A" bus TA signal, "A" address bus etc.). This is illustrated in Fig. 3.4.

Use of such a distributed shared memory mechanism has several advantages. First, the shared bus traffic is effectively halved, because only writes need to go through the shared bus. Second, in the design of Fig. 3.2, a processor that is granted the bus is delayed in completing its shared memory access, all other processors waiting for the bus get stalled; this does not happen for half the transactions in the modified design of Fig. 3.3 because receiver reads are local. Thus there is more tolerance to variations in the time at which a receiver reads data sent to it. Last, a processor can broadcast data to all (or any subset) of processors in the system by simultaneously writing to more than one FIFO buffer. Such broadcast is not possible with a central shared memory.

The modified design, however, involves a significantly higher hardware cost than the design proposed earlier. As a result, the OMA prototype was built around the central shared memory design and not the FIFO based design. In addition, the DSP96002 processor has an on-chip host interface unit that can be used as

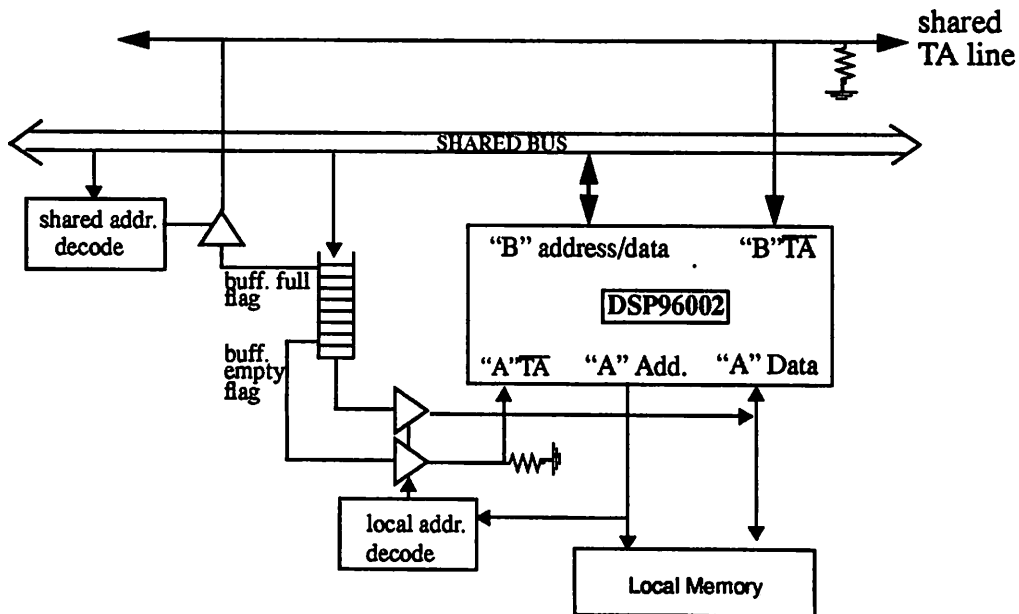


Figure 3.4. Details of the "TA" line mechanism (only one processor is shown).

a 2-deep FIFO; therefore, the potential advantage of using distributed FIFOs can still be evaluated to some degree by using the chip host interface even in the absence of external FIFO hardware.

Simulation models were written for both the above designs using the Thor hardware simulator [Thor86] under the Frigg multi-processor simulator system [Bier89]. Frigg allows the Thor simulator to communicate with a timing-driven functional simulator for the DSP96002 processor provided by Motorola Inc. The Motorola simulator also simulates Input/Output (I/O) operations of the pins of the processor, and Frigg interfaces the signals on the pins to with the rest of the Thor simulation; as a result, hardware associated with each processor (memories, address decoding logic, etc.) and interaction between processors can be simulated using Frigg. This allows functionality of the entire system to be verified by running actual programs on the processor simulators. We did not use this model for performance evaluation, however, because with just a four processor system the cycle-by-cycle Frigg simulation was far too slow, even for very simple programs. A higher-level (behavioral) simulation would be more useful than a cycle-by-cycle simulation for the purposes of performance evaluation, although we did not attempt such high-level simulation of our designs.

The remainder of this chapter describes hardware and software design details of an OMA board prototype.

### **3.4 Design details of a prototype**

A proof-of-concept prototype of the OMA architecture has been designed and implemented. The single printed circuit board design is comprised of four DSP96002 processors; the transaction controller is implemented on a Xilinx FPGA (Field Programmable Gate Array). The Xilinx chip also handles the host interface functions, and implements a simple I/O mechanism. A hierarchical description of the hardware design follows.



### 3.4.1 Top level design

This section refers to Fig. 3.5. At the top level, there are four “processing element” blocks that consist of the processor, local memory, local address decoder, and some glue logic. Address, data, and control busses from the PE blocks are connected to form the shared bus. Shared memory is connected to this bus; address decoding is done by the “shared address decoder” PAL (programmable array logic) chip. A central clock generator provides a common clock signal to all processing elements.

A Xilinx FPGA (XC3090) implements the transaction controller and a simple I/O mechanism, and is also used to implement latches and buffers during bootup, thus saving glue logic. A fast static RAM (up to 32K x 8) stores the bus access order in the form of processor identifications (IDs). The sequence of processor IDs is stored in this “schedule RAM”, and this determines the bus access order. An external latch is used to store the processor ID read from the schedule RAM. This ID is then decoded to obtain the processor bus grants.

A subset of the 32 address lines connect to the Xilinx chip, for addressing the I/O registers and other internal registers. All 32 lines from the shared data bus are connected to the Xilinx. The shared data bus can be accessed from the external connector (the “right side” connector in Fig. 3.5) only through the Xilinx chip. This feature can be made use of when connecting multiple OMA boards: shared busses from different boards can be made into one contiguous bus, or they can be left disconnected, with communication between busses occurring via asynchronous “bridges” implemented on the Xilinx FPGAs. We discuss this further in section 3.4.7.

Connectors on both ends of the board bring out the shared bus in its entirety. Both left and right side connectors follow the same format, so that multiple boards can be easily connected together. Shared control and address busses are

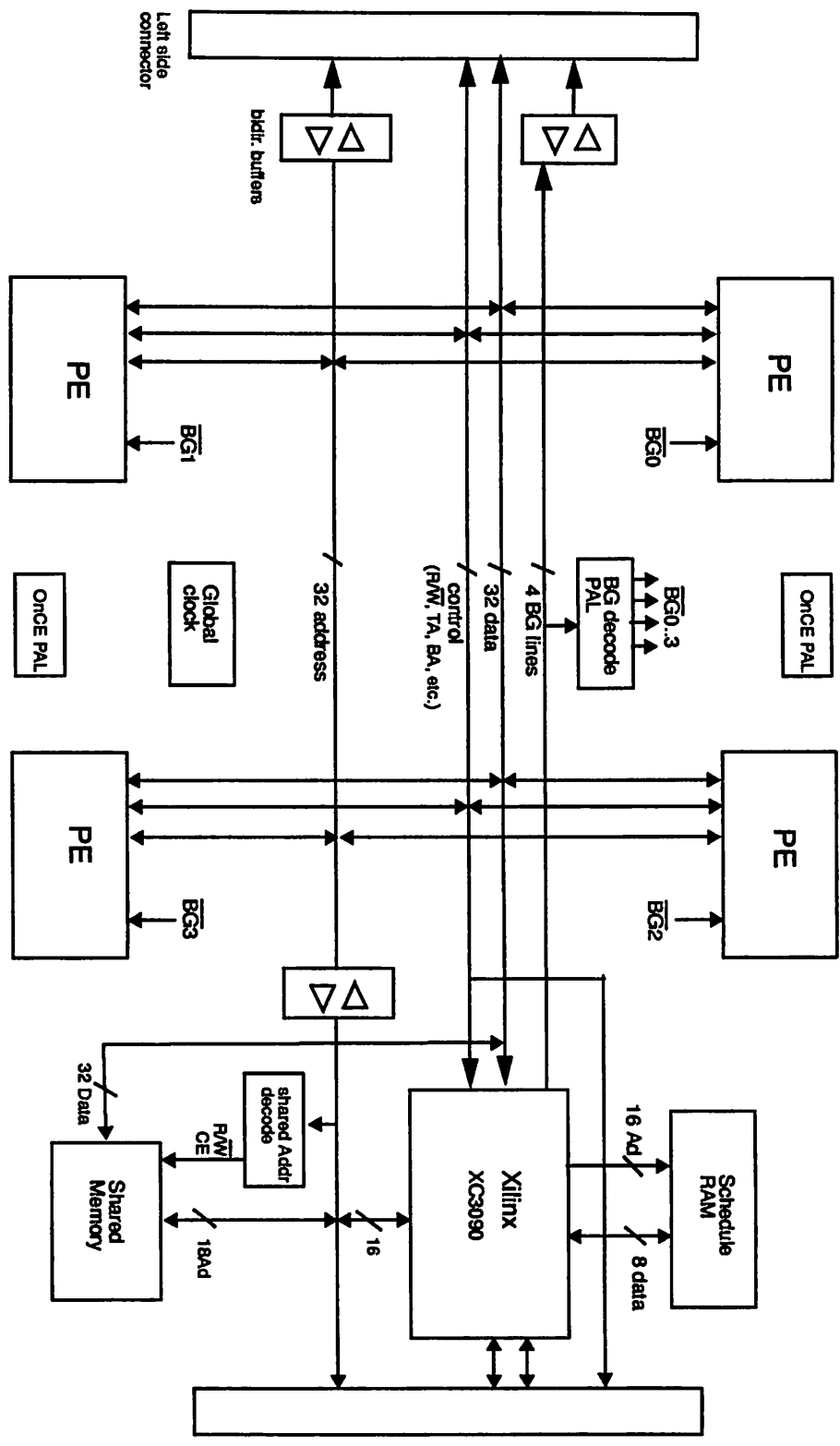


Figure 3.5. Top-level schematic of the OMA prototype

buffered before they go off board via the connectors, and the shared data bus is buffered within the Xilinx.

The DSP96000 processors have on chip emulation (“OnCE” in Motorola terminology) circuitry for debugging purposes, whereby a serial interface to the OnCE port of a processor can be used for in-circuit debugging. On the OMA board, the OnCE ports of the four processors are multiplexed and brought out as a single serial port; a host may select any one of the four OnCE ports and communicate to it through a serial interface.

We discuss the design details of the individual components of the prototype system next.

### **3.4.2 Transaction order controller**

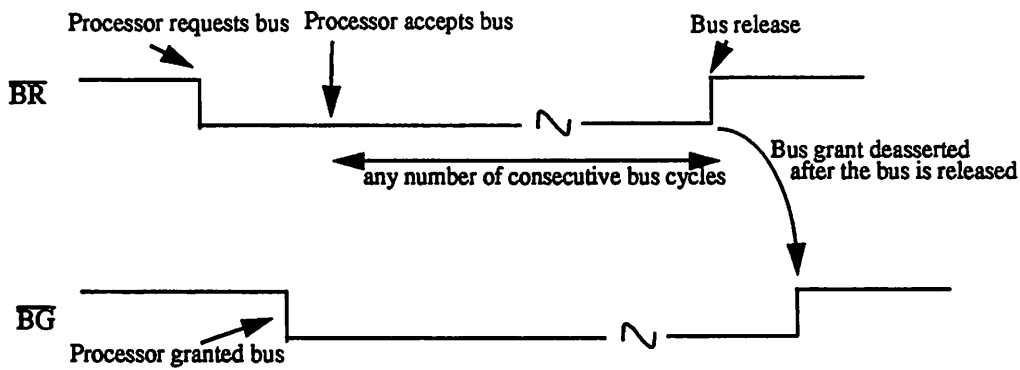
The task of the transaction order controller is to enforce the predetermined bus access order at run time. A given transaction order determines the sequence of processor bus accesses that must be enforced at run time. We refer to this sequence of bus accesses by the term **bus access order list**. Since the bus access order list is program dependent, the controller must possess memory into which this list is downloaded after the scheduling and code generation steps are completed, and when the transaction order that needs to be enforced is determined. The controller must step through the access order list, and must loop back to the first processor ID in the list when it reaches the end. In addition the controller must be designed to effectively use bus arbitration logic present on-chip, to conserve hardware.

#### **3.4.2.1. Processor bus arbitration signals**

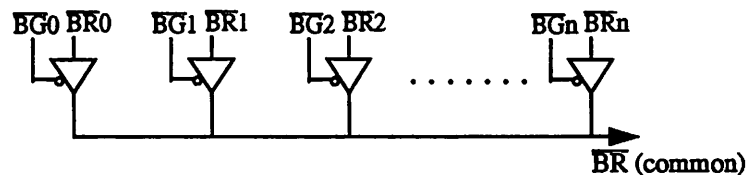
We use the bus grant ( $\overline{BG}$ ) signal on the DSP chip to allow the processor to perform a shared bus access, and we use the bus request ( $\overline{BR}$ ) signal to tell the controller when a processor completes its shared bus access.

Each of the two ports on the DSP96002 has its own set of arbitration signals; the  $\overline{BG}$  and  $\overline{BR}$  signals are most relevant for our purposes, and these signals

are relevant only for the processor port connected to the shared bus. As the name suggests, the  $\overline{BG}$  line (which is an input to the processor) must be asserted before a processor can begin a bus cycle: the processor is forced to wait for  $\overline{BG}$  to be asserted before it can proceed with the instruction that requires access to the bus. Whenever an external bus cycle needs to be performed, a processor asserts its  $\overline{BR}$  signal, and this signal remains asserted until an instruction that does not access the shared bus is executed. We can therefore use the  $\overline{BR}$  signal to determine when a shared bus owner has completed its usage of the shared bus (Fig. 3.6 (a)).



(a)



(b)

Figure 3.6. Using processor bus arbitration signals for controlling bus access

The rising edge of the  $\overline{BR}$  line is used to detect when a processor releases the bus. To reduce the number of signals going from the processors to the control-

ler, we multiplexed the  $\overline{\text{BR}}$  signals from all processors onto a common  $\overline{\text{BR}}$  signal. The current bus owner has its  $\overline{\text{BR}}$  output enabled onto this common reverse signal; this provides sufficient information to the controller because the controller only needs to observe the  $\overline{\text{BR}}$  line from the current bus owner. This arrangement is shown in Fig. 3.6 (b); the controller grants access to a processor by asserting the corresponding  $\overline{\text{BG}}$  line, and then it waits for an upper edge on the reverse  $\overline{\text{BR}}$  line. On receiving a positive going edge on this line it grants the bus to the next processor in its list.

### 3.4.2.2. A simple implementation

One straightforward implementation of the above functionality is to use a counter addressing a RAM that stores the access order list in the form of processor IDs. We call this counter the *schedule counter* and the memory that stores the processor IDs is called the *schedule RAM*. Decoding the output of the RAM provides the required  $\overline{\text{BG}}$  lines. The counter is incremented at the beginning of a processor transaction by the negative going edge of the common  $\overline{\text{BR}}$  signal and the output of the RAM is latched at the positive going edge of  $\overline{\text{BR}}$ , thus granting the bus to the next processor as soon as the current processor completes its shared memory transaction. The counter is reset to zero after it reaches the end of the list (i.e. the counter counts modulo the bus access list size). This is shown in Fig. 3.7. Incrementing the counter as soon as  $\overline{\text{BR}}$  goes low ensures enough time for the counter outputs and the RAM outputs to stabilize. For a 33MHz processor with zero wait states,  $\overline{\text{BR}}$  width is a minimum of 60 nanoseconds. Thus the counter incrementing and the RAM access must both finish before this time. Consequently, we need a fast counter and fast static RAM for the schedule memory. The width of the counter determines the maximum allowable size of the access list (a counter width of size  $n$  implies a maximum list size of  $2^n$ ); a wider counter, however, implies a slower counter. If, for a certain width, the counter (implemented on the Xilinx part in our case) turns out to be too slow — i.e. the output of the schedule memory will not stabilize at least one latch set up period before the positive going edge of  $\overline{\text{BR}}$

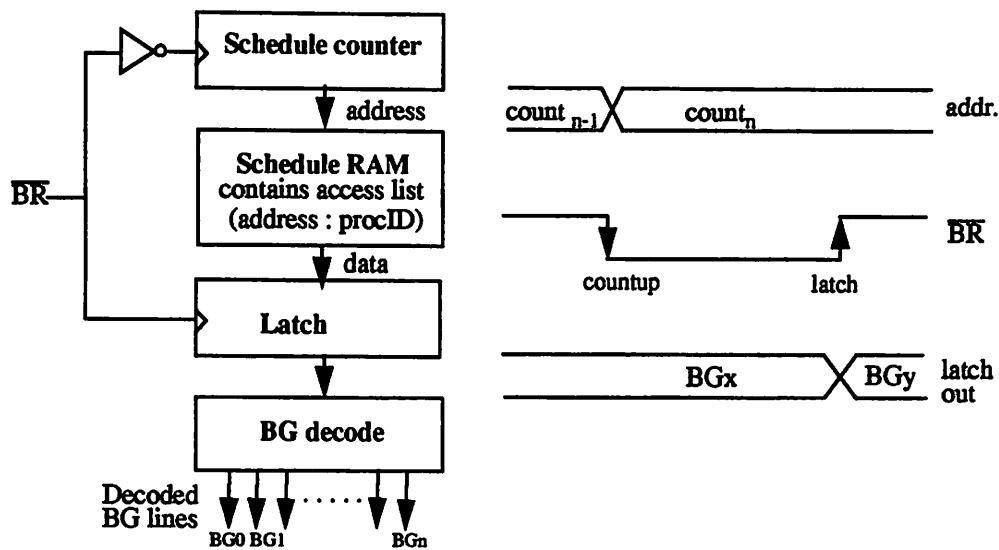


Figure 3.7. Ordered Transaction Controller implementation

arrives — wait states may have to be inserted in the processor bus cycle to delay the positive edge of  $\overline{BR}$ . We found that a 10 bit wide counter does not require any wait states, and allows a maximum of 1024 processor IDs in the access order list.

### 3.4.2.3. Presetable counter

A single bus access list implies we can only enforce one bus access pattern at run time. In order to allow for some run time flexibility we have implemented the OMA controller using a presetable counter. The processor that currently owns the bus can preset this counter by writing to a certain shared memory location. This causes the controller to jump to another location in the schedule memory, allowing the multiple bus access schedules to be maintained in the schedule RAM and switching between them at run time depending on the outcome of computations in the program. The counter appears as an address in the shared memory map of the processors. The presetable counter mechanism is shown in Fig. 3.8.

An arbitrary number of lists may, in principle, be maintained in the schedule memory. This feature can be used to support algorithms that display data

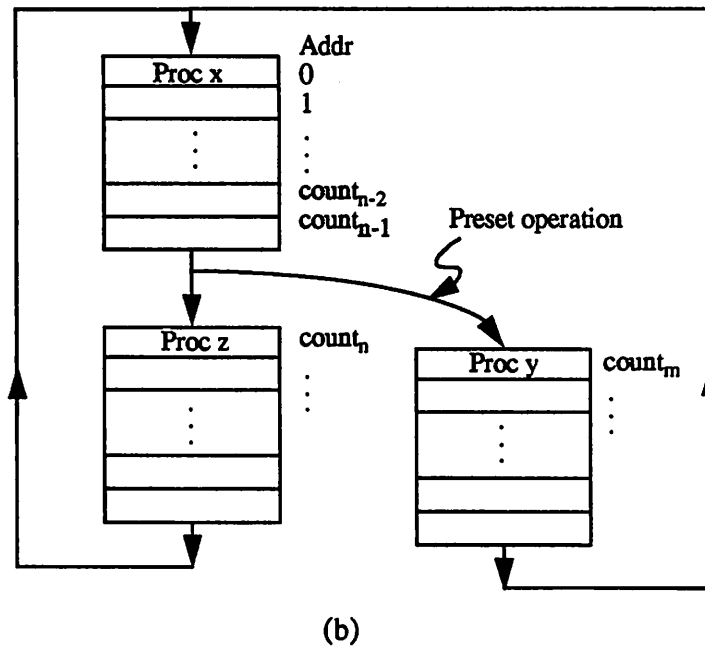
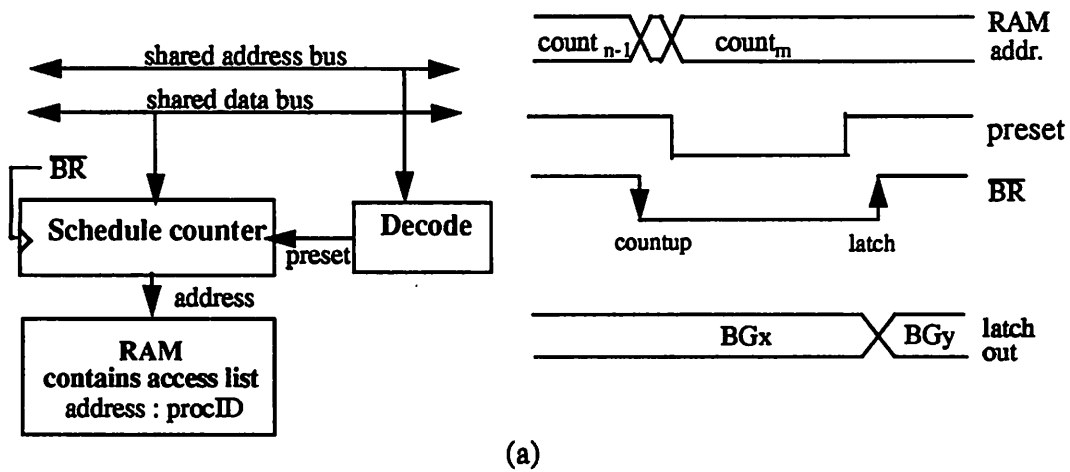


Figure 3.8. Presetable counter implementation

dependency in their execution. For example, a dataflow graph with a conditional construct will, in general, require a different access schedule for each outcome of the conditional. Two different SDF graphs are executed in this case, depending on the branch outcome, and the processor that determines the branch outcome can also be assigned the task of presetting the counter, making it branch to the access

list of the appropriate SDF subgraph. The access controller behaves as in Fig 3.8 (b).

We discuss the use of this presettable feature in detail in Chapter 6.

### **3.4.3 Host interface**

The function of the host interface is to allow downloading programs onto the OMA board, controlling the board, setting parameters of the application being run, and debugging from a host workstation. The host for the OMA board connects to the shared bus through the Xilinx chip, via one of the shared bus connectors. Since part of the host interface is configured inside the Xilinx, different hosts (32 bit, 16 bit) with different handshake mechanisms can be used with the board.

The host that is being used for the prototype is a DSP56000 based DSP board called the S-56X card, manufactured by Ariel Corp [Ariel91]. The S-56X card is designed to fit into one of the Sbus slots in a Sun Sparc workstation; a user level process can communicate with the S-56X card via a unix device driver. Thus the OMA board too can be controlled (via the S-56X card) by a user process running on the workstation. The host interface configuration is depicted in Fig. 3.9.

Unlike the DSP56000 processors, the DSP96002 processors do not have built in serial ports, so the S-56X board is also used as a serial I/O processor for the OMA board. It essentially performs serial to parallel conversion of data, buffering of data, and interrupt management. The Xilinx on the OMA board implements the necessary transmit & receive registers, and synchronization flags — we discuss the details of the Xilinx circuitry shortly.

The S-56X card communicates with the Sparc Sbus using DMA (direct memory access). A part of the DSP56000 bus and control signals are brought out of the S-56X card through another Xilinx FPGA (XC3040) on the S-56X. For the purpose of interfacing the S-56X board with the OMA board, the Xilinx on the S-56X card is configured to bring out 16 bits of data and 5 bits of address from the



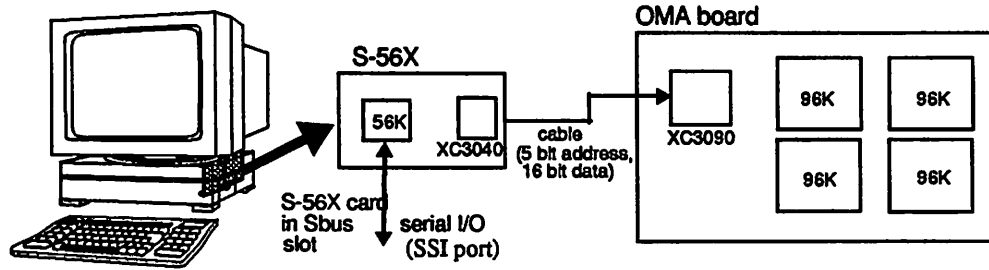


Figure 3.9. Host interface

DSP56000 processor onto the cable connected to the OMA (see Fig. 3.9). In addition, the serial I/O port (the SSI port) is also brought out, for interface with I/O devices such as A/D and D/A converters. By making the DSP56000 write to appropriate memory locations, the 5 bits of address and 16 bits of data going into the OMA may be set and strobed for a read or a write, to or from the OMA board. In other words, the OMA board occupies certain locations in the DSP56000 memory map; host communication is done by reading and writing to these memory locations.

### 3.4.4 Processing element

Each processing element (PE) consists of a DSP96002 processor, local memory, address buffers, local address decoder, and some address decoding logic. The circuitry of each processing element is very similar to the design of the Motorola 96000 ADS (Application Development System) board [Moto90]. The local address, control, and data busses are brought out into a 96 pin euro-connector, following the format of the 96ADS. This connector can be used for local memory expansion; we have used it for providing local I/O interface to the processing element (as an alternative to using the shared bus for I/O). Port A of the processor forms the local bus, connecting to local memory and address decoding PAL. Each PE also contains address buffers, and logic to set up the bootup mode upon reset and powerup. Port B of the processor is connected to the shared bus.

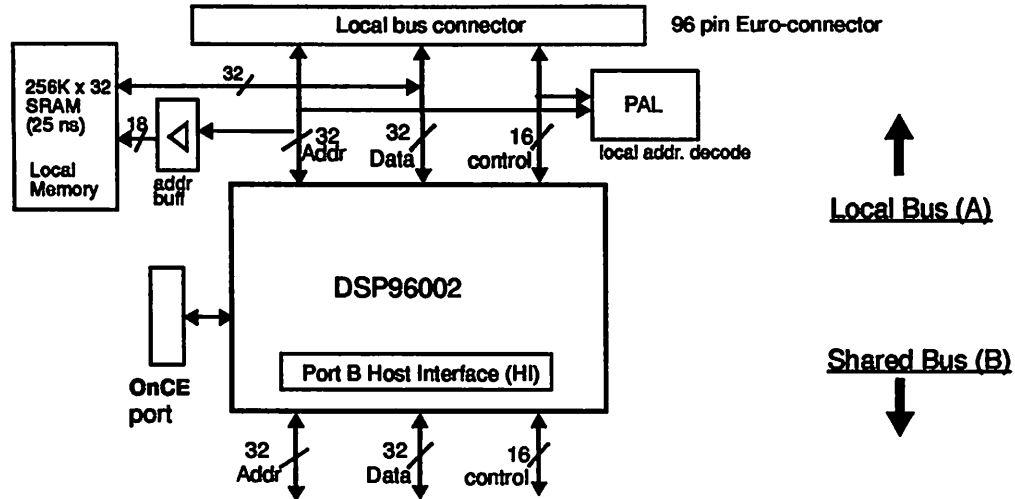


Figure 3.10. Processing element

The DSP96002 processor has a Host Interface (HI) on each of its ports. The port B HI is memory mapped to the shared bus, so that HI registers may be read and written from the shared bus. This feature allows a host to download code and control information into each processor through the shared bus. Furthermore a processor, when granted the shared bus, may also access the port B HI of other processors. This allows processors to bypass the shared memory while communicating with one another and to broadcast data to all processors. In effect, the HI on each processor can be used as a two-deep local FIFO, similar to the scheme in section 3.3.2, except that the FIFO is internal to each processor.

### 3.4.5 Xilinx circuitry

As mentioned previously, the XC3090 Xilinx chip is used to implement the transaction controller as well as a simple I/O interface. It is also configured to provide latches and buffers for addressing the Host Interface (HI) ports on the DSP96002 during bootup and downloading of code onto the processors. For this to work, the Xilinx is first configured to implement the bootup and download related circuitry, which consists of latches to drive the shared address bus and to access the

schedule memory. After downloading code onto the processors, and downloading the bus access order into the schedule RAM, the Xilinx chip is reconfigured to implement the ordered transaction controller and the I/O interface. Thus the process of downloading and running a program requires configuring the Xilinx chip twice.

There are several possible ways in which a Xilinx part may be programmed. For the OMA board, the configuration bitmap is downloaded byte-wise by the host (Sun workstation through the S-56X card). The bitmap file, generated and stored as a binary file on a workstation, is read in by a function implemented in the *qdm* software (discussed in section 3.5, which describes the OMA software interface) and the bytes thus read are written into the appropriate memory location on the S-56X card. The DSP56K processor on the S-56X then strobes these bytes into the Xilinx configuration port on the OMA board. The user can reset and reconfigure the Xilinx chip from a Sun Sparc workstation by manipulating the Xilinx control pins by writing to a "Xilinx configuration latch" on the OMA board. Various configuration pins of the Xilinx chip are manipulated by writing different values into this latch.

We use two different Xilinx circuits (*bootup.bit* and *momal.bit*), one during bootup and the other during run time. The Xilinx configuration during bootup helps eliminate some glue logic that would otherwise be required to latch and decode address and data from the S-56X host. This configuration allows the host to read and write from any of the HI ports of the processors, and also to access the schedule memory and the shared memory on board.

Run time configuration on the Xilinx consists of the transaction controller implemented as a presettable counter. The counter can be preset through the shared bus. It addresses an external fast RAM (8 nanosecond access time) that contains processor IDs corresponding to the bus access schedule. Output from the schedule memory is externally latched and decoded to yield bus grant ( $\overline{BG}$ ) lines (Fig. 3.7).

A schematic of the Xilinx configuration at run time is given in Fig. 3.11. This configuration is for I/O with an S-56X (16 bit data) host, although it can easily be modified to work with a 32 bit host.

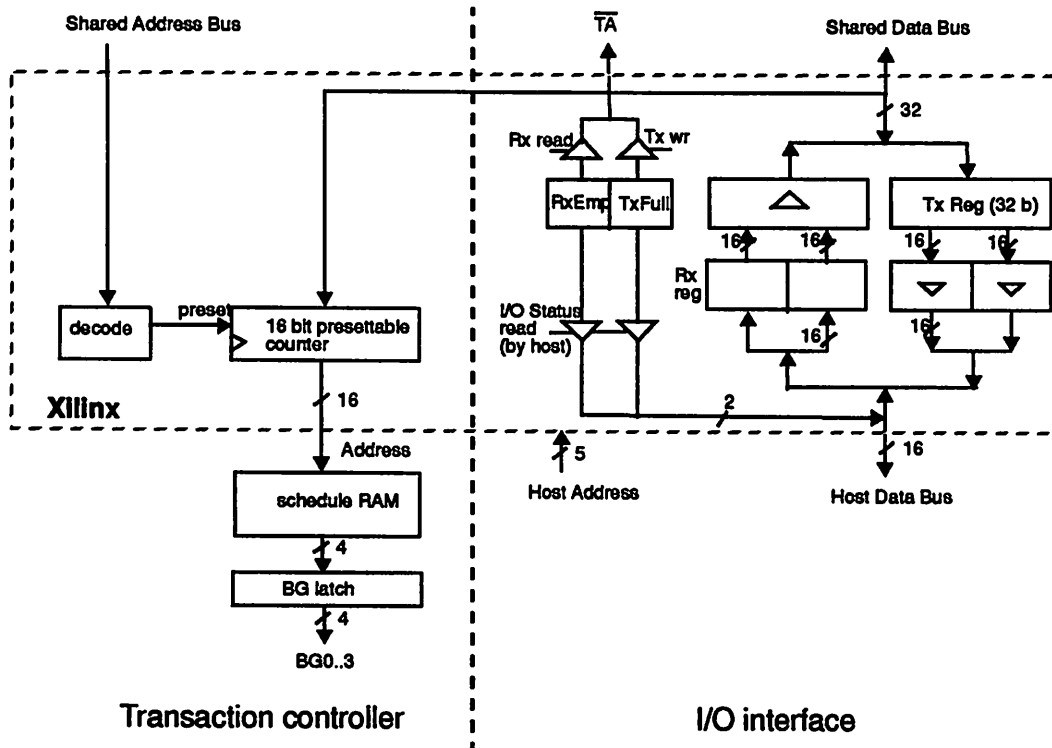


Figure 3.11. Xilinx configuration at run time

### 3.4.5.1. I/O interface

The S-56X board reads data from the Transmit (Tx) register and writes into the receive (Rx) register on the Xilinx. These registers are memory-mapped to the shared bus, such that any processor that possesses the bus may write to the Tx register or read from the Rx register. For a 16 bit host, two transactions are required to perform a read or write with the 32 bit Tx and Rx registers. The processors themselves need only one bus access to load or unload data from the I/O interface. Synchronization on the S-56X (host) side is done by polling status bits that indicate an Rx empty flag (if true, the host performs a write, otherwise it busy waits) and a Tx full flag (if true, the host performs a read, otherwise it busy waits). On the OMA side, synchronization is done by the use of the TA (transfer acknowledge) pin on

the processors. When a processor attempts to read Rx or write Tx, the appropriate status flags are enabled onto the TA line, and wait states are automatically inserted in the processor bus cycle whenever the TA line is not asserted, which in our implementation translates to wait states whenever the status flags are false. Thus processors do not have the overhead of polling the I/O status flags; an I/O transaction is identical to a normal bus access, with zero or more wait states inserted automatically.

The DSP56000 processor on the S-56X card is responsible for performing I/O with the actual (possibly asynchronous) data source and acts as the interrupt processor for the OMA board, relieving the board of tasks such as interrupt servicing and data buffering. This of course has the downside that the S-56X host needs to be dedicated to “spoon-feeding” the OMA processor board, and limits other tasks that could potentially run on the host.

### **3.4.6 Shared memory**

Space for two shared memory modules are provided, so that up to 512K x 32 bits of shared static RAM can reside on board. The memory must have an access time of 25ns to achieve zero wait state operation.

### **3.4.7 Connecting multiple boards**

Several features have been included in the design to facilitate connecting together multiple OMA boards. The connectors on either end of the shared bus are compatible, so that boards may be connected together in a linear fashion (Fig. 3.12). As mentioned before, the shared data bus goes to the “right side connector” through the Xilinx chip. By configuring the Xilinx to “short” the external and internal shared data busses, processors on different boards can be made to share one contiguous bus. Alternatively, busses can be “cleaved” on the Xilinx chip, with communication between busses implemented on the Xilinx via an asynchro-

nous mechanism (e.g. read and write latches synchronized by “full” and “empty” flags).

This concept is similar to the idea used in the SMART processor array [Koh90], where the processing elements are connected to a switchable bus: when the bus switches are open processors are connected only to their neighbors (forming a linear processor array), and when the switches are closed processors are connected onto a contiguous bus. Thus the SMART array allows formation of clusters of processors that reside on a common bus; these clusters then communicate with adjacent clusters. When we connect multiple OMA boards together, we get a similar effect: in the “shorted” configuration processors on different boards connect to a single bus, whereas in the “cleaved” configuration processors on different boards reside on common busses, and neighboring boards communicate through an asynchronous interface.

Fig. 3.12 illustrates the above scheme. The highest 3 bits of the shared address bus are used as the “board ID” field. Memory, processor Host Interface ports, configuration latches etc. decode the board ID field to determine if a shared memory or host access is meant for them. Thus, a total of 8 boards can be hooked onto a common bus in this scheme.

## **3.5 Hardware and software implementation**

### **3.5.1 Board design**

We used single sided through-hole printed circuit board technology for the OMA prototype. The printed circuit board design was done using the ‘SIERA’ system developed at Professor Brodersen’s group at Berkeley [Sriv92]. Under this system, a design is entered hierarchically using a netlist language called SDL (Structure Description Language). Geometric placement of components can be easily specified in the SDL netlist itself. A ‘tiling’ feature is also provided to ease compact fitting of components. The SDL files were written in a modular fashion;

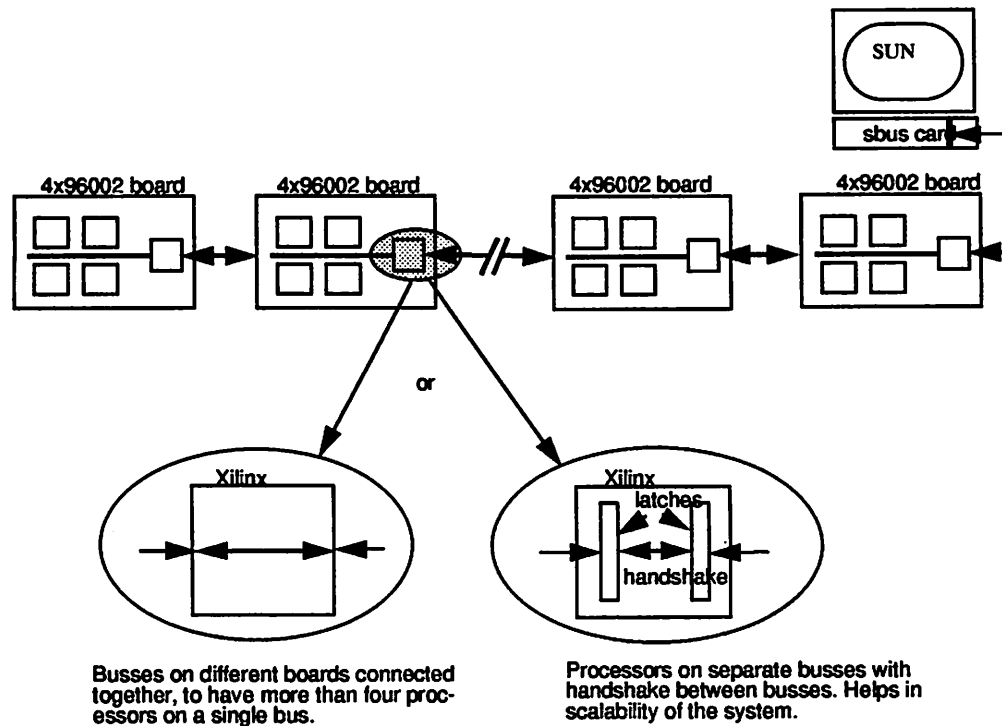


Figure 3.12. Connecting multiple boards

the schematics hierarchy is shown in Fig. 3.13. The SIERA design manager (DMoct) was then used to translate the netlists into an input file acceptable by Racal, a commercial PCB layout tool, which was then used to autoroute the board in ten layers, including one Vcc and one Ground plane. The files corresponding to the traces on each layer (gerber files) were generated using Racal, and these files were then sent to Mosis for board fabrication. Component procurement was mainly done using the FAST service, but some components were ordered/bought directly from electronic component distributors. Table 3.1 lists the salient physical features of the board, and Fig. 3.14 shows a photograph of the board.

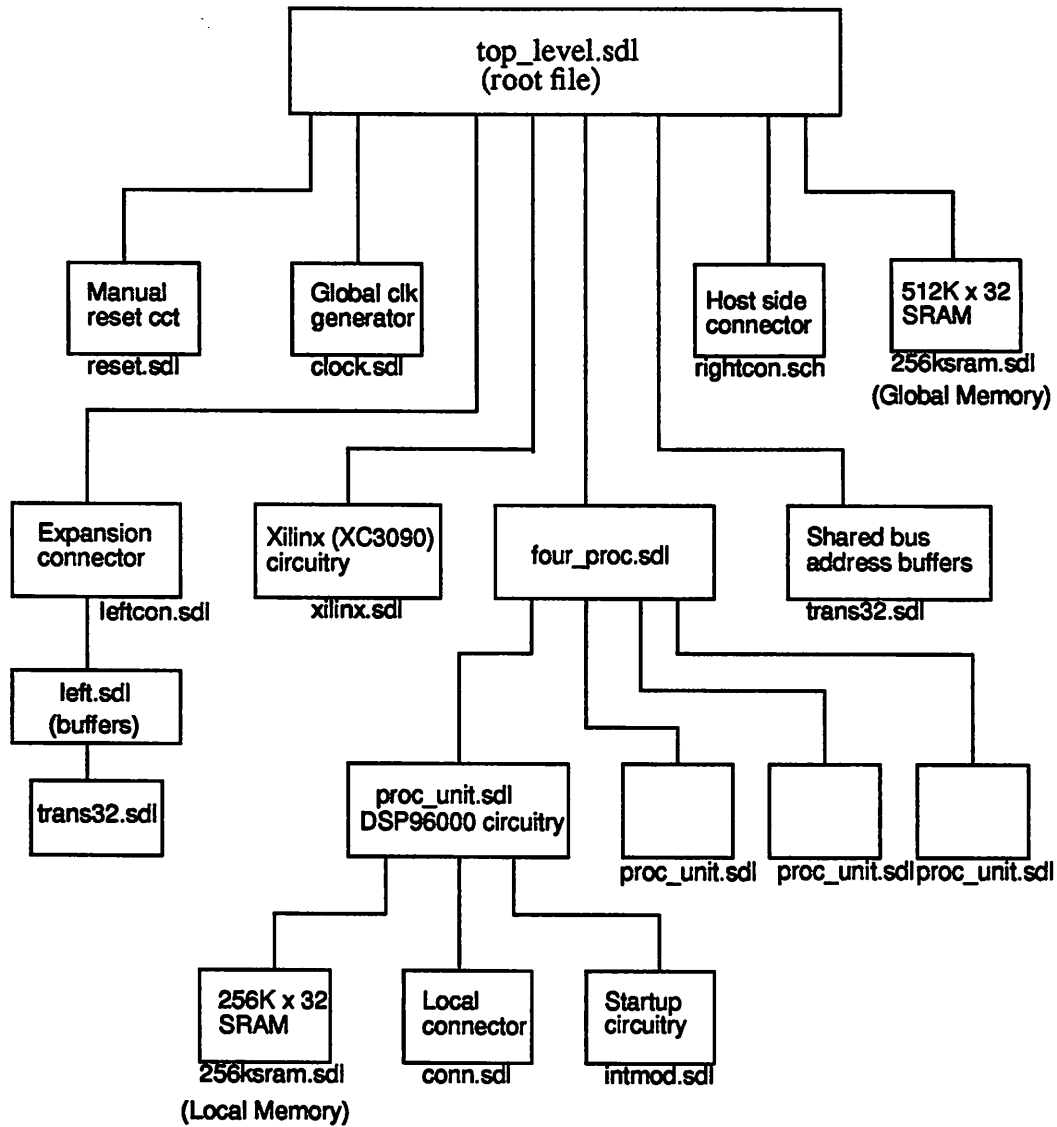


Figure 3.13. Schematics hierarchy of the four processor OMA architecture

Table 3.1. OMA board physical specs

Dimensions	30 cm. x 17 cm.
Layers	10 (including ground and Vcc plane)
Number of Components	230 parts + 170 bypass capacitors
<i>sdl</i> code	2800 lines
Memory	512K words shared, 256K words local



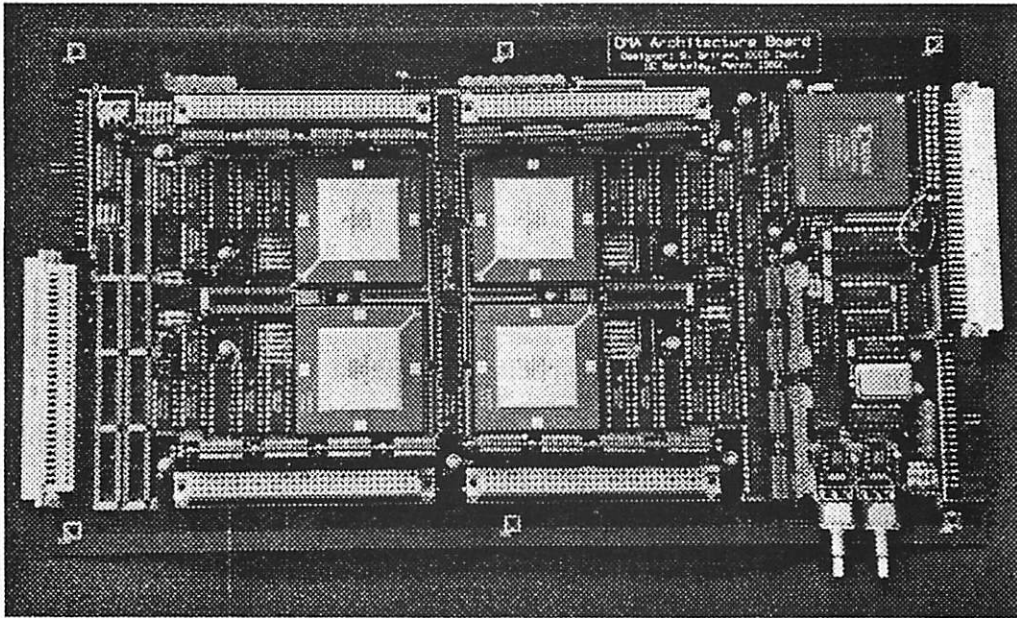


Figure 3.14. OMA prototype board photograph

### 3.5.2 Software interface

As discussed earlier, we use an S-56X card attached to a Sparc as a host for the OMA board. The Xilinx chip on the S-56X card is configured to provide 16 bits of data and 5 bits of address. We use the *qdm* [Laps91] software as an interface for the S-56X board; *qdm* is a debugger/monitor that has several useful built-in routines for controlling the S-56X board, for example data can be written and read from any location in the DSP56000 address space through function calls in *qdm*. Another useful feature of *qdm* is that it uses 'tcl', an embeddable, extensible, shell-like interpreted command language [Ous94]. Tcl provides a set of built-in functions (such as an expression evaluator, variables, control-flow statements etc.) that can be executed via user commands typed at its textual interface, or from a specified command file. Tcl can be extended with application-specific commands; in our case these commands correspond to the debugging/monitor commands implemented in *qdm* as well as commands specific to the OMA. Another useful feature of tcl is the scripting facility it provides; sequences of commands can be conve-

niently integrated into scripts, which are in turn executed by issuing a single command.

Some functions specific to the OMA hardware that have been compiled into *qdm* are the following:

```
omaxiload fileName.bit :    load OMA Xilinx with configuration
                               specified by file.bit

omapboot fileName.lod proc# :  load bootstrap monitor code into the
                               specified processor

omapload fileName.lod proc# :  load DSP96002 .lod file into the
                               specified processor

schedload accessOrder :    load OMA bus access schedule memory
```

These functions use existing *qdm* functions for reading and writing values to the DSP56000 memory locations that are mapped to the OMA board host interface. The sequence of commands needed to download code onto the OMA board and run it is summarized in Fig. 3.15.

```
proc omaDoAll {} {
    xload                                # configure S-56X board Xilinx
    omareset                             # reset OMA board
    omaxiload bootup.bit                 # configure OMA Xilinx for booting procs
    foreach i {0 1 2 3} {
        omapboot $i omaMon.lod          # load bootstrap monitor routine
        omapload $i $icode.lod         # load code (0code.lod, 1code.lod etc.)
        schedload access_order.data    # load bus access schedule into
    }                                     schedule memory
    omaxiload moma1.bit                 # reconfigure OMA Xilinx to implement
                                         Transaction Controller and I/O
    load host.lod; run                 # run and load I/O interrupt routines on S-
                                         56X board
    synch                               # start all processors synchronously
}
```

Figure 3.15. Steps required for downloading code (*tcl* script *omaDoAll*)

Each processor is programmed through its Host Interface via the shared bus. First, a monitor program (`omaMon.lod`) consisting of interrupt routines is loaded and run on the selected processor. Code is then loaded into processor memory by writing address and data values into the HI port and interrupting the processor. The interrupt routine on the processor is responsible for inserting data into the specified memory location. The S-56X host forces different interrupt routines, for specifying which of the three (X, Y, or P) memories the address refers to and for specifying a read or a write to or from that location. This scheme is similar to that employed in downloading code onto the S-56X card [Ariel91].

Status and control registers on the OMA board are memory mapped to the S-56X address space and can be accessed to reset, reboot, monitor, and debug the board. Tcl scripts were written to simplify commands that used are most often (e.g. `'change y:fff0 0x0'` was aliased to `'omareset'`). The entire code downloading procedure is executed by the tcl script `'omaDoAll'` (see Fig. 3.15).

A Ptolemy multiprocessor hardware target (Chapter 12, Section 2 in [Pto194]) was written for the OMA board, for automatic partitioning, code generation, and execution of an algorithm from a block diagram specification. A simple heterogeneous multiprocessor target was also written in Ptolemy for the OMA and S-56X combination; this target generates DSP56000 code for the S-56X card, and generates DSP96000 multiprocessor code for the OMA.

### **3.6 Ordered I/O and parameter control**

We have implemented a mechanism whereby I/O can be done over the shared bus. We make use of the fact that I/O for DSP applications is periodic; samples (or blocks of samples) typically arrive at constant, periodic intervals, and the processed output is again required (by, say, a D/A convertor) at periodic intervals. With this observation, it is in fact possible to schedule the I/O operations within the multiprocessor schedule, and consequently determine when, relative to the other shared bus accesses due to IPC, the shared bus is required for I/O. This

allows us to include bus accesses for I/O in the bus access order list. In our particular implementation, I/O is implemented as shared address locations that address the Tx and Rx registers in the Xilinx chip (section 3.4.5), which in turn communicate with the S-56X board; a processor accesses these registers as if they were a part of shared memory. It obtains access to these registers when the transaction controller grants access to the shared bus; bus grants for the purpose of I/O are taken into account when constructing the access order list. Thus we order access to shared I/O resources much as we order access to the shared bus and memory.

We also experimented with application of the ordered memory access idea to run time parameter control. By run time parameter control we mean controlling parameters in the DSP algorithm (gain of some component, bit-rate of a coder, pitch of synthesized music sounds, etc.) while the algorithm is running in real time on the hardware. Such a feature is obviously very useful and sometimes indispensable. Usually, one associates such parameter control with an asynchronous user input: the user changes a parameter (ideally by means of a suitable GUI on his or her computer) and this change causes an interrupt to occur on a processor, and the interrupt handler then performs the appropriate operations that cause the parameter change that the user requested.

For the OMA architecture, however, unpredictable interrupts are not desirable, as was noted earlier in this chapter; on the other hand shared I/O and IPC are relatively inexpensive owing to the OT mechanism. To exploit this trade-off, we implemented the parameter control in the following fashion: The S-56X host handles the task of accepting user interrupts; whenever a parameter is altered, the DSP56000 on the S-56X card receives an interrupt and it modifies a particular location in its memory (call it  $M$ ). The OMA board on the other hand receives the contents of  $M$  on every schedule period, whether  $M$  was actually modified or not. Thus the OMA processors never “see” a user created interrupt; they in essence update the parameter corresponding to the value stored in  $M$  in every iteration of

the dataflow graph. Since reading in the value of  $M$  costs two instruction cycles, the overhead involved in this scheme is minimal.

An added practical advantage of the above scheme is that the tcl/tk [Ous94] based GUI primitives that have been implemented in Ptolemy for the S-56X (see “CG56 Domain” in Volume 1 of [Ptol94]) can be directly used with the OMA board for parameter control purposes.

## **3.7 Application examples**

### **3.7.1 Music synthesis**

The Karplus-Strong algorithm is a well known approach for synthesizing the sound of a plucked string. The basic idea is to pass a noise source in a feedback loop containing a delay, a low pass filter, and a multiplier with a gain of less than one. The delay determines the pitch of the generated sound, and the multiplier gain determines the rate of decay. Multiple voices can be generated and combined by implementing one feedback loop for each voice and then adding the outputs from all the loops. If we want to generate sound at a sampling rate of 44.1 KHz (compact disc sampling rate), we can implement 7 voices on a single processor in real time using the blocks from the Ptolemy DSP96000 code generation library (CG96). These 7 voices consume 370 instruction cycles out of the 380 instruction cycles available per sample period.

Using four processors on the OMA board, we implemented 28 voices in real time. The hierarchical block diagram for this is shown in Fig. 3.16. The resulting schedule is shown in Fig. 3.17. The makespan for this schedule is 377 instruction cycles, which is just within the maximum allowable limit of 380. This schedule uses 15 IPCs, and is therefore not communication intensive. Even so, a higher IPC cost than the 3 instruction cycles the OMA architecture affords us would not allow this schedule to execute in real time at a 44.1 KHz sampling rate, because there is only a 3 instruction cycle margin between the makespan of this

schedule and the maximum allowable makespan. To schedule this application, we employed Hu-level scheduling along with manual assignment of some of the blocks.

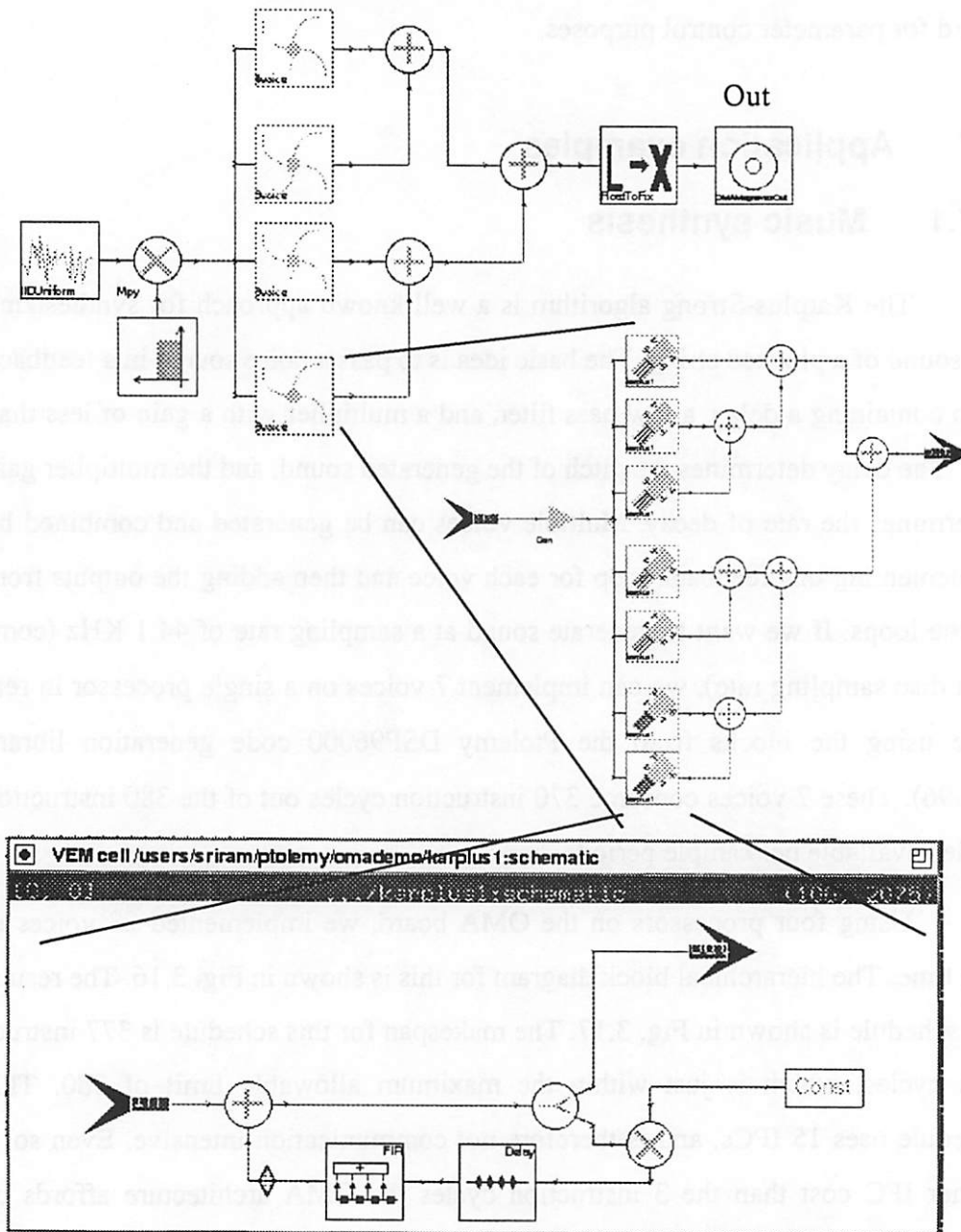


Figure 3.16. Hierarchical specification of the Karplus-Strong algorithm in 28 voices.

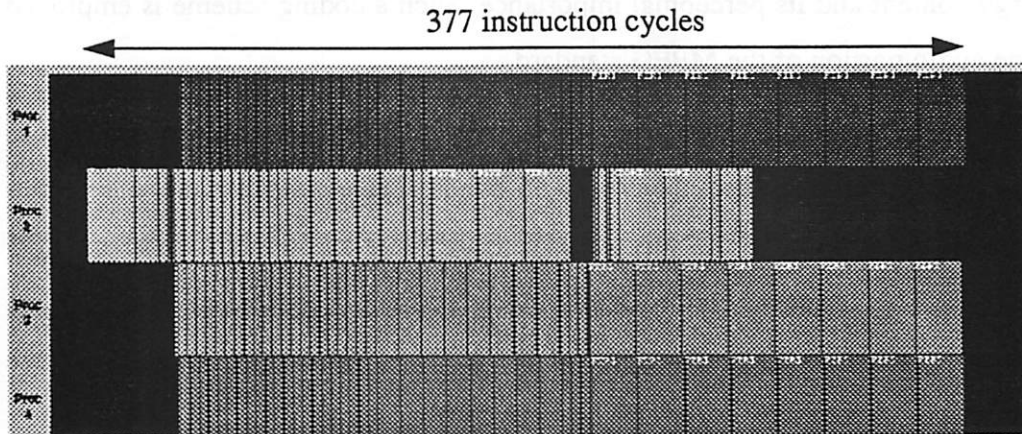


Figure 3.17. Four processor schedule for the Karplus-Strong algorithm in 28 voices. Three processors are assigned 8 voices each, the fourth (Proc 1) is assigned 4 voices along with the noise source.

### 3.7.2 QMF filter bank

A Quadrature Mirror Filter (QMF) bank consists of a set of *analysis* filters used to decompose a signal (usually audio) into frequency bands, and a bank of *synthesis* filters is used to reconstruct the decomposed signal [Vai93]. In the analysis bank, a filter pair is used to decompose the signal into high pass and low pass components, which are then decimated by a factor of two. The low pass component is then decomposed again into low pass and high pass components, and this process proceeds recursively. The synthesis bank performs the complementary operation of upsampling, filtering, and combining the high pass and low pass components; this process is again performed recursively to reconstruct the input signal. Fig. 3.18(a) shows a block diagram of a synthesis filter bank followed by an analysis bank.

QMF filter banks are designed such that the analysis bank cascaded with the synthesis bank yields a transfer function that is a pure delay (i.e. has unity response except for a delay between the input and the output). Such filter banks are also called *perfect reconstruction* filter banks, and they find applications in high quality audio compression; each frequency band is quantized according to its

energy content and its perceptual importance. Such a coding scheme is employed in the audio portion of the MPEG standard.

We implemented a perfect-reconstruction QMF filter bank to decompose audio from a compact disc player into 15 bands. The synthesis bank was implemented together with the analysis part. There are a total of 36 multirate filters of 18 taps each. This is shown hierarchically in Fig. 3.18(a). Note that delay blocks are required in the first 13 output paths of the analysis bank to compensate for the delay through successive stages of the analysis filter bank.

There are 1010 instruction cycles of computation per sample period in this example. Using Sih's Dynamic Level (DL) scheduling heuristic, we were able to achieve an average iteration period of 366 instruction cycles, making use of 40 IPCs. The schedule that is actually constructed (Gantt chart of Fig. 3.18(b)) operates on a block of 512 samples because these many samples are needed before all the actors in the graph fire at least once; this makes manual scheduling very difficult. We found that the DL heuristic performs close to 20% better than the Hu-level heuristic in this example, although the DL heuristic takes more than twice the time to compute the schedule compared to Hu-level.

### **3.7.3 1024 point complex FFT**

For this example, input data (1024 complex numbers) is assumed to be present in shared memory, and the transform coefficients are written back to shared memory. A single 96002 processor on the OMA board performs a 1024 point complex FFT in 3.0 milliseconds (ms). For implementing the transform on all four processors, we used the first stage of a radix four, decimation in frequency FFT computation, after which each processor independently performs a 256 point FFT. In this scheme, each processor reads all 1024 complex inputs at the beginning of the computation, combines them into 256 complex numbers on which it performs a 256 point FFT, and then writes back its result to shared memory using bit reversed addressing. The entire operation takes 1.0 ms. Thus we achieve a speedup



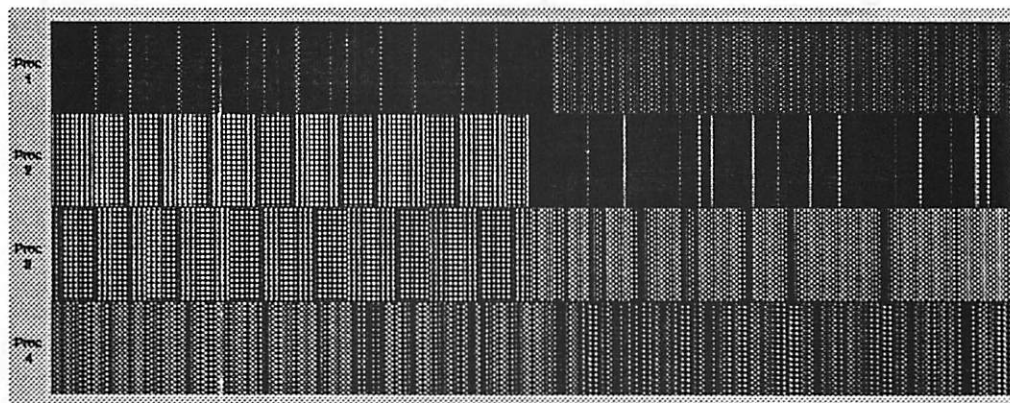
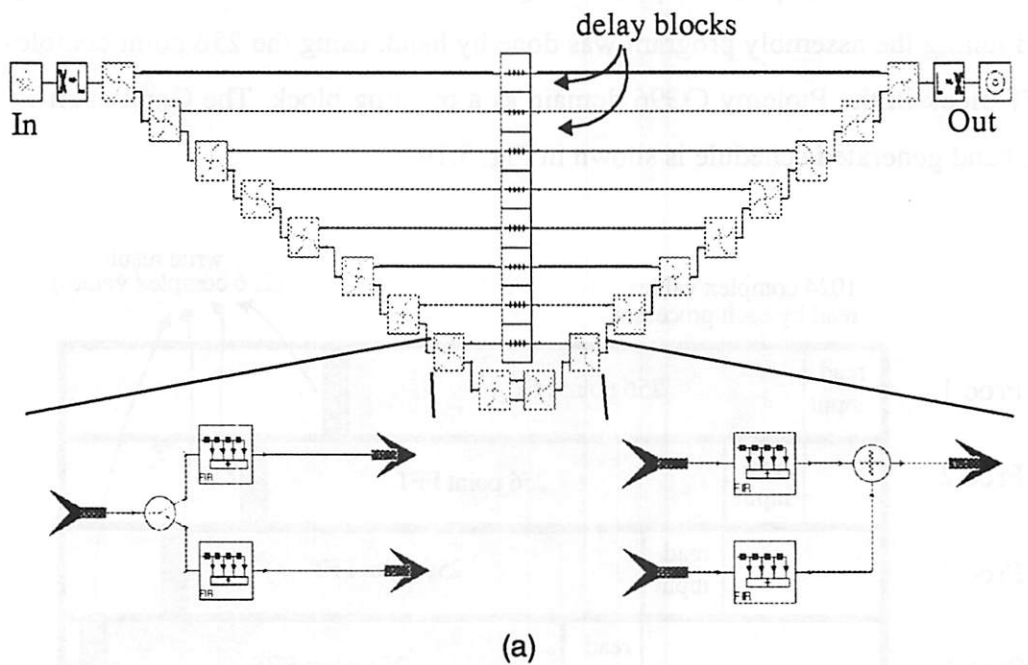


Figure 3.18. (a) Hierarchical block diagram for a 15 band analysis and synthesis filter bank. (b) Schedule on four processors (using Sih's DL heuristic [Sih90]).

of 3 over a single processor. This example is communication intensive; the throughput is limited by the available bus bandwidth. Indeed, if all processors had independent access to the shared memory (if the shared memory were 4-ported for example), we could achieve an ideal speedup of four, because each 256 point FFT is independent of the others except for data input and output.

For this example, data partitioning, shared memory allocation, scheduling, and tuning the assembly program was done by hand, using the 256 point complex FFT block in the Ptolemy CG96 domain as a building block. The Gantt chart for the hand generated schedule is shown in Fig. 3.19.

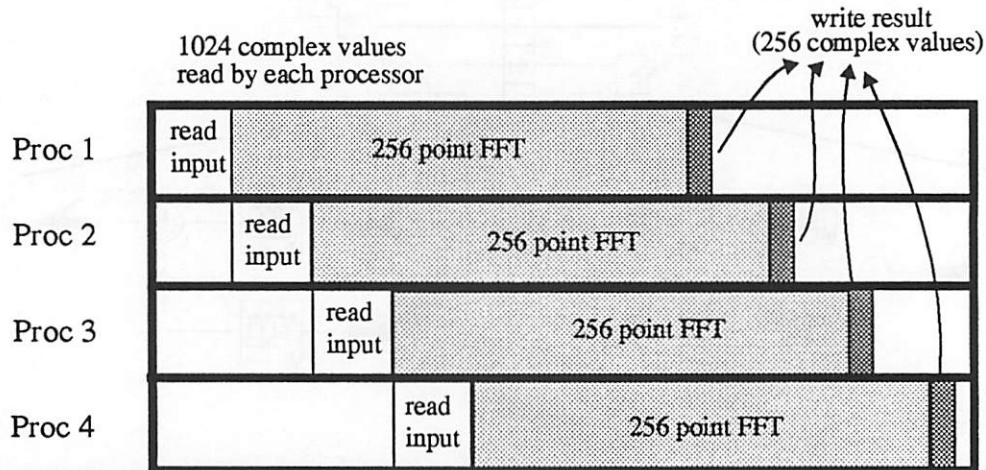


Figure 3.19. Schedule for the FFT example.

### 3.8 Summary

In this chapter we discussed the ideas behind the Ordered Transactions scheduling strategy. This strategy combines compile time analysis of the IPC pattern with simple hardware support to minimize interprocessor communication overhead. We discussed the hardware design and implementation details of a prototype shared bus multiprocessor — the Ordered Memory Access architecture — that uses the ordered transactions principle to statically assign the sequence of processor accesses to shared memory. External I/O and user control inputs can also be taken into account when scheduling accesses to the shared bus. We also discussed the software interface details of the prototype and presented some applications that were implemented on the OMA prototype.

# 4

---

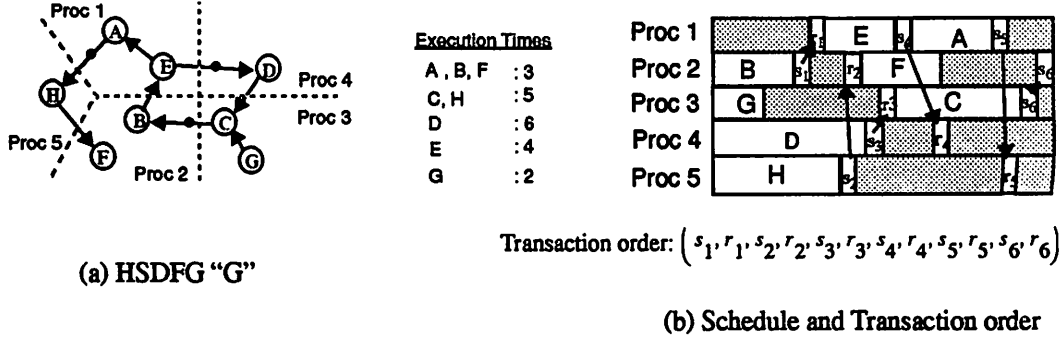
## AN ANALYSIS OF THE OT STRATEGY

---

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

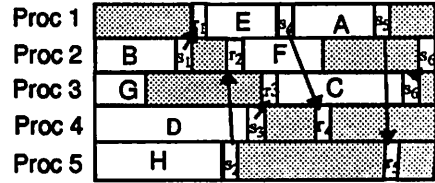
The effect of imposing a transaction order on a self-timed schedule is best illustrated by the following example. Let us assume that we use the dataflow graph and its schedule that was introduced in Chapter 1 (Fig. 1.2), and that we enforce the transaction order of Fig. 3.1; we reproduce these for convenience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

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



Execution Times

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



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

Figure 4.1. Fully-static schedule on five processors

signed to it), we get the “unfolded” schedule of Fig. 4.2; successive iterations of the HSDFG overlap in a natural manner. This is of course an idealized scenario where IPC costs are ignored; we do so to avoid unnecessary detail in the diagram, since IPC costs can be included in our analysis in a straightforward manner. Note that the ST schedule in Fig. 4.2 eventually settles to a periodic pattern consisting of two iterations of the HSDFG; the average iteration period under the self-timed schedule is 9 units. The average iteration period (which we will refer to as  $T_{ST}$ ) for such an idealized (zero IPC cost) self-timed schedule represents a **lower bound** on the iteration period achievable by *any* schedule that maintains the same processor assignment and actor ordering. This is because the only run time constraint on processors that the ST schedule imposes is due to data dependencies: each processor executes actors assigned to it (including the communication actors) according to the compile time determined order. An actor at the head of this ordered list is executed as soon as data is available for it. Any other schedule that maintains the same processor assignment and actor ordering, and respects data precedences in  $G$ , cannot result in an execution where actors fire earlier than they do in the idealized ST schedule. In particular, the overlap of successive iterations of the HSDFG in the idealized ST schedule ensures that  $T_{ST} \leq T_{FS}$  in general.

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

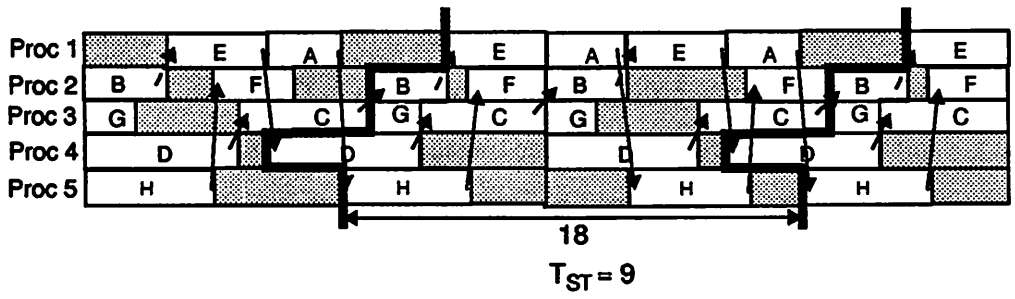


Figure 4.2. Self-timed schedule

successive iterations are ordered differently: in the first iteration the order in which IPCs occur is indeed  $(s_1, r_1, s_2, r_2, s_3, r_3, s_4, r_4, s_5, r_5, s_6, r_6)$ ; once the schedule settles into a periodic pattern, the order alternates between:

$$(s_3, r_3, s_1, r_1, s_2, r_2, s_4, r_4, s_6, r_6, s_5, r_5)$$

and

$$(s_1, r_1, s_3, r_3, s_4, r_4, s_2, r_2, s_5, r_5, s_6, r_6) .$$

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

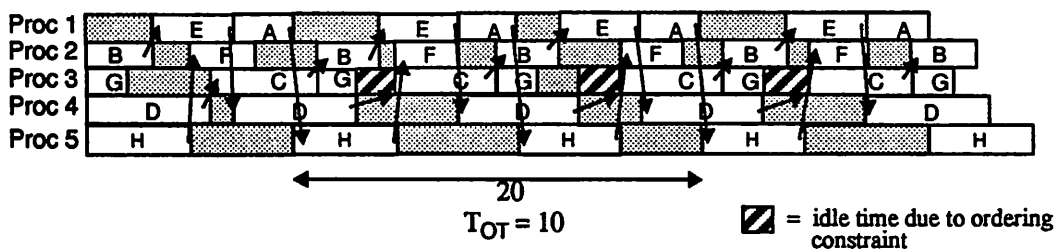


Figure 4.3. Schedule evolution when the transaction order of Fig. 3.1 is enforced

$T_{FS} \geq T_{OT} \geq T_{ST}$ : the ST schedule only has assignment and ordering constraints, the OT schedule has the transaction ordering constraints in addition to the constraints in the ST schedule, whereas the FS schedule has exact timing constraints that subsume the constraints in the ST and OT schedules. The question we would like to answer is: is it possible to choose the transaction ordering more intelligently than the straightforward one (obtained by sorting) chosen in Fig. 3.1?

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

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

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

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

processor communication. For example, the self-timed schedule in Fig. 4.1 (b) can be modelled by the IPC graph in Fig. 4.4.

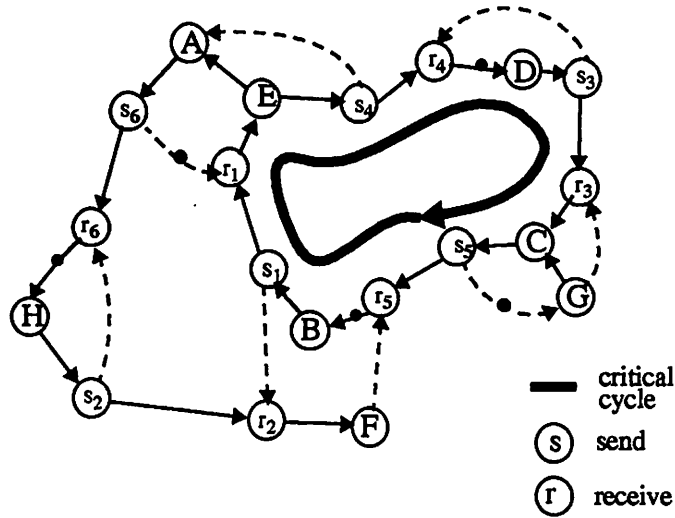


Figure 4.4. The IPC graph for the schedule in Fig. 4.1.

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

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

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

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

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

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

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

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



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

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

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

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

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

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

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

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

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

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

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

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

blocked.

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

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

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

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

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

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

of  $G_{ipc}$ : That is,

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

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

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

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

## 4.2 Execution time estimates

If we only have execution time estimates available instead of exact values, and we set  $t(v)$  in the previous section to be these estimated values, then we obtain the *estimated* iteration period by calculating  $MCM$ . Henceforth we will assume that we know the **estimated throughput**  $\frac{1}{MCM}$  calculated by setting the  $t(v)$  values to the available timing estimates. As mentioned in Chapter 1, for most practical sce-

narios, we can only assume such compile time estimates, rather than clock-cycle accurate execution time estimates. In fact this is the reason we had to rely on self-timed scheduling, and we proposed the ordered transaction strategy as a means of achieving efficient IPC despite the fact that we do not assume knowledge of exact actor execution times.

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

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

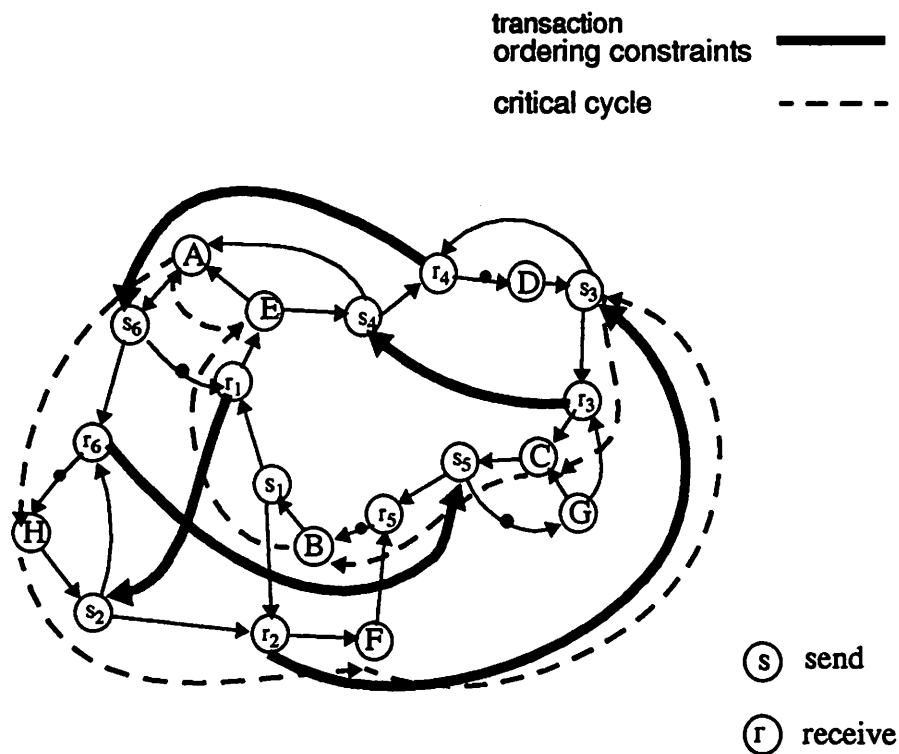


Figure 4.5. Transaction ordering constraints

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

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

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

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

#### 4.4 Periodicity

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

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

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

Thus after a “transient” that lasts  $K$  iterations, the ST schedule evolves into a periodic pattern. The periodic pattern itself spans  $N$  iterations; we call  $N$  the **periodicity**. The periodicity depends on the number of delays in the critical cycles of  $G_{ipc}$ ; it can be as high as the number least common multiple of the number of delays in

the critical cycles of  $G_{ipc}$  [Bacc92]. For example, the IPC graph of Fig. 4.4 has one critical cycle with two delays on it, and thus we see a periodicity of two for the schedule in Fig. 4.2. The “transient” region defined by  $K$  (which is 1 in Fig. 4.2) can also be exponential.

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

Returning to the problem of determining the optimal transaction order, one possible scheme is to derive the transaction order from the repeating pattern that the ST schedule settles into. That is, instead of using the transaction order of Fig. 3.1, if we enforce the transaction order that repeats over two iterations in the evolution of the ST schedule of Fig. 4.2, the OT schedule would “mimic” the ST schedule exactly, and we would obtain an OT schedule that performs as well as the ideal ST schedule, and yet involves low IPC costs in practice. However, as pointed out above, the number of iterations that the repeating pattern spans depends on the critical cycles of  $G_{ipc}$ , and it can be exponential in the size of the HSDFG [Bacc92].

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

## 4.5 Optimal order

In this section we show how to determine an order  $O^*$  on the IPCs in the schedule such that imposing  $O^*$  yields an OT schedule that has iteration period within one unit of the ideal ST schedule ( $T_{ST} \leq T_{OT} \leq \lceil T_{ST} \rceil$ ). Thus imposing the order we determine results in essentially no loss in performance over an unrestrained schedule, and at the same time we get the benefit of cheaper IPC.

Our approach to determining the transaction order  $O^*$  is to modify a given fully-static schedule so that the resulting FS schedule has  $T_{FS}$  equal to  $\lceil T_{ST} \rceil$ , and then to derive the transaction order from that modified schedule. Intuitively it appears that, for a given processor assignment and ordering of actors on processors, the ST approach *always* performs better than the FS or OT approaches ( $T_{FS} > T_{OT} > T_{ST}$ ) simply because it allows successive iterations to overlap. The following result, however, tells us that it is always possible to modify any given fully-static schedule so that it performs nearly as well as its self-timed counterpart. Stated more precisely:

**Claim 4.1:** Given a fully-static schedule  $S \equiv \{\sigma_p(v), \sigma_t(v), T_{FS}\}$ , let  $T_{ST}$  be the average iteration period for the corresponding ST schedule (as mentioned before,  $T_{FS} \geq T_{ST}$ ). Suppose  $T_{FS} > T_{ST}$ ; then, there exists a valid fully-static schedule  $S'$  that has the same processor assignment as  $S$ , the same order of execution of actors on each processor, but an iteration period of  $\lceil T_{ST} \rceil$ . That is,



$S' \equiv \{ \sigma_p(v), \sigma'_t(v), \lceil T_{ST} \rceil \}$  where, if actors  $v_i, v_j$  are on the same processor (i.e.  $\sigma_p(v_i) = \sigma_p(v_j)$ ) then  $\sigma_t(v_i) > \sigma_t(v_j) \Rightarrow \sigma'_t(v_i) > \sigma'_t(v_j)$ . Furthermore,  $S'$  is obtained by solving the following set of linear inequalities for  $\sigma'_t$ :

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

**Proof:** Let  $S'$  have a period equal to  $T$ . Then, under the schedule  $S'$ , the  $k$ th starting time of actor  $v_i$  is given by:

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

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

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

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

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

That is:

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

Note that the construction of  $G_{ipc}$  ensures that processor assignment constraints are automatically met: if  $\sigma_p(v_i) = \sigma_p(v_j)$  and  $v_i$  is to be executed immediately after  $v_j$  then there is an edge  $(v_j, v_i)$  in  $G_{ipc}$ . The relations in Eqn. 4-6 represent a system of  $|E_{ipc}|$  inequalities in  $|V|$  unknowns (the quantities  $\sigma'_t(v_i)$ ).

The system of inequalities in Eqn. 4-6 is a difference constraint problem that can be solved in polynomial time ( $O(|E_{ipc}||V|)$ ) using the Bellman-Ford shortest-path algorithm [Law76][Corm92]. The details of this approach are well described in [Corm92]; the essence of it is to construct a constraint graph that has one vertex for each unknown  $\sigma'_t(v_i)$ . Each difference constraint is then represented by an

edge between the vertices corresponding to the unknowns, and the weight on that edge is set to be equal to the RHS of the difference constraint. A “dummy” vertex is added to the constraint graph, and zero weight edges are added from the dummy vertex to each of the remaining vertices in the constraint graph. Then, setting the value of  $\sigma'_i(v_i)$  to be the weight of the shortest path from the dummy vertex to the vertex that corresponds to  $\sigma'_i(v_i)$  in the constraint graph results in a solution to the system of inequalities, if indeed a solution exists. A feasible solution exists if and only if the constraint graph does not contain a negative weight cycle [Corm92], which is equivalent to the following condition:

$$T \geq \max_{\text{cycle } C \text{ in } G_{ipc}} \left\{ \frac{\sum_{v \in C} t(v)}{D(C)} \right\}; \text{ and, from Eqn. 4-3, this is equivalent to } T \geq T_{ST}.$$

If we set  $T = \lceil T_{ST} \rceil$ , then the right hand sides of the system of inequalities in 4-6 are integers, and the Bellman-Ford algorithm yields integer solutions for  $\sigma'_i(v)$ . This is because the weights on the edges of the constraint graph, which are equal to the RHS of the difference constraints, are integers if  $T$  is an integer; consequently, the shortest paths calculated on the constraint graph are integers.

Thus  $S' \equiv \{\sigma_p(v), \sigma'_i(v), \lceil T_{ST} \rceil\}$  is a valid fully-static schedule. *QED.*

*Remark:* Claim 4.1 essentially states that an FS schedule can be modified by skewing the relative starting times of processors so that the resulting schedule has iteration period less than  $(T_{ST} + 1)$ ; the resulting iteration period lies within one time unit of its lower bound for the specified processor assignment and actor ordering. It is possible to unfold the graph and generate a fully-static schedule with average period exactly  $T_{ST}$ , but the resulting increase in code size is usually not worth the benefit of (at most) one time unit decrease in the iteration period. Recall that a “time unit” is essentially the clock period; therefore, one time unit can usually be neglected.

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

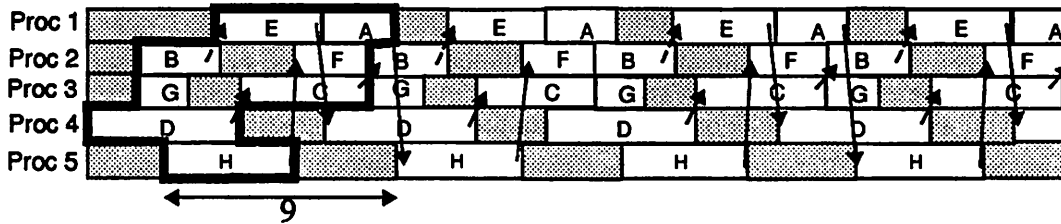


Figure 4.6. Modified schedule  $S'$

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

Claim 4.1 may not seem useful at first sight: why not obtain a fully-static schedule that has a period  $\lceil T_{ST} \rceil$  to begin with, thus eliminating the post-processing step suggested in Claim 4.1? Recall that an FS schedule is usually obtained using heuristic techniques that are either based on blocked non-overlapped scheduling (which use critical path based heuristics) [Sih91] or are based on overlapped scheduling techniques that employ list scheduling heuristics [deGroot92][Lam88]]. None of these techniques guarantee that the generated FS schedule will have an iteration period within one unit of the period achieved if the same schedule were run in a self-timed manner. Thus for a schedule generated using any of these techniques, we might be able to obtain a gain in performance, essentially for free, by performing the post-processing step suggested in Claim 4.1. What we propose can therefore be added as an efficient post-processing step in existing schedulers. Of course, an exhaustive search procedure like the one proposed in [Schw85] will certainly find the schedule  $S'$  directly.

We set the transaction order  $O'$  to be the transaction order suggested by the modified schedule  $S'$  (as opposed to the transaction order from  $S$  used in Fig. 3.1). Thus  $O' = (s_1, r_1, s_3, r_3, s_2, r_2, s_4, r_4, s_6, r_6, s_5, r_5)$ . Imposing the transaction order  $O'$  as in Fig. 4.6 results in  $T_{OT}$  of 9 units instead of 10 that we get if the transaction order of Fig. 3.1 is used. Under the transaction order specified by  $S'$ ,  $T_{ST} \leq T_{OT} \leq \lceil T_{ST} \rceil$ ; thus imposing the order  $O'$  ensures that the average period is within one unit of the unconstrained ST strategy. Again, unfolding may be required to obtain a transaction ordered schedule that has period exactly equal to  $T_{ST}$ , but the extra cost of a larger controller (to enforce the transaction ordering) outweighs the small gain of at most one unit reduction in the iteration period. Thus for all practical purposes  $O'$  is the *optimal* transaction order. The “optimality” is in the sense that the transaction order  $O'$  we determine statically is the best possible one, given the timing information available at compile time.

## 4.6 Effects of changes in execution times

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

Compile time estimates of actor execution times may be different from their actual values at run time due to errors in estimating execution times of actors that otherwise have fixed execution times, and due to actors that display run time variations in their execution times, because of conditionals or data-dependent loops within them for example. The first case is simple to model, and we will show in section 4.6.1 how the throughput of a given self-timed schedule changes as a function

of actor execution times. The second case is inherently difficult; how do we model run time changes in execution times due to data-dependencies, or due to events such as error handling, cache misses, and pipeline effects? In section 4.6.2 below we briefly discuss a very simple model for such run time variations; we assume actors have random execution times according to some known probability distribution. We conclude that analysis of even such a simple model for the expected value of the throughput is often intractable, and we discuss efficiently computable upper and lower bounds for the expected throughput.

### 4.6.1 Deterministic case

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

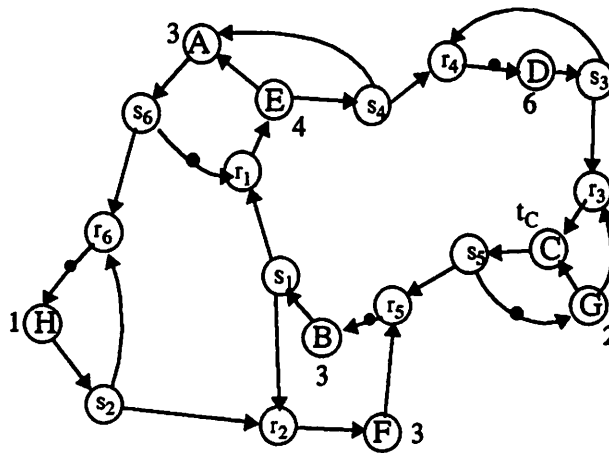


Figure 4.7.  $G_{ipc}$ , actor C has execution time  $t_C$ , constant over all invocations of C

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

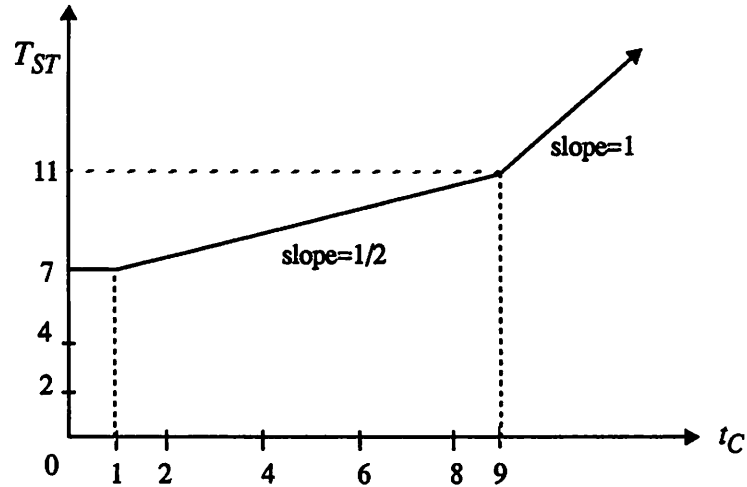


Figure 4.8.  $T_{ST}(t_C)$

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

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

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

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

**Definition 4.4:** A function  $f(x)$  is said to be convex over an interval  $(a, b)$  if for

every  $x_1, x_2 \in (a, b)$  and  $0 \leq \lambda \leq 1$ ,

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

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

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

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

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

## 4.6.2 Modeling run time variations in execution times

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

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

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

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

The normalization (with respect to actor  $v_1$  in the above case) is done to make the state space finite; the number of distinct values that the vector  $s$  (as defined above) can assume is shown to be finite in [Ols90]. The states of the Markov



chain correspond to each of the distinct values of  $s$ . The average iteration period, which is defined as:

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

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

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

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

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

If we assume that all actors have exponentially distributed execution times, then the system can be analyzed using continuous-time Markov chains [Moll82]. This is done by exploiting the memoryless property of the exponential distribution: when an actor fires, the state of the system at any moment does not depend on how long that actor has spent executing its function; the state changes only when that actor completes execution. The number of states for such a system is equal to the num-

ber of different valid token configurations on the edges of the dataflow graph, where by “valid” we imply any token configuration that can be reached by a sequence of firings of enabled actors in the HSDFG. This is also equal to the number of *valid retimings* [Lei91] that exist for the HSDFG. This number, unfortunately, is again exponential in the size of the HSDFG.

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

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

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

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

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

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

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

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

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

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

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

Equation (4-7) summarizes the useful bounds we know for expected value of the iteration period for graphs that contain actors with random execution times. It should be noted that good upper bounds on  $E [T]$  are not known. Rajsbaum and Sidi propose upper bounds for exponentially distributed execution times [Rajs94]; these upper bounds are typically more than twice the exact value of  $E [T]$  , and hence not very useful in practice. We attempted to simplify the Markov chain model

(i.e. reduce the number of states) for the self-timed execution of a stochastic HSDFG by representing such an execution by a set of self-timed schedules of deterministic HSDFGs, between which the system makes transitions randomly. This representation reduces the number of states of the Markov chain to the number of different deterministic graphs that arise from the stochastic HSDFG. We were able to use this idea to determine an upper bound for  $E[T]$ ; however, this bound also proved to be too loose in general (hence we omit the details of this construction here).

### 4.6.3 Implications for the OT schedule

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

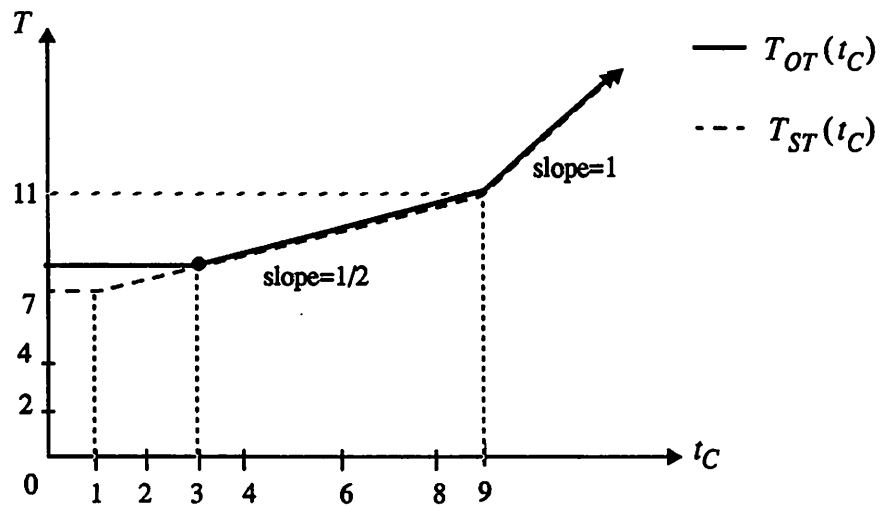


Figure 4.10.  $T_{ST}(t_C)$  and  $T_{OT}(t_C)$

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

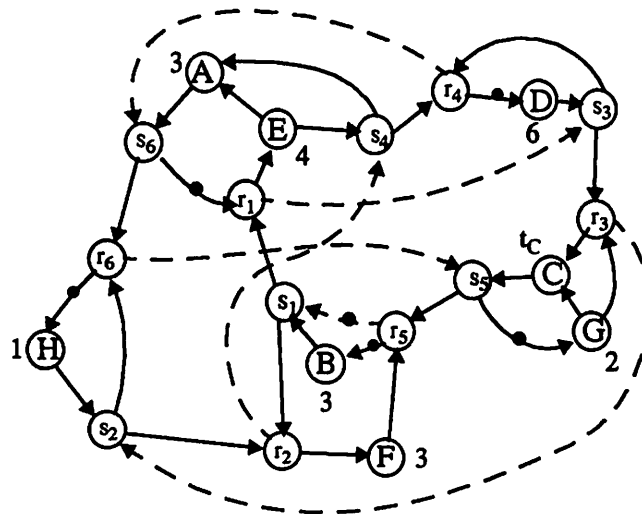


Figure 4.9.  $G_{ipc}$  with transaction ordering constraints represented as dashed lines

## 4.7 Summary

In this chapter we presented a quantitative analysis of ST and OT schedules and showed how to determine the effects of imposing a transaction order on an ST schedule. If the actual execution times do not deviate significantly from the estimated values, the difference in performance of the ST and OT strategies is minimal. If the execution times do in fact vary significantly, then even an ST strategy is not practical; it then becomes necessary to use a more dynamic strategy such as static assignment or fully dynamic scheduling [Lee89] to make the best use of computing resources. Under the assumption that the variations in execution times are small enough so that an ST or an OT strategy is viable, we argue that it is in fact wiser to use the OT strategy rather than ST because of the cheaper IPC of the OT strategy. This is because we can determine the transaction order  $O^*$  such that the ordering constraints do not sacrifice performance; if the execution times of actors are close to their estimates, the OT schedule with  $O^*$  as the transaction order has iteration period close to the minimum achievable period  $T_{ST}$ . Thus we make the best possible use of compile time information when we determine the transaction order  $O^*$ .

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

# 5

---

## MINIMIZING SYNCHRONIZATION COSTS IN SELF-TIMED SCHEDULES

---

The previous three chapters dealt with the Ordered Transactions strategy, which is a hardware approach to reducing IPC and synchronization costs in self-timed schedules. In this chapter we present algorithms that minimize synchronization costs in the final implementation of a given self-timed schedule, and we do not assume the availability of any hardware support for employing the OT approach.

Recall that the self-timed scheduling strategy introduces synchronization checks whenever processors communicate. A straightforward implementation of a self-timed schedule would require that for each inter-processor communication (IPC), the sending processor ascertain that the buffer it is writing to is not full, and the receiver ascertain that the buffer it is reading from is not empty. The processors block (suspend execution) when the appropriate condition is not met. Such sender-receiver synchronization can be implemented in many ways depending on the particular hardware platform under consideration: in shared memory machines, such synchronization involves testing and setting semaphores in shared memory; in machines that support synchronization in hardware (such as barriers), special synchronization instructions are used; and in the case of systems that consist of a mix of programmable processors and custom hardware elements, synchronization is achieved by employing interfaces that support blocking reads and writes.

In each type of platform, each IPC that requires a synchronization check

costs performance, and sometimes extra hardware complexity. Semaphore checks cost execution time on the processors, synchronization instructions that make use of special synchronization hardware such as barriers also cost execution time, and blocking interfaces between a programmable processor and custom hardware in a combined hardware/software implementations require more hardware than non-blocking interfaces [Huis93].

In this chapter we present algorithms and techniques that reduce the rate at which processors must access shared memory for the purpose of synchronization in multiprocessor implementations of SDF programs. One of the procedures we present, for example, detects when the objective of one synchronization operation is guaranteed as a side effect of other synchronizations in the system, thus enabling us to eliminate such superfluous synchronization operations. The optimization procedure that we propose can be used as a post-processing step in any static scheduling technique (any one of the techniques presented in Chapter 1, section 1.2) for reducing synchronization costs in the final implementation. As before we assume that “good” estimates are available for the execution times of actors and that these execution times rarely display large variations so that self-timed scheduling is viable for the applications under consideration. If additional timing information is available, such as guaranteed upper and lower bounds on the execution times of actors, it is possible to use this information to further optimize synchronizations in the schedule. However, use of such timing bounds will be left as future work; we mention this again in Chapter 7.

This chapter is a part of ongoing research in collaboration with Dr. Shuvra Bhattacharyya, who is a Research Scientist at Hitachi America Ltd.

## 5.1 Related work

Among the prior art that is most relevant to this chapter is the *barrier-MIMD* principle of Dietz, Zaafrani, and O’Keefe, which is a combined hardware and software solution to reducing run-time synchronization overhead [Dietz92]. In



this approach, a shared-memory MIMD computer is augmented with hardware support that allows arbitrary subsets of processors to synchronize precisely with respect to one another by executing a synchronization operation called a *barrier*. If a subset of processors is involved in a barrier operation, then each processor in this subset will wait at the barrier until all other processors in the subset have reached the barrier. After all processors in the subset have reached the barrier, the corresponding processes resume execution in *exact synchrony*.

In [Dietz92], the barrier mechanism is applied to minimize synchronization overhead in a self-timed schedule with hard lower and upper bounds on the task execution times. The execution time ranges are used to detect situations where the earliest possible execution time of a task that requires data from another processor is guaranteed to be later than the latest possible time at which the required data is produced. When such an inference cannot be made, a barrier is instantiated between the sending and receiving processors. In addition to performing the required data synchronization, the barrier resets (to zero) the uncertainty between the relative execution times for the processors that are involved in the barrier, and thus enhances the potential for subsequent timing analysis to eliminate the need for explicit synchronizations.

The techniques of barrier MIMD do not apply to the problem that we address because they assume that a hardware barrier mechanism exists; they assume that tight bounds on task execution times are available; they do not address iterative, self-timed execution, in which the execution of successive iterations of the dataflow graph can overlap; and even for non-iterative execution, there is no obvious correspondence between an optimal solution that uses barrier synchronizations and an optimal solution that employs decoupled synchronization checks at the sender and receiver end (**directed synchronization**). This last point is illustrated in Fig. 5.1. Here, in the absence of execution time bounds, an optimal application of barrier synchronizations can be obtained by inserting two barriers — one barrier across  $A_1$  and  $A_3$ , and the other barrier across  $A_4$  and  $A_5$ . This is illustrated in Figure 5.1(c). However, the corresponding collection of directed synchro-

nizations ( $A_1$  to  $A_3$ , and  $A_5$  to  $A_4$ ) is not sufficient since it does not guarantee that the data required by  $A_6$  from  $A_1$  is available before  $A_6$  begins execution.

In [Sha89], Shaffer presents an algorithm that minimizes the number of directed synchronizations in the self-timed execution of a dataflow graph. However, this work, like that of Dietz *et al.*, does not allow the execution of successive iterations of the dataflow graph to overlap. It also avoids having to consider dataflow edges that have delay. The technique that we present for removing redundant synchronizations can be viewed as a generalization of Shaffer's algorithm to handle delays and overlapped, iterative execution, and we will discuss this further in section 5.6. The other major techniques that we present for optimizing synchronization — handling the feedforward edges of the *synchronization graph* (to be defined in section 5.4.2), discussed in section 5.7, and “resynchronization”, defined and addressed in sections 5.9 and the appendix — are fundamentally different from Shaffer's technique since they address issues that are specific to our

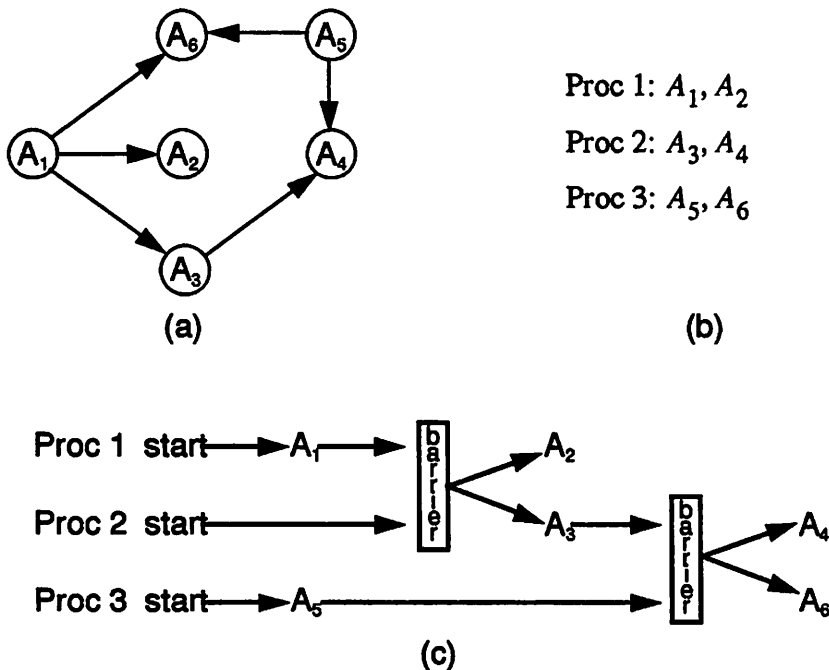


Figure 5.1. (a) An HSDFG (b) A three-processor self-timed schedule for (a). (c) An illustration of execution under the placement of barriers.

more general context of overlapped, iterative execution.

As discussed in Chapter 1, section 1.2.2, a multiprocessor executing a self-timed schedule is one where each processor is assigned a sequential list of actors, some of which are *send* and *receive* actors, which it executes in an infinite loop. When a processor executes a communication actor, it synchronizes with the processor(s) it communicates with. Thus exactly when a processor executes each actor depends on when, at run time, all input data for that actor is available, unlike the fully-static case where no such run time check is needed. In this chapter we use “processor” in slightly general terms: a processor could be a programmable component, in which case the actors mapped to it execute as software entities, or it could be a hardware component, in which case actors assigned to it are implemented and execute in hardware. See [Kala93] for a discussion on combined hardware/software synthesis from a single dataflow specification. Examples of application-specific multiprocessors that use programmable processors and some form of static scheduling are described in [Bork88][Koh90], which were also discussed in Chapter 1, section 1.3.

Inter-processor communication between processors is assumed to take place via shared memory. Thus the sender writes to a particular shared memory location and the receiver reads from that location. The shared memory itself could be global memory between all processors, or it could be distributed between pairs of processors (as a hardware FIFO queues or dual ported memory for example). Each inter-processor communication edge in our HSDFG thus translates into a buffer of a certain size in shared memory.

Sender-receiver synchronization is also assumed to take place by setting flags in shared memory. Special hardware for synchronization (barriers, semaphores implemented in hardware, etc.) would be prohibitive for the embedded multiprocessor machines for applications such as DSP that we are considering. Interfaces between hardware and software are typically implemented using memory-mapped registers in the address space of the programmable processor (again a kind of shared memory), and synchronization is achieved using flags that can be

tested and set by the programmable component, and the same can be done by an interface controller on the hardware side [Huis93].

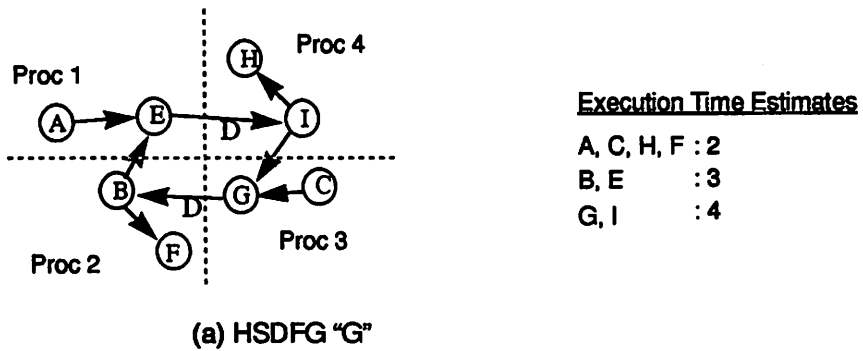
Under the model above, the benefits that our proposed synchronization optimization techniques offer become obvious. Each synchronization that we eliminate directly results in one less synchronization check, or, equivalently, one less shared memory access. For example, where a processor would have to check a flag in shared memory before executing a *receive* primitive, eliminating that synchronization implies there is no longer need for such a check. This translates to one less shared memory read. Such a benefit is especially significant for simplifying interfaces between a programmable component and a hardware component: a *send* or a *receive* without the need for synchronization implies that the interface can be implemented in a non-blocking fashion, greatly simplifying the interface controller. As a result, eliminating a synchronization directly results in simpler hardware in this case.

Thus the metric for the optimizations we present in this chapter is the total number of accesses to shared memory that are needed for the purpose of synchronization in the final multiprocessor implementation of the self-timed schedule. This metric will be defined precisely in section 5.5.

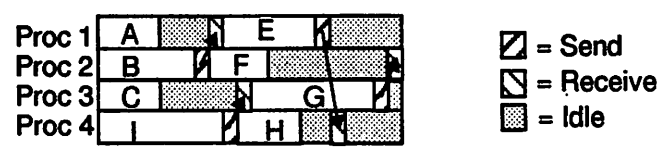
## 5.2 Analysis of self-timed execution

We model synchronization in a self-timed implementation using the IPC graph model introduced in the previous chapter. As before, an IPC graph  $G_{ipc}(V, E_{ipc})$  is extracted from a given HSDFG  $G$  and multi-processor schedule; Fig. 5.2 shows one such example, which we use throughout this chapter.

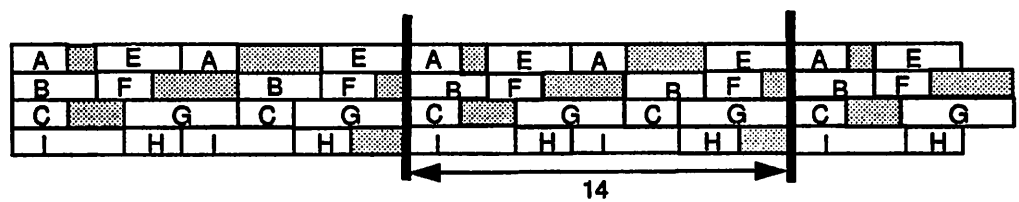
We will find it useful to partition the edges of the IPC graph in the following manner:  $E_{ipc} \equiv E_{int} \cup E_{comm}$ , where  $E_{comm}$  are the **communication edges** (shown dotted in Fig. 5.2(d)) that are directed from the send to the receive actors in  $G_{ipc}$ , and  $E_{int}$  are the “internal” edges that represent the fact that actors assigned to a particular processor (actors internal to that processor) are executed sequen-



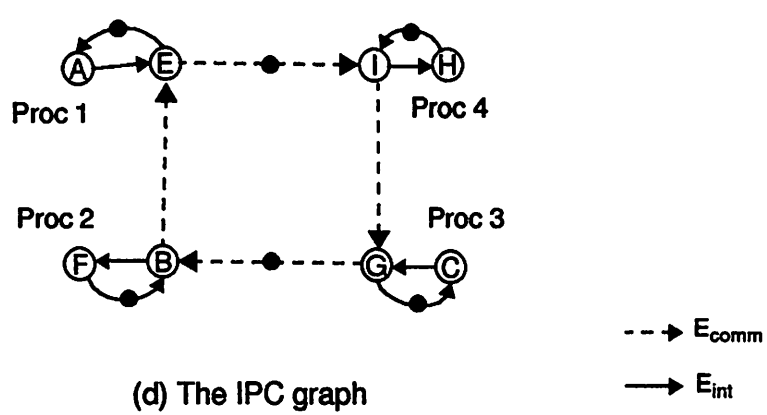
(a) HSDFG "G"



(b) Schedule on four processors



(c) Self-timed execution



(d) The IPC graph

Figure 5.2. Self-timed execution

tially according to the order predetermined by the self-timed schedule. A communication edge  $e \in E_{comm}$  in  $G_{ipc}$  represents two functions: 1) reading and writing of data values into the buffer represented by that edge; and 2) synchronization

between the sender and the receiver. As mentioned before, we assume the use of shared memory for the purpose of synchronization; the synchronization operation itself must be implemented using some kind of software *protocol* between the sender and the receiver. We discuss these synchronization protocols shortly.

### 5.2.1 Estimated throughput

Recall from Eqn. 4-3 that the average iteration period corresponding to a self-timed schedule with an IPC graph  $G_{ipc}$  is given by the maximum cycle mean of the graph  $MCM(G_{ipc})$ . If we only have execution time estimates available instead of exact values, and we set the execution times of actors  $t(v)$  to be equal to these estimated values, then we obtain the *estimated* iteration period by computing  $MCM(G_{ipc})$ . Henceforth we will assume that we know the **estimated throughput**  $MCM^{-1}$  calculated by setting the  $t(v)$  values to the available timing estimates.

In all the transformations that we present in the rest of the chapter, we will preserve the estimated throughput by preserving the maximum cycle mean of  $G_{ipc}$ , with each  $t(v)$  set to the estimated execution time of  $v$ . In the absence of more precise timing information, this is the best we can hope to do.

### 5.3 Strongly connected components and buffer size bounds

In dataflow semantics, the edges between actors represent infinite buffers. Accordingly, the edges of the IPC graph are potentially buffers of infinite size. However, from Lemma 4.1, every **feedback edge** (an edge that belongs to a strongly connected component, and hence to some cycle) can only have a finite number of tokens at any time during the execution of the IPC graph. We will call this constant the **self-timed buffer bound** of that edge, and for a feedback edge  $e$  we will represent this bound by  $B_{fb}(e)$ . Lemma 4.1 yields the following self-timed buffer bound:

$$B_{ff}(e) = \min ( \{ Delay (C) \mid C \text{ is a cycle that contains } e \} ) \quad (5-1)$$

**Feedforward edges** (edges that do not belong to any SCC) have no such bound on buffer size; therefore for practical implementations we need to *impose* a bound on the sizes of these edges. For example, Figure 5.3(a) shows an IPC graph where the communication edge  $(s, r)$  could be unbounded when the execution time of  $A$  is less than that of  $B$ , for example. In practice, we need to bound the



Figure 5.3. An IPC graph with a feedforward edge: (a) original graph (b) imposing bounded buffers.

buffer size of such an edge; we will denote such an “imposed” bound for a feedforward edge  $e$  by  $B_{ff}(e)$ . Since the effect of placing such a restriction includes “artificially” constraining  $src(e)$  from getting more than  $B_{ff}(e)$  invocations ahead of  $snk(e)$ , its effect on the estimated throughput can be modelled by adding a reverse edge that has  $m$  delays on it, where  $m = B_{ff}(e) - delay(e)$ , to  $G_{ipc}$  (grey edge in Fig. 5.3(b)). Since the addition of this edge introduces a new cycle in  $G_{ipc}$ , it has the potential to reduce the estimated throughput; to prevent such a reduction,  $B_{ff}(e)$  must be chosen to be large enough so that the maximum cycle mean remains unchanged upon adding the reverse edge with  $m$  delays.

Sizing buffers optimally such that the maximum cycle mean remains unchanged has been studied by Kung, Lewis and Lo in [Kung87], where the authors propose an integer linear programming formulation of the problem, with the number of constraints equal to the number of fundamental cycles in the HSDFG (potentially an exponential number of constraints).

An efficient albeit suboptimal procedure to determine  $B_{ff}$  is to note that if

$$B_{ff}(e) \geq \left\lceil \left( \sum_{x \in V} t(x) \right) / (MCM(G_{ipc})) \right\rceil$$

holds for each feedforward edge  $e$ , then the maximum cycle mean of the resulting graph does not exceed  $MCM$ .

Then, a binary search on  $B_{ff}(e)$  for each feedforward edge, while computing the maximum cycle mean at each search step and ascertaining that it is less than  $MCM(G_{ipc})$ , results in a buffer assignment for the feedforward edges. Although this procedure is efficient, it is suboptimal because the order that the edges  $e$  are chosen is arbitrary and may effect the quality of the final solution.

As we will see in section 5.7, however, imposing such a bound  $B_{ff}$  is a *naive* approach for bounding buffer sizes, because such a bound entails an added synchronization cost. In section 5.7 we show that there is a better technique for bounding buffer sizes; this technique achieves bounded buffer sizes by transforming the graph into a strongly connected graph by adding a minimal number of additional synchronization edges. Thus, in our final algorithm, we will not in fact find it necessary to use or compute these bounds  $B_{ff}$ .

## 5.4 Synchronization model

### 5.4.1 Synchronization protocols

We define two basic synchronization protocols for a communication edge based on whether or not the length of the corresponding buffer is guaranteed to be bounded from the analysis presented in the previous section. Given an IPC graph  $G$ , and a communication edge  $e$  in  $G$ , if the length of the corresponding buffer is not bounded — that is, if  $e$  is a feedforward edge of  $G$  — then we apply a synchronization protocol called **unbounded buffer synchronization (UBS)**, which guarantees that (a) an invocation of  $snk(e)$  never attempts to read data from an empty buffer; and (b) an invocation of  $src(e)$  never attempts to write data into the buffer unless the number of tokens in the buffer is less than some pre-specified limit  $B_{ff}(e)$ , which is the amount of memory allocated to the buffer as discussed



in the previous section.

On the other hand, if the topology of the IPC graph guarantees that the buffer length for  $e$  is bounded by some value  $B_{fb}(e)$  (the self-timed buffer bound of  $e$ ), then we use a simpler protocol, called **bounded buffer synchronization (BBS)**, that only explicitly ensures (a) above. Below, we outline the mechanics of the two synchronization protocols defined so far.

**BBS.** In this mechanism, a *write pointer*  $wr(e)$  for  $e$  is maintained on the processor that executes  $src(e)$ ; a *read pointer*  $rd(e)$  for  $e$  is maintained on the processor that executes  $snk(e)$ ; and a copy of  $wr(e)$  is maintained in some shared memory location  $sv(e)$ . The pointers  $rd(e)$  and  $wr(e)$  are initialized to zero and  $delay(e)$ , respectively. Just after each execution of  $src(e)$ , the new data value produced onto  $e$  is written into the shared memory buffer for  $e$  at offset  $wr(e)$ ;  $wr(e)$  is updated by the following operation —  $wr(e) \leftarrow (wr(e) + 1) \bmod B_{fb}(e)$ ; and  $sv(e)$  is updated to contain the new value of  $wr(e)$ . Just before each execution of  $snk(e)$ , the value contained in  $sv(e)$  is repeatedly examined until it is found to be *not equal* to  $rd(e)$ ; then the data value residing at offset  $rd(e)$  of the shared memory buffer for  $e$  is read; and  $rd(e)$  is updated by the operation  $rd(e) \leftarrow (rd(e) + 1) \bmod B_{fb}(e)$ .

**UBS.** This mechanism also uses the read/write pointers  $rd(e)$  and  $wr(e)$ , and these are initialized the same way; however, rather than maintaining a copy of  $wr(e)$  in the shared memory location  $sv(e)$ , we maintain a count (initialized to  $delay(e)$ ) of the number of unread tokens that currently reside in the buffer. Just after  $src(e)$  executes,  $sv(e)$  is repeatedly examined until its value is found to be less than  $B_{ff}(e)$ ; then the new data value produced onto  $e$  is written into the shared memory buffer for  $e$  at offset  $wr(e)$ ;  $wr(e)$  is updated as in BBS (except that the new value is not written to shared memory); and the count in  $sv(e)$  is incremented. Just before each execution of  $snk(e)$ , the value contained in  $sv(e)$  is repeatedly examined until it is found to be nonzero; then the data value residing at offset  $rd(e)$  of the shared memory buffer for  $e$  is read; the count in  $sv(e)$  is decremented; and  $rd(e)$  is updated as in BBS.

Note that we are assuming that there is enough shared memory to hold a separate buffer of size  $B_{ff}(e)$  for each feedforward communication edge  $e$  of  $G_{ipc}$ , and a separate buffer of size  $B_{fb}(e)$  for each feedback communication edge  $e$ . When this assumption does not hold, smaller bounds on some of the buffers must be imposed, possibly for feedback edges as well as for feedforward edges, and in general, this may require some sacrifice in estimated throughput. Note that whenever a buffer bound smaller than  $B_{fb}(e)$  is imposed on a feedback edge  $e$ , then a protocol identical to UBS must be used. The problem of optimally choosing which edges should be subject to stricter buffer bounds when there is a shortage of shared memory, and the selection of these stricter bounds is an interesting area for further investigation.

#### 5.4.2 The synchronization graph $G_s$

As we discussed in the beginning of this chapter, some of the communication edges in  $G_{ipc}$  need not have explicit synchronization, whereas others require synchronization, which need to be implemented either using the UBS protocol or the BBS protocol. All communication edges also represent buffers in shared memory. Thus we divide the set of communication edges as follows:  $E_{comm} \equiv E_s \cup E_r$ , where the edges  $E_s$  need explicit synchronization operations to be implemented, and the edges  $E_r$  need no explicit synchronization. We call the edges  $E_s$  **synchronization edges**.

Recall that a communication edge  $(v_j, v_i)$  of  $G_{ipc}$  represents the **synchronization constraint**:

$$start(v_i, k) \geq end(v_j, k - delay((v_j, v_i))) \quad \forall k > delay(v_j, v_i) \quad (5-2)$$

Thus, before we perform any optimization on synchronizations,  $E_{comm} \equiv E_s$  and  $E_r \equiv \phi$ , because every communication edge represents a synchronization point. However, in the following sections we describe how we can move certain edges from  $E_s$  to  $E_r$ , thus reducing synchronization operations in the final

implementation. At the end of our optimizations, the communication edges of the IPC graph fall into either  $E_s$  or  $E_r$ . At this point the edges  $E_s \cup E_r$  in  $G_{ipc}$  represent buffer activity, and must be implemented as buffers in shared memory, whereas the edges  $E_s$  represent synchronization constraints, and are implemented using the UBS and BBS protocols introduced in the previous section. For the edges in  $E_s$  the synchronization protocol is executed before the buffers corresponding to the communication edge are accessed so as to ensure sender-receiver synchronization. For edges in  $E_r$ , however, no synchronization needs to be done before accessing the shared buffer. Sometimes we will also find it useful to introduce synchronization edges without actually communicating data between the sender and the receiver (for the purpose of ensuring finite buffers for example), so that no shared buffers need to be assigned to these edges, but the corresponding synchronization protocol is invoked for these edges.

All optimizations that move edges from  $E_s$  to  $E_r$  must respect the synchronization constraints implied by  $G_{ipc}$ . If we ensure this, then we only need to implement the synchronization protocols for the edges in  $E_s$ . We call the graph  $G_s = (V, E_{int} \cup E_s)$  the **synchronization graph**. The graph  $G_s$  represents the synchronization constraints in  $G_{ipc}$  that need to be explicitly ensured, and the algorithms we present for minimizing synchronization costs operate on  $G_s$ . Before any synchronization related optimizations are performed  $G_s \equiv G_{ipc}$ , because  $E_{comm} \equiv E_s$  at this stage, but as we move communication edges from  $E_s$  to  $E_r$ ,  $G_s$  has fewer and fewer edges. Thus moving edges from  $E_s$  to  $E_r$  can be viewed as removal of edges from  $G_s$ . Whenever we remove edges from  $G_s$  we have to ensure, of course, that the synchronization graph  $G_s$  at that step respects all the synchronization constraints of  $G_{ipc}$ , because we only implement synchronizations represented by the edges  $E_s$  in  $G_s$ . The following theorem is useful to formalize the concept of when the synchronization constraints represented by one synchronization graph  $G_s^1$  imply the synchronization constraints of another graph  $G_s^2$ . This theorem provides a useful constraint for synchronization optimization, and it underlies the validity of the main techniques that we will present in this chapter.

**Theorem 5.1:** The synchronization constraints in a synchronization graph  $G_s^1 = (V, E_{int} \cup E_s^1)$  imply the synchronization constraints of the synchronization graph  $G_s^2 = (V, E_{int} \cup E_s^2)$  if the following condition holds:  $\forall \epsilon$  s.t.  $\epsilon \in E_s^2, \epsilon \notin E_s^1, \rho_{G_s^1}(src(\epsilon), snk(\epsilon)) \leq delay(\epsilon)$ ; that is, if for each edge  $\epsilon$  that is present in  $G_s^2$  but not in  $G_s^1$  there is a minimum delay path from  $src(\epsilon)$  to  $snk(\epsilon)$  in  $G_s^1$  that has total delay of at most  $delay(\epsilon)$ .

(Note that since the vertex sets for the two graphs are identical, it is meaningful to refer to  $src(\epsilon)$  and  $snk(\epsilon)$  as being vertices of  $G_s^1$  even though there are edges  $\epsilon$  s.t.  $\epsilon \in E_s^2, \epsilon \notin E_s^1$ .)

First we prove the following lemma.

**Lemma 5.1:** If there is a path  $p = (e_1, e_2, e_3, \dots, e_n)$  in  $G_s^1$ , then

$$start(snk(e_n), k) \geq end(src(e_1), k - Delay(p)).$$

*Proof of Lemma 5.1:*

The following constraints hold along such a path  $p$  (as per Eqn. 4-1)

$$start(snk(e_1), k) \geq end(src(e_1), k - delay(e_1)). \quad (5-3)$$

Similarly,

$$start(snk(e_2), k) \geq end(src(e_2), k - delay(e_2)).$$

Noting that  $src(e_2)$  is the same as  $snk(e_1)$ , we get

$$start(snk(e_2), k) \geq end(snk(e_1), k - delay(e_2)).$$

Causality implies  $end(v, k) \geq start(v, k)$ , so we get

$$start (snk (e_2), k) \geq start (snk (e_1), k - delay (e_2)) . \quad (5-4)$$

Substituting Eqn. 5-3 in Eqn. 5-4,

$$start (snk (e_2), k) \geq end (src (e_1), k - delay (e_2) - delay (e_1)) .$$

Continuing along  $p$  in this manner, it can easily be verified that

$$start (snk (e_n), k) \geq end (src (e_1), k - delay (e_n) - delay (e_{n-1}) - \dots - delay (e_1))$$

that is,

$$start ((snk (e_n), k) \geq end (src (e_1), k - Delay (p))) . \quad QED.$$

*Proof of Theorem 5.1:* If  $\epsilon \in E_s^2, \epsilon \in E_s^1$ , then the synchronization constraint due to the edge  $\epsilon$  holds in both graphs. But for each  $\epsilon$  s.t.  $\epsilon \in E_s^2, \epsilon \notin E_s^1$  we need to show that the constraint due to  $\epsilon$ :

$$start (snk (\epsilon), k) > end (src (\epsilon), k - delay (\epsilon)) \quad (5-5)$$

holds in  $G_s^1$  provided  $\rho_{G_s^1}(src (\epsilon), snk (\epsilon)) \leq delay (\epsilon)$ , which implies there is

at least one path  $p = (e_1, e_2, e_3, \dots, e_n)$  from  $src (\epsilon)$  to  $snk (\epsilon)$  in  $G_s^1$  ( $src (e_1) = src (\epsilon)$  and  $snk (e_n) = snk (\epsilon)$ ) such that  $Delay (p) \leq delay (\epsilon)$ .

From Lemma 5.1, existence of such a path  $p$  implies

$$start ((snk (e_n), k) \geq end (src (e_1), k - Delay (p))) .$$

that is,

$$start ((snk (\epsilon), k) \geq end (src (\epsilon), k - Delay (p))) . \quad (5-6)$$

If  $Delay (p) \leq delay (\epsilon)$ , then

$end(src(\epsilon), k - Delay(p)) \geq end(src(\epsilon), k - delay(\epsilon))$ . Substituting this in Eqn. 5-6 we get

$$start((snk(\epsilon), k) \geq end(src(\epsilon), k - delay(\epsilon))) .$$

The above relation is identical to Eqn. 5-5, and this proves the Theorem. *QED*.

The above theorem motivates the following definition.

**Definition 5.1:** If  $G_s^1 = (V, E_{int} \cup E_s^1)$  and  $G_s^2 = (V, E_{int} \cup E_s^2)$  are synchronization graphs with the same vertex-set, we say that  $G_s^1$  preserves  $G_s^2$  if  $\forall \epsilon$  s.t.  $\epsilon \in E_2, \epsilon \notin E_1$ , we have  $\rho_{G_s^1}(src(\epsilon), snk(\epsilon)) \leq delay(\epsilon)$ .

Thus, Theorem 5.1 states that the synchronization constraints of  $(V, E_{int} \cup E_s^1)$  imply the synchronization constraints of  $(V, E_{int} \cup E_s^2)$  if  $(V, E_{int} \cup E_s^1)$  preserves  $(V, E_{int} \cup E_s^2)$ .

Given an IPC graph  $G_{ipc}$ , and a synchronization graph  $G_s$  such that  $G_s$  preserves  $G_{ipc}$ , suppose we implement the synchronizations corresponding to the synchronization edges of  $G_s$ . Then, the iteration period of the resulting system is determined by the maximum cycle mean of  $G_s$  ( $MCM(G_s)$ ). This is because the synchronization edges alone determine the interaction between processors; a communication edge without synchronization does not constrain the execution of the corresponding processors in any way.

## 5.5 Formal problem statement

We refer to each access of the shared memory “synchronization variable”  $sv(e)$  by  $src(e)$  and  $snk(e)$  as a synchronization access<sup>1</sup> to shared memory. If synchronization for  $e$  is implemented using UBS, then we see that on average, 4 synchronization accesses are required for  $e$  in each iteration period, while BBS

implies 2 synchronization accesses per iteration period. We define the **synchronization cost** of a synchronization graph  $G_s$  to be the average number of synchronization accesses required per iteration period. Thus, if  $n_{ff}$  denotes the number of synchronization edges in  $G_s$  that are feedforward edges, and  $n_{fb}$  denotes the number of synchronization edges that are feedback edges, then the synchronization cost of  $G_s$  can be expressed as  $(4n_{ff} + 2n_{fb})$ . In the remainder of this paper we develop techniques that apply the results and the analysis framework developed in sections 4.1 and sections 5.2-5.4 to minimize the synchronization cost of a self-timed implementation of an HSDFG without sacrificing the integrity of any inter-processor data transfer or reducing the estimated throughput.

We will explore three mechanisms for reducing synchronization accesses. The first (presented in section 5.6) 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. This technique essentially detects the set of edges that can be moved from the  $E_s$  to the set  $E_r$ . In section 5.7, we 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 us to implement all synchronization edges with BBS. We address optimization criteria in performing such a conversion, and we will show that the extra synchronization accesses required for such a conversion are always (at least) compensated by the number of synchronization accesses that are saved by the more expensive UBSs that are converted to BBSs. Finally, in section 5.9 we outline a mechanism, which we call *resynchronization*, for inserting synchronization edges in a way that

---

1. Note that in our measure of the number of shared memory accesses required for synchronization, we neglect the accesses to shared memory that are performed while the sink actor is waiting for the required data to become available, or the source actor is waiting for an “empty slot” in the buffer. The number of accesses required to perform these “busy-wait” or “spin-lock” operations is dependent on the exact relative execution times of the actor invocations. Since in our problem context this information is not generally available to us, we use the *best case* number of accesses — the number of shared memory accesses required for synchronization assuming that IPC data on an edge is always produced before the corresponding sink invocation attempts to execute — as an approximation.

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

## 5.6 Removing redundant synchronizations

The first technique that we explore for reducing synchronization overhead is removal of *redundant synchronization edges* from the synchronization graph, i.e. finding a minimal set of edges  $E_s$  that need explicit synchronization. Formally, a synchronization edge is **redundant** in a synchronization graph  $G$  if its removal yields a synchronization graph that preserves  $G$ . Equivalently, from definition 5.1, a synchronization edge  $e$  is redundant in the synchronization graph  $G$  if there is a path  $p \neq (e)$  in  $G$  directed from  $src(e)$  to  $snk(e)$  such that  $Delay(p) \leq delay(e)$ .

Thus, the synchronization function associated with a redundant synchronization edge “comes for free” as a by product of other synchronizations. Fig. 5.4 shows an example of a redundant synchronization edge. Here, before executing actor  $D$ , the processor that executes  $\{A, B, C, D\}$  does not need to synchronize with the processor that executes  $\{E, F, G, H\}$  because, due to the synchronization edge  $x_1$ , the corresponding invocation of  $F$  is guaranteed to complete before each invocation of  $D$  is begun. Thus,  $x_2$  is redundant in Fig. 5.4 and can be removed from  $E_s$  into the set  $E_r$ . It is easily verified that the path

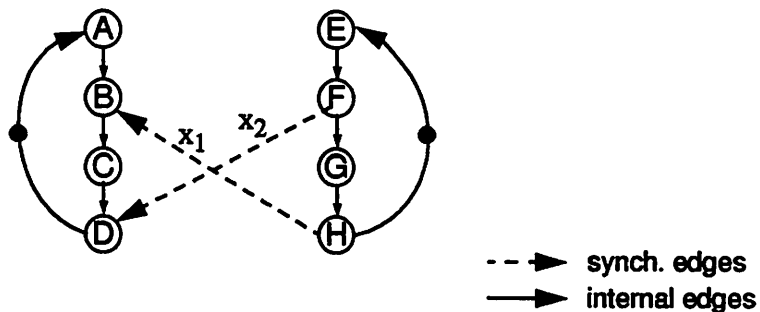


Figure 5.4.  $x_2$  is an example of a redundant synchronization edge.



$p = ((F, G), (G, H), x_1, (B, C), (C, D))$  is directed from  $src(x_2)$  to  $snk(x_2)$ , and has a path delay (zero) that is equal to the delay on  $x_2$ .

In this section we develop an efficient algorithm to optimally remove redundant synchronization edges from a synchronization graph.

### 5.6.1 The independence of redundant synchronizations

The following theorem establishes that the order in which we remove redundant synchronization edges is not important; therefore all the redundant synchronization edges can be removed together.

**Theorem 5.2:** Suppose that  $G_s = (V, E_{int} \cup E_s)$  is a synchronization graph,  $e_1$  and  $e_2$  are distinct redundant synchronization edges in  $G_s$  (i.e. these are edges that could be individually moved to  $E_r$ ), and  $\tilde{G}_s = (V, E_{int} \cup (E - \{e_1\}))$ . Then  $e_2$  is redundant in  $\tilde{G}_s$ . Thus both  $e_1$  and  $e_2$  can be moved into  $E_r$  together.

*Proof:* Since  $e_2$  is redundant in  $G_s$ , there is a path  $p \neq (e_2)$  in  $G_s$  directed from  $src(e_2)$  to  $snk(e_2)$  such that

$$Delay(p) \leq delay(e_2). \quad (5-7)$$

Similarly, there is a path  $p' \neq (e_1)$ , contained in both  $G_s$  and  $\tilde{G}_s$ , that is directed from  $src(e_1)$  to  $snk(e_1)$ , and that satisfies

$$Delay(p') \leq delay(e_1). \quad (5-8)$$

Now, if  $p$  does not contain  $e_1$ , then  $p$  exists in  $\tilde{G}_s$ , and we are done. Otherwise, let  $p' = (x_1, x_2, \dots, x_n)$ ; observe that  $p$  is of the form

$p = (y_1, y_2, \dots, y_{k-1}, e_1, y_k, y_{k+1}, \dots, y_m)$  ; and define

$$p'' \equiv (y_1, y_2, \dots, y_{k-1}, x_1, x_2, \dots, x_n, y_k, y_{k+1}, \dots, y_m) .$$

Clearly,  $p''$  is a path from  $src(e_2)$  to  $snk(e_2)$  in  $\tilde{G}_s$ . Also,

$$\begin{aligned} Delay(p'') &= \sum delay(x_i) + \sum delay(y_i) \\ &= Delay(p') + (Delay(p) - delay(e_1)) \\ &\leq Delay(p) && \text{(from Eqn. 5-8)} \\ &\leq delay(e_2) && \text{(from Eqn. 5-7).} \end{aligned}$$

*QED.*

Theorem 5.2 tells us that we can avoid implementing synchronization for *all* redundant synchronization edges since the “redundancies” are not interdependent. Thus, an optimal removal of redundant synchronizations can be obtained by applying a straightforward algorithm that successively tests the synchronization edges for redundancy in some arbitrary sequence, and since computing the weight of the shortest path in a weighted directed graph is a tractable problem, we can expect such a solution to be practical.

## 5.6.2 Removing redundant synchronizations

Fig. 5.5 presents an efficient algorithm, based on the ideas presented in the previous subsection, for optimal removal of redundant synchronization edges. In this algorithm, we first compute the path delay of a minimum-delay path from  $x$  to  $y$  for each ordered pair of vertices  $(x, y)$  ; here, we assign a path delay of  $\infty$  whenever there is no path from  $x$  to  $y$ . This computation is equivalent to solving an instance of the well known *all points shortest paths problem* [Corm92]. Then, we examine each synchronization edge  $e$  — in some arbitrary sequence — and determine whether or not there is a path from  $src(e)$  to  $snk(e)$  that does not contain  $e$ , and that has a path delay that does not exceed  $delay(e)$ . This check for redundancy is equivalent to the check that is performed by the *if* statement in

*RemoveRedundantSynchs* because if  $p$  is a path from  $src(e)$  to  $snk(e)$  that contains more than one edge and that contains  $e$ , then  $p$  must contain a cycle  $c$  such that  $c$  does not contain  $e$ ; and since all cycles must have positive path delay (from Lemma 4.1), the path delay of such a path  $p$  must exceed  $delay(e)$ . Thus, if  $e_0$  satisfies the inequality in the *if* statement of *RemoveRedundantSynchs*, and  $p^*$  is a path from  $snk(e_0)$  to  $snk(e)$  such that  $Delay(p^*) = \rho(snk(e_0), snk(e))$ , then  $p^*$  cannot contain  $e$ . This observation allows us to avoid having to recompute the shortest paths after removing a candidate redundant edge from  $G_s$ .

From the definition of a redundant synchronization edge, it is easily verified that the removal of a redundant synchronization edge does not alter any of the minimum-delay path values (path delays). That is, given a redundant synchronization edge  $e_r$  in  $G_s$ , and two arbitrary vertices  $x, y \in V$ , if we let  $\hat{G}_s = (V, E_{int} \cup (E_s - \{e_r\}))$ , then  $\rho_{\hat{G}_s}(x, y) = \rho_{G_s}(x, y)$ . Thus, none of the

### Function RemoveRedundantSynchs

**Input:** A synchronization graph  $G_s = E_{int} \cup E_s$

**Output:** The synchronization graph  $G_s^* = (V, E_{int} \cup (E_s - E_r))$

1. Compute  $\rho_{G_s}(x, y)$  for each ordered pair of vertices in  $G_s$ .
2.  $E_r \leftarrow \emptyset$
3. **For each**  $e \in E_s$ 
  - For each** output edge  $e_o$  of  $src(e)$  **except for**  $e$ 
    - If**  $delay(e_o) + \rho_{G_s}(snk(e_o), snk(e)) \leq delay(e)$
    - Then**
      - $E_r \leftarrow E_r \cup \{e\}$
      - Break** /\* exit the innermost enclosing **For** loop \*/
    - Endif**
  - Endfor**
4. **Return**  $(V, E_{int} \cup (E_s - E_r))$ .

Figure 5.5. An algorithm that optimally removes redundant synchronization edges.

minimum-delay path values computed in Step 1 need to be recalculated after removing a redundant synchronization edge in Step 3.

Observe that the complexity of the function *RemoveRedundantSynchs* is dominated by Step 1 and Step 3. Since all edge delays are non-negative, we can repeatedly apply Dijkstra's single-source shortest path algorithm (once for each vertex) to carry out Step 1 in  $O(|V|^3)$  time; a modification of Dijkstra's algorithm can be used to reduce the complexity of Step 1 to  $O(|V|^2 \log_2(|V|) + |V||E|)$  [Corm92]. In Step 3,  $|E|$  is an upper bound for the number of synchronization edges, and in the worst case, each vertex has an edge connecting it to every other member of  $V$ . Thus, the time complexity of Step 3 is  $O(|V||E|)$ , and if we use the modification to Dijkstra's algorithm mentioned above for Step 1, then the time complexity of *RemoveRedundantSynchs* is

$$O(|V|^2 \log_2(|V|) + |V||E| + |V||E|) = O(|V|^2 \log_2(|V|) + |V||E|).$$

### 5.6.3 Comparison with Shaffer's approach

In [Sha89], Shaffer presents an algorithm that minimizes the number of directed synchronizations in the self-timed execution of an HSDFG under the (implicit) assumption that the execution of successive iterations of the HSDFG are not allowed to overlap. In Shaffer's technique, a construction identical to our synchronization graph is used except that there is no feedback edge connecting the last actor executed on a processor to the first actor executed on the same processor, and edges that have delay are ignored since only intra-iteration dependencies are significant. Thus, Shaffer's synchronization graph is acyclic. *RemoveRedundantSynchs* can be viewed as an extension of Shaffer's algorithm to handle self-timed, iterative execution of an HSDFG; Shaffer's algorithm accounts for self-timed execution only within a graph iteration, and in general, it can be applied to iterative dataflow programs only if all processors are forced to synchronize between graph iterations.

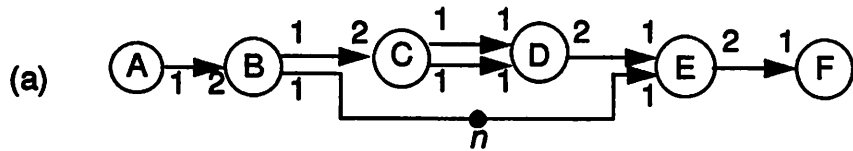
### 5.6.4 An example

In this subsection, we illustrate the benefits of removing redundant synchronizations through a practical example. Fig. 5.6(a) shows an abstraction of a three channel, multi-resolution quadrature mirror (QMF) filter bank, which has applications in signal compression [Vai93]. This representation is based on the general (not homogeneous) SDF model, and accordingly, each edge is annotated with the number of tokens produced and consumed by its source and sink actors. Actors  $A$  and  $F$  represent the subsystems that, respectively, supply and consume data to/from the filter bank system;  $B$  and  $C$  each represents a parallel combination of decimating high and low pass FIR analysis filters;  $D$  and  $E$  represent the corresponding pairs of interpolating synthesis filters. The amount of delay on the edge directed from  $B$  to  $E$  is equal to the sum of the filter orders of  $C$  and  $D$ . For more details on the application represented by Fig. 5.6(a), we refer the reader to [Vai93].

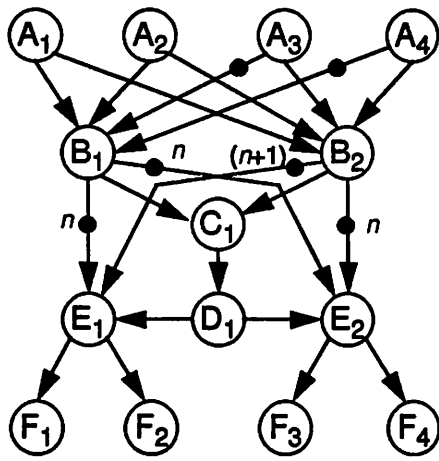
To construct a periodic, parallel schedule we must first determine the number of times  $q(N)$  that each actor  $N$  must be invoked in the periodic schedule. Systematic techniques to compute these values are presented in [Lee87]. Next, we must determine the precedence relationships between the actor invocations. In determining the exact precedence relationships, we must take into account the dependence of a given filter invocation on not only the invocation that produces the token that is “consumed” by the filter, but also on the invocations that produce the  $n$  preceding tokens, where  $n$  is the order of the filter. Such dependence can easily be evaluated with an additional dataflow parameter on each actor input that specifies the number of *past tokens* that are accessed [Prin91]<sup>1</sup>. Using this infor-

---

1. It should be noted that some SDF-based design environments choose to forego parallelization across multiple invocations of an actor in favor of simplified code generation and scheduling. For example, in the GRAPE system, this restriction has been justified on the grounds that it simplifies inter-processor data management, reduces code duplication, and allows the derivation of efficient scheduling algorithms that operate directly on general SDF graphs without requiring the use of the acyclic precedence graph (APG) [Bil94].



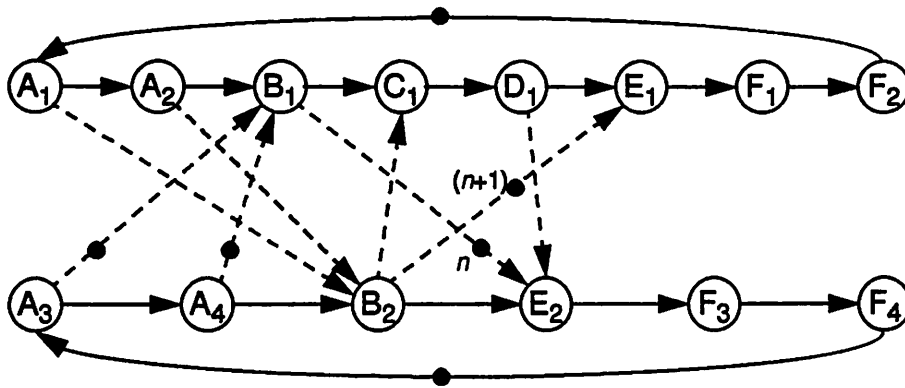
(b)



Proc. 1  $A_1, A_2, B_1, C_1, D_1, E_1, F_1, F_2$

Proc. 2  $A_3, A_4, B_2, E_2, F_3, F_4$

(c)



(d)

--> synth. edges  
 —> internal edges

Figure 5.6. (a) A multi-resolution QMF filter bank used to illustrate the benefits of removing redundant synchronizations. (b) The precedence graph for (a). (c) A self-timed, two-processor, parallel schedule for (a). (d) The initial synchronization graph for (c).

mation, together with the invocation counts specified by  $q$ , we obtain the precedence relationships specified by the graph of Fig. 5.6(b), in which the  $i$ th invocation of actor  $N$  is labeled  $N_i$ , and each edge  $e$  specifies that invocation  $snk(e)$  requires data produced by invocation  $src(e)$   $delay(e)$  iteration periods after the iteration period in which the data is produced.

A self-timed schedule for Fig. 5.6(b) that can be obtained from Hu’s list scheduling method [Hu61] (described in is specified in Chapter 1 section 1.2) is specified in Fig. 5.6(c), and the synchronization graph that corresponds to the IPC graph of Fig. 5.6(b) and Fig. 5.6(c) is shown in Fig. 5.6(d). All of the dashed edges in Fig. 5.6(d) are synchronization edges. If we apply Shaffer’s method, which considers only those synchronization edges that do not have delay, we can eliminate the need for explicit synchronization along only one of the 8 synchronization edges — edge  $(A_1, B_2)$ . In contrast, if we apply *RemoveRedundantSynchs*, we can detect the redundancy of  $(A_1, B_2)$  as well as four additional redundant synchronization edges —  $(A_3, B_1)$ ,  $(A_4, B_1)$ ,  $(B_2, E_1)$ , and  $(B_1, E_2)$ . Thus, *RemoveRedundantSynchs* reduces the number of synchronizations from 8 down to 3 — a reduction of 62%. Fig. 5.7 shows the synchronization graph of Fig. 5.6(d) after all redundant synchronization edges are removed. It is easily verified that the synchronization edges that remain in this graph are not redundant; explicit synchronizations need only be implemented for these edges.

## 5.7 Making the synchronization graph strongly connected

In section 5.4.1, we defined two different synchronization protocols — bounded buffer synchronization (BBS), which has a cost of 2 synchronization accesses per iteration period, and can be used whenever the associated edge is contained in a strongly connected component of the synchronization graph; and unbounded buffer synchronization (UBS), which has a cost of 4 synchronization accesses per iteration period. We pay the additional overhead of UBS whenever

the associated edge is a feedforward edge of the synchronization graph.

One alternative to implementing UBS for a feedforward edge  $e$  is to add synchronization edges to the synchronization graph so that  $e$  becomes encapsulated in a strongly connected component; such a transformation would allow  $e$  to be implemented with BBS. However, extra synchronization accesses will be required to implement the new synchronization edges that are inserted. In this section, we show that by adding synchronization edges through a certain simple procedure, the synchronization graph can be transformed into a strongly connected graph in a way that the overhead of implementing the extra synchronization edges is always compensated by the savings attained by being able to avoid the use of UBS. That is, our transformations ensure that the total number of synchronization accesses required (per iteration period) for the transformed graph is less than or equal to the number of synchronization accesses required for the original synchronization graph. Through a practical example, we show that this transformation can significantly reduce the number of required synchronization accesses. Also, we discuss a technique to compute the delay that should be added to each of the new edges added in the conversion to a strongly connected graph. This technique com-

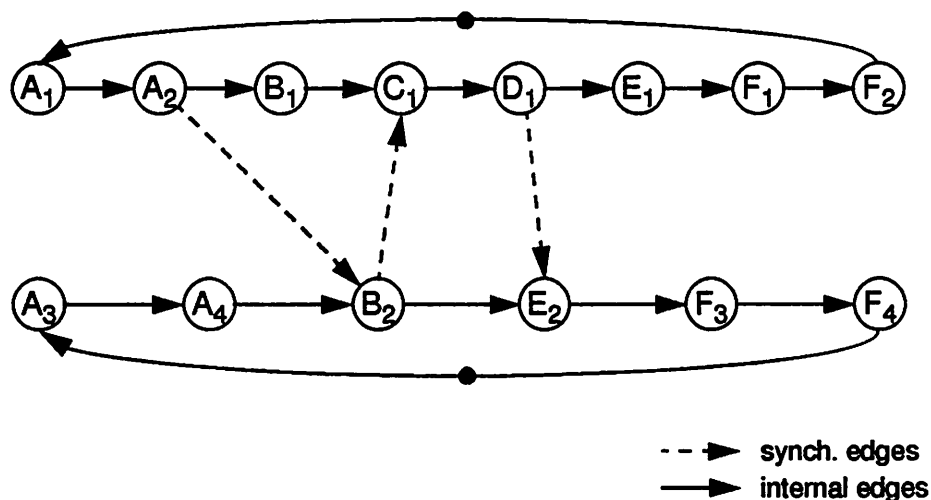


Figure 5.7. The synchronization graph of Fig. 5.6(d) after all redundant synchronization edges are removed.



puts the delays in a way that the estimated throughput of the IPC graph is preserved with minimal increase in the shared memory storage cost required to implement the communication edges.

### 5.7.1 Adding edges to the synchronization graph

Fig. 5.8 presents our algorithm for transforming a synchronization graph that is not strongly connected into a strongly connected graph. This algorithm simply “chains together” the source SCCs, and similarly, chains together the sink SCCs. The construction is completed by connecting the first SCC of the “source chain” to the last SCC of the sink chain with an edge that we call the **sink-source edge**. From each source or sink SCC, the algorithm selects a vertex that has mini-

**Function** *Convert-to-SC-graph*

**Input:** A synchronization graph  $G$  that is not strongly connected.

**Output:** A strongly connected graph obtained by adding edges between the SCCs of  $G$ .

1. Generate an ordering  $C_1, C_2, \dots, C_m$  of the source SCCs of  $G$ , and similarly, generate an ordering  $D_1, D_2, \dots, D_n$  of the sink SCCs of  $G$ .
2. Select a vertex  $v_1 \in C_1$  that minimizes  $t(*)$  over  $C_1$ .
3. **For**  $i = 2, 3, \dots, m$ 
  - Select a vertex  $v_i \in C_i$  that minimizes  $t(*)$  over  $C_i$ .
  - Instantiate the edge  $d_0(v_{i-1}, v_i)$ .
- End For**
4. Select a vertex  $w_1 \in D_1$  that minimizes  $t(*)$  over  $D_1$ .
5. **For**  $i = 2, 3, \dots, n$ 
  - Select a vertex  $w_i \in D_i$  that minimizes  $t(*)$  over  $D_i$ .
  - Instantiate the edge  $d_0(w_{i-1}, w_i)$ .
- End For**
6. Instantiate the edge  $d_0(w_m, v_1)$ .

Figure 5.8. An algorithm for converting a synchronization graph that is not strongly connected into a strongly connected graph.

imum execution time to be the chain “link” corresponding to that SCC. Minimum execution time vertices are chosen in an attempt to minimize the amount of delay that must be inserted on the new edges to preserve the estimated throughput of the original graph. In section 5.7.2, we discuss the selection of delays for the edges introduced by *Convert-to-SC-graph*.

It is easily verified that algorithm *Convert-to-SC-graph* always produces a strongly connected graph, and that a conversion to a strongly connected graph cannot be attained by adding fewer edges than the number of edges added by *Convert-to-SC-graph*. Fig. 5.9 illustrates a possible solution obtained by algorithm *Convert-to-SC-graph*. Here, the black dashed edges are the synchronization edges contained in the original synchronization graph, and the grey dashed edges are the edges that are added by *Convert-to-SC-graph*. The dashed edge labeled  $e_s$  is the sink-source edge.

Assuming the synchronization graph is connected, the number of feedforward edges  $n_f$  must satisfy  $n_f \geq (n_c - 1)$ , where  $n_c$  is the number of SCCs. This follows from the fundamental graph theoretic fact that in a connected graph  $(V^*, E^*)$ ,  $|E^*|$  must be at least  $(|V^*| - 1)$ . Now, it is easily verified that the

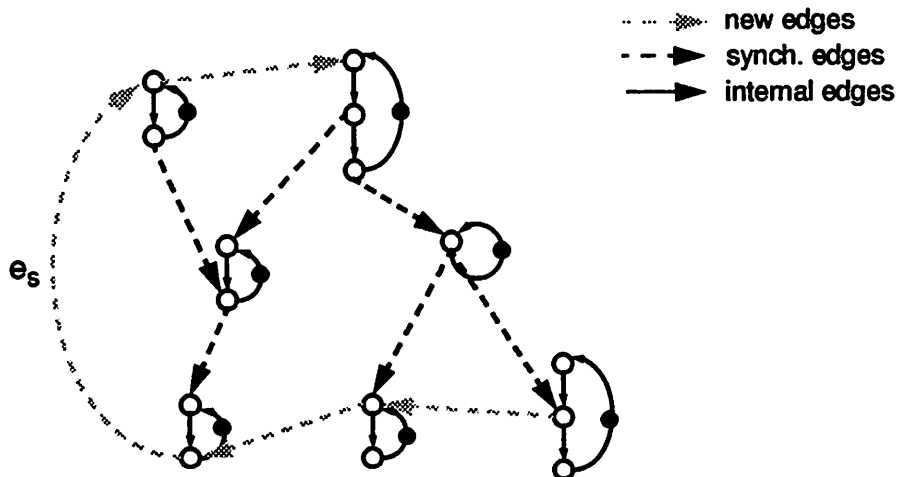


Figure 5.9. An illustration of a possible solution obtained by algorithm *Convert-to-SC-graph*.

number of new edges introduced by *Convert-to-SC-graph* is equal to  $(n_{src} + n_{snk} - 1)$ , where  $n_{src}$  is the number of source SCCs, and  $n_{snk}$  is the number of sink SCCs. Thus, the number of synchronization accesses per iteration period,  $S_+$ , that is required to implement the edges introduced by *Convert-to-SC-graph* is  $(2 \times (n_{src} + n_{snk} - 1))$ , while the number of synchronization accesses,  $S_-$ , eliminated by *Convert-to-SC-graph* (by allowing the feedforward edges of the original synchronization graph to be implemented with BBS rather than UBS) equals  $2n_f$ . It follows that the net change  $(S_+ - S_-)$  in the number of synchronization accesses satisfies

$$(S_+ - S_-) = 2(n_{src} + n_{snk} - 1) - 2n_f = 2(n_c - 1 - n_f) \leq 2(n_c - 1 - (n_c - 1)),$$

and thus,  $(S_+ - S_-) \leq 0$ . We have established the following result.

**Theorem 5.3:** Suppose that  $G$  is a synchronization graph, and  $\hat{G}$  is the graph that results from applying algorithm *Convert-to-SC-graph* to  $G$ . Then the synchronization cost of  $\hat{G}$  is less than or equal to the synchronization cost of  $G$ .

For example, without the edges added by *Convert-to-SC-graph* (the dashed grey edges) in Fig. 5.9, there are 6 feedforward edges, which require 24 synchronization accesses per iteration period to implement. The addition of the 4 dashed edges requires 8 synchronization accesses to implement these new edges, but allows us to use UBS for the original feedforward edges, which leads to a savings of 12 synchronization accesses for the original feedforward edges. Thus, the net effect achieved by *Convert-to-SC-graph* in this example is a reduction of the total number of synchronization accesses by  $(12 - 8) = 4$ . As another example, consider Fig. 5.10, which shows the synchronization graph topology (after redundant synchronization edges are removed) that results from a four-processor schedule of a synthesizer for plucked-string musical instruments in seven voices based on the Karplus-Strong technique. This algorithm was also discussed in Chapter 3, as an

example application that was implemented on the ordered memory access architecture prototype. This graph contains  $n_i = 6$  synchronization edges (the dashed edges), all of which are feedforward edges, so the synchronization cost is  $4n_i = 24$  synchronization access per iteration period. Since the graph has one source SCC and one sink SCC, only one edge is added by *Convert-to-SC-graph*, and adding this edge reduces the synchronization cost to  $2n_i + 2 = 14$  — a 42% savings. Fig. 5.11 shows the topology of a possible solution computed by *Convert-to-SC-graph* on this example. Here, the dashed edges represent the synchronization edges in the synchronization graph returned by *Convert-to-SC-graph*.

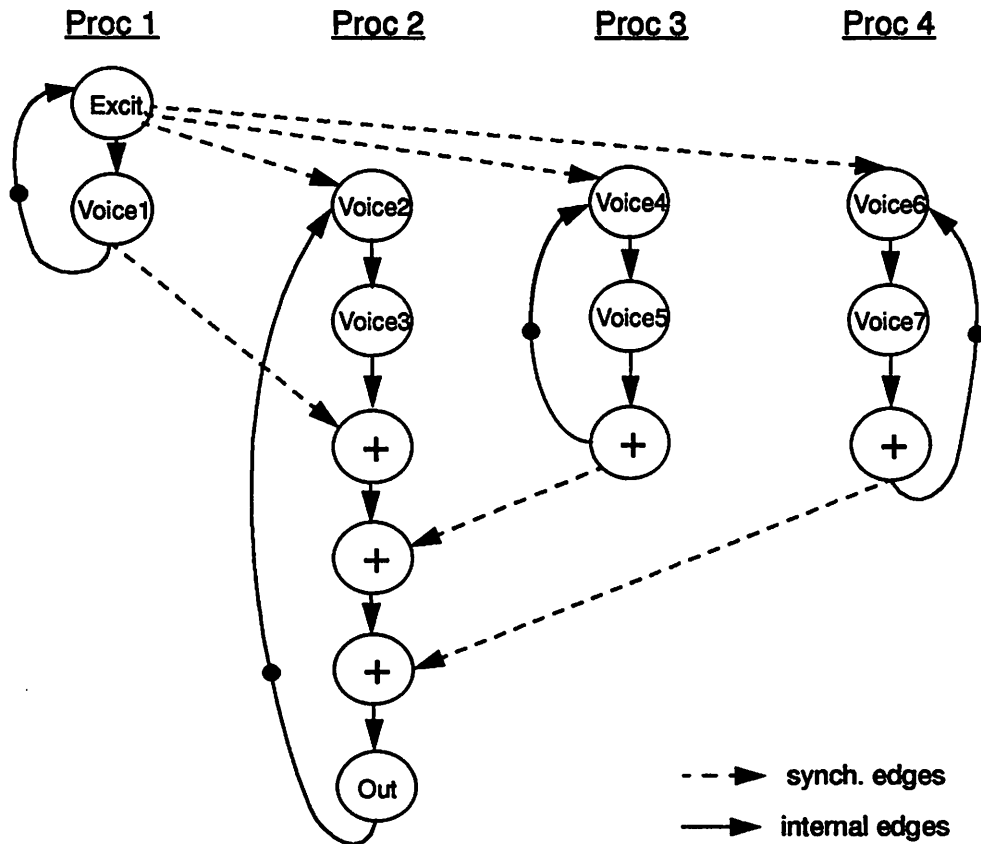


Figure 5.10. The synchronization graph, after redundant synchronization edges are removed, induced by a four-processor schedule of a music synthesizer based on the Karplus-Strong algorithm.

## 5.7.2 Insertion of delays

One issue remains to be addressed in the conversion of a synchronization graph  $G_s$  into a strongly connected graph  $\hat{G}_s$  — the proper insertion of delays so that  $\hat{G}_s$  is not deadlocked, and does not have lower estimated throughput than  $G_s$ . The potential for deadlock and reduced estimated throughput arise because the conversion to a strongly connected graph must necessarily introduce one or more new fundamental cycles. In general, a new cycle may be delay-free, or its cycle mean may exceed that of the critical cycle in  $G_s$ . Thus, we may have to insert delays on the edges added by *Convert-to-SC-graph*. The location (edge) and magnitude of the delays that we add are significant since they effect the self-timed buffer bounds of the communication edges, as shown subsequently in Theorem 5.4. Since the self-timed buffer bounds determine the amount of memory that we allocate for the corresponding buffers, it is desirable to prevent deadlock and decrease in estimated throughput in a way that the sum of the self-timed buffer bounds over all communication edges is minimized. In this section, we outline a simple and efficient algorithm for addressing this problem. Our algorithm pro-

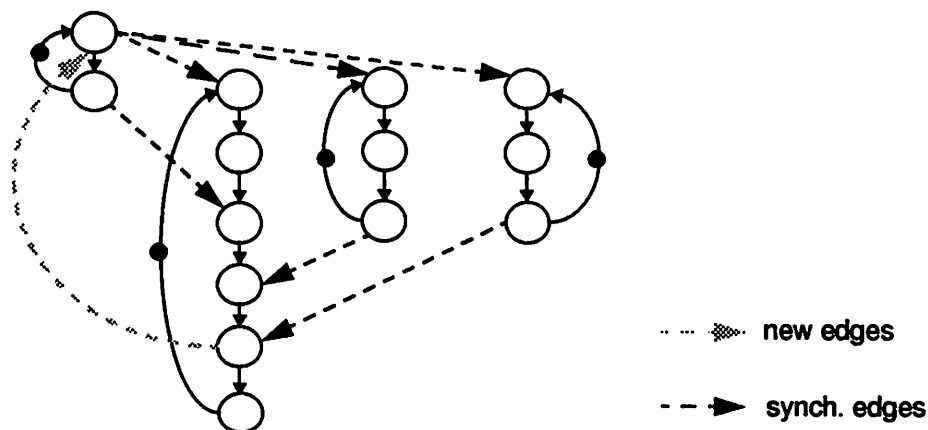


Figure 5.11. A possible solution obtained by applying *Convert-to-SC-graph* to the example of Figure 5.10.

duces an optimal result if  $G_s$  has only one source SCC or only one sink SCC; in other cases, the algorithm must be viewed as a heuristic.

Fig. 5.12 outlines the restricted version of our algorithm that applies when the synchronization graph  $G_s$  has exactly one source SCC. Here, *BellmanFord* is assumed to be an algorithm that takes a synchronization graph  $Z$  as input, and repeatedly applies the Bellman-Ford algorithm discussed in pp. 94-97 of [Law76] to return the cycle mean of the critical cycle in  $Z$ ; if one or more cycles exist that have zero path delay, then *BellmanFord* returns  $\infty$ . Details of this procedure can be found in [Bhat95a].

Fig. 5.13 illustrates a solution obtained from *DetermineDelays*. Here we assume that  $t(v) = 1$  for each vertex  $v$ , and we assume that the set of communication edges are  $e_a$  and  $e_b$ . The grey dashed edges are the edges added by *Convert-to-SC-graph*. We see that *MCM* is determined by the cycle in the sink SCC of the original graph, and inspection of this cycle yields  $MCM = 4$ . The solution determined by *DetermineDelays* for Fig. 5.13 is one delay on  $e_a$  and one delay on  $e_b$  ( $\delta_0, \delta_1 = 1$ ); the resulting self-timed buffer bounds of  $e_a$  and  $e_b$  are, respectively, 1 and 2; the total buffer sizes for the communication edges is thus 3 (sum of the self-timed buffer bounds).

*DetermineDelays* can be extended to yield heuristics for the general case in which the original synchronization graph  $G_s$  contains more than one source SCC

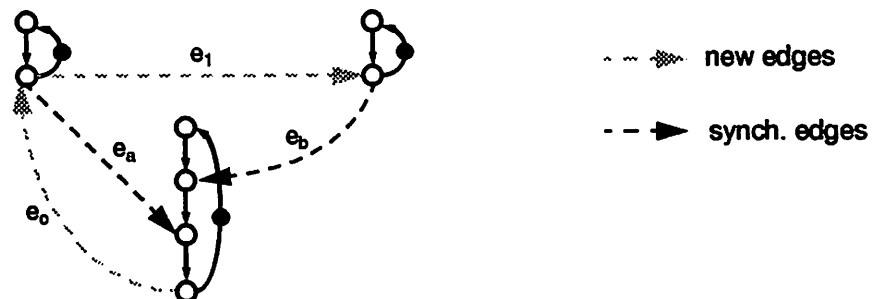


Figure 5.13. An example used to illustrate a solution obtained by algorithm *DetermineDelays*.

**Function *DetermineDelays***

**Input:** Synchronization graphs  $G_s = (V, E)$  and  $\hat{G}_s$ , where  $\hat{G}_s$  is the graph computed by *Convert-to-SC-graph* when applied to  $G_s$ . The ordering of source SCCs generated in Step 2 of *Convert-to-SC-graph* is denoted  $C_1, C_2, \dots, C_m$ . For  $i = 1, 2, \dots, m-1$ ,  $e_i$  denotes the edge instantiated by *Convert-to-SC-graph* from a vertex in  $C_i$  to a vertex in  $C_{i+1}$ . The sink-source edge instantiated by *Convert-to-SC-graph* is denoted  $e_0$ .

**Output:** Non-negative integers  $d_0, d_1, \dots, d_{m-1}$  such that the estimated throughput when  $\text{delay}(e_i) = d_i$ ,  $0 \leq i \leq m-1$ , equals estimated throughput of  $G_s$ .

```

 $X_0 = \hat{G}_s [e_0 \rightarrow \infty, \dots, e_{m-1} \rightarrow \infty]$  /* set delays on each edge to be infinite */
 $\lambda_{max} = \text{BellmanFord}(X_0)$  /* compute the max. cycle mean of  $G_s$  */
 $d_{ub} = \left\lceil \left( \sum_{x \in V} t(x) \right) / MCM \right\rceil$  /* an upper bound on the delay required for any
 $e_i$  */
For  $i = 0, 1, \dots, m-1$ 
     $\delta_i = \text{MinDelay}(X_i, e_i, MCM, d_{ub})$ 
     $X_{i+1} = X_i [e_i \rightarrow \delta_i]$  /* fix the delay on  $e_i$  to be  $\delta_i$  */
End For
Return  $\delta_0, \delta_1, \dots, \delta_{m-1}$ .

```

**Function *MinDelay*( $X, e, \lambda, B$ )**

**Input:** A synchronization graph  $X$ , an edge  $e$  in  $X$ , a positive real number  $\lambda$ , and a positive integer  $B$ .

**Output:** Assuming  $X [e \rightarrow B]$  has estimated throughput no less than  $\lambda^{-1}$ , determine the minimum  $d \in \{0, 1, \dots, B\}$  such that the estimated throughput of  $X [e \rightarrow d]$  is no less than  $\lambda^{-1}$ .

Perform a binary search in the range  $[0, 1, \dots, B]$  to find the minimum value of  $r \in \{0, 1, \dots, B\}$  such that  $\text{BellmanFord}(X [e \rightarrow r])$  returns a value less than or equal to  $\lambda$ . Return this minimum value of  $r$ .

Figure 5.12. An algorithm for determining the delays on the edges introduced by algorithm *Convert-to-SC-graph*.

and more than one sink SCC. For example, if  $(a_1, a_2, \dots, a_k)$  denote edges that were instantiated by *Convert-to-SC-graph* “between” the source SCCs — with each  $a_i$  representing the  $i$ th edge created — and similarly,  $(b_1, b_2, \dots, b_l)$  denote the sequence of edges instantiated between the sink SCCs, then algorithm *DetermineDelays* can be applied with the modification that  $m = k + l + 1$ , and  $(e_0, e_1, \dots, e_{m-1}) \equiv (e_s, a_1, a_2, \dots, a_k, b_l, b_{l-1}, \dots, b_1)$ , where  $e_s$  is the sink-source edge from *Convert-to-SC-graph*. Further details related to these issues can be found in [Bhat95a].

*DetermineDelays* and its variations have complexity  $O(|V|^4 (\log_2(|V|))^2)$  [Bhat95a]. It is also easily verified that the time complexity of *DetermineDelays* dominates that of *Convert-to-SC-graph*, so the time complexity of applying *Convert-to-SC-graph* and *DetermineDelays* in succession is again  $O(|V|^4 (\log_2(|V|))^2)$ .

Although the issue of deadlock does not explicitly arise in algorithm *DetermineDelays*, the algorithm does guarantee that the output graph is not deadlocked, assuming that the input graph is not deadlocked. This is because (from Lemma 4.1) deadlock is equivalent to the existence of a cycle that has zero path delay, and is thus equivalent to an infinite maximum cycle mean. Since *DetermineDelays* does not increase the maximum cycle mean, it follows that the algorithm cannot convert a graph that is not deadlocked into a deadlocked graph.

Converting a mixed grain HSDFG that contains feedforward edges into a strongly connected graph has been studied by Zivojnovic [Zivo94b] in the context of retiming when the assignment of actors to processors is fixed beforehand. In this case, the objective is to retime the input graph so that the number of communication edges that have nonzero delay is maximized, and the conversion is performed to constrain the set of possible retimings in such a way that an integer linear programming formulation can be developed. The technique generates two dummy vertices that are connected by an edge; the sink vertices of the original graph are connected to one of the dummy vertices, while the other dummy vertex is connected to each source. It is easily verified that in a self-timed execution, this



scheme requires at least four more synchronization accesses per graph iteration than the method that we have proposed. We can obtain further relative savings if we succeed in detecting one or more beneficial resynchronization opportunities. The effect of Zivojnovic's retiming algorithm on synchronization overhead is unpredictable since one hand a communication edge becomes "easier to make redundant" when its delay increases, while on the other hand, the edge becomes less useful in making other communication edges redundant since the path delay of all paths that contain the edge increase.

## 5.8 Computing buffer bounds from $G_s$ and $G_{ipc}$

After all the optimizations are complete we have a final synchronization graph  $G_s = (V, E_{int} \cup E_s)$  that preserves  $G_{ipc}$ . Since the synchronization edges in  $G_s$  are the ones that are finally implemented, it is advantageous to calculate the self-timed buffer bound  $B_{fb}$  as a final step after all the transformations on  $G_s$  are complete, instead of using  $G_{ipc}$  itself to calculate these bounds. This is because addition of the edges in the *Convert-to-SC-graph* and *Resynchronize* steps may reduce these buffer bounds. It is easily verified that removal of edges cannot change the buffer bounds in Eqn. 5-1 as long as the synchronizations in  $G_{ipc}$  are preserved. Thus, in the interest of obtaining minimum possible shared buffer sizes, we compute the bounds using the optimized synchronization graph. The following theorem tells us how to compute the self-timed buffer bounds from  $G_s$ .

**Theorem 5.4:** If  $G_s$  preserves  $G_{ipc}$  and the synchronization edges in  $G_s$  are implemented, then for each feedback communication edge  $e$  in  $G_{ipc}$ , the self-timed buffer bound of  $e$  ( $B_{fb}(e)$ ) — an upper bound on the number of data tokens that can be present on  $e$  — is given by:

$$B_{fb}(e) = \rho_{G_s}(snk(e), src(e)) + delay(e) ;$$

*Proof:* By Lemma 5.1, if there is a path  $p$  from  $snk(e)$  to  $src(e)$  in  $G_s$ , then

$$start(src(e), k) \geq end(snk(e), k - Delay(p)) .$$

Taking  $p$  to be an arbitrary minimum-delay path from  $snk(e)$  to  $src(e)$  in  $G_s$ , we get

$$start(src(e), k) \geq end(snk(e), k - \rho_{G_s}(snk(e), src(e))) .$$

That is,  $src(e)$  cannot be more than  $\rho_{G_s}(snk(e), src(e))$  iterations “ahead” of  $snk(e)$ . Thus there can never be more than  $\rho_{G_s}(snk(e), src(e))$  tokens more than the initial number of tokens on  $e$  —  $delay(e)$ . Since the initial number of tokens on  $e$  was  $delay(e)$ , the size of the buffer corresponding to  $e$  is bounded above by  $B_{fb}(e) = \rho_{G_s}(snk(e), src(e)) + delay(e)$ . *QED.*

The quantities  $\rho_{G_s}(snk(e), src(e))$  can be computed using Dijkstra’s algorithm [Corm92] to solve the all-pairs shortest path problem on the synchronization graph in time  $O(|V|^3)$ .

## 5.9 Resynchronization

It is sometimes possible to reduce the total number of synchronization edges  $E_s$  by adding new synchronization edges to a synchronization graph. We refer to the process of adding one or more new synchronization edges and removing the redundant edges that result as *resynchronization*; Fig. 5.14(a) illustrates this concept, where the dashed edges represent synchronization edges. Observe that if we insert the new synchronization edge  $d_0(C, H)$ , then two of the original synchronization edges —  $(B, G)$  and  $(E, J)$  — become redundant, and the net effect is that we require one less synchronization edge to be implemented. In Fig. 5.14(b), we show the synchronization graph that results from inserting the *resynchronization edge*  $d_0(C, H)$  (grey edge) into Fig. 5.14(a), and then removing the redundant synchronization edges that result.

We refer to the problem of finding a resynchronization with the fewest number of final synchronization edges as the **resynchronization problem**. In

[Bhat95a] we formally establish that the resynchronization problem is NP-hard by deriving a polynomial time reduction from the classic *minimal set covering problem*, which is known to be NP-hard [Garey79], to the pair-wise resynchronization problem. The complexity remains the same whether we consider a general resynchronization problem that also attempts to insert edges within SCCs, or a restricted version that only adds feed-forward edges between SCCs (the *Resynchronize* procedure in [Bhat95a] restricts itself to the latter, because in this case it is simpler to ensure that the estimated throughput is unaffected by the added edges).

Although the correspondence that we establish between the resynchronization problem and set covering shows that the resynchronization problem probably cannot be attacked optimally with a polynomial-time algorithm, the correspondence allows any heuristic for set covering to be adapted easily into a heuristic for the pair-wise resynchronization problem, and applying such a heuristic to each pair of SCCs in a general synchronization graph yields a heuristic for the general (not just pair-wise) resynchronization problem [Bhat95a]. This is fortunate since the set covering problem has been studied in great depth, and efficient heuristic methods

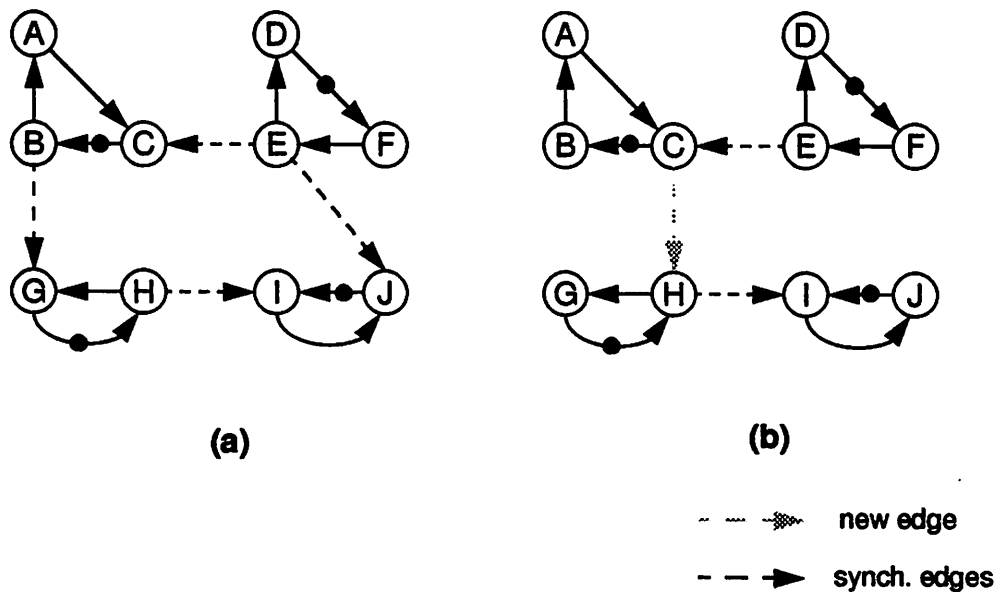


Figure 5.14. An example of resynchronization.

have been devised for it [Corm92].

For a certain class of IPC graphs (formally defined in [Bhat95b]) a provably optimum resynchronization can be obtained, using a procedure similar to pipelining. This procedure, however, leads to an implementation that in general has a larger latency than the implementation we start out with. The resynchronization procedure as outlined in [Bhat95a] in general can lead to implementations with increased latency. Latency is measured as the time delay between when an input data sample is available and when the corresponding output is generated. In [Bhat95b] we show how we can modify the resynchronization procedure to trade off synchronization cost with latency. An optimal latency constrained synchronization, however, is again shown to be NP-hard.

The work on resynchronization is very much ongoing research, a brief outline of which we have presented in this section.

## 5.10 Summary

We have addressed the problem of minimizing synchronization overhead in self-timed multiprocessor implementations. The metric we use to measure synchronization cost is the number of accesses made to shared memory for the purpose of synchronization, per schedule period. We used the IPC graph framework introduced in the previous chapter to extend an existing technique — detection of redundant synchronization edges — for noniterative programs to the iterative case. We presented a method for the conversion of the synchronization graph into a strongly connected graph, which again results in reduced synchronization overhead. Also, we briefly outlined the resynchronization procedure, which involves adding synchronization points in the schedule such that the overall synchronization costs are reduced. Details of resynchronization can be found in [Bhat95a] and [Bhat95b]. We demonstrated the relevance of our techniques through practical examples.

The input to our algorithm is an HSDFG and a parallel schedule for it. The

output is an IPC graph  $G_{ipc} = (V, E_{ipc})$ , which represents buffers as communication edges; a strongly connected synchronization graph  $G_s = (V, E_{int} \cup E_s)$ , which represents synchronization constraints; and a set of shared-memory buffer sizes  $\{B_{fb}(e) \mid e \text{ is an IPC edge in } G_{ipc}\}$ . Fig. 5.15 specifies the complete algorithm.

A code generator can then accept  $G_{ipc}$  and  $G_s$ , allocate a buffer in shared memory for each communication edge  $e$  specified by  $G_{ipc}$  of size  $B_{fb}(e)$ , and generate synchronization code for the synchronization edges represented in  $G_s$ . These synchronizations may be implemented using the BBS protocol. The resulting synchronization cost is  $2n_s$ , where  $n_s$  is the number of synchronization edges in the synchronization graph  $G_s$  that is obtained after all optimizations are com-

**Function** *MinimizeSynchCost*

**Input:** An HSDFG  $G$  and a self-timed schedule  $S$  for this HSDFG.

**Output:**  $G_{ipc}$ ,  $G_s$ , and  $\{B_{fb}(e) \mid e \text{ is an IPC edge in } G_{ipc}\}$ .

1. Extract  $G_{ipc}$  from  $G$  and  $S$
2.  $G_s \leftarrow G_{ipc}$  /\* Each communication edge is also a synchronization edge to begin with \*/
3.  $G_s \leftarrow Resynchronize(G_s)$
4.  $G_s \leftarrow Convert-to-SC-graph(G_s)$
5.  $G_s \leftarrow DetermineDelays(G_s)$
6.  $G_s \leftarrow RemoveRedundantSynchs(G_s)$
7. Calculate the buffer size  $B_{fb}(e)$  for each communication edge  $e$  in  $G_{ipc}$ .
  - a) Compute  $\rho_{G_s}(src(e), snk(e))$
  - b)  $B_{fb}(e) \leftarrow \rho_{G_s}(src(e), snk(e)) + delay(e)$

Figure 5.15. The complete synchronization optimization algorithm.

pleted.

... (3.1) = ...

... (3.2) = ...

... (3.3) = ...

... (3.4) = ...

... (3.5) = ...

... (3.6) = ...

... (3.7) = ...

... (3.8) = ...

... (3.9) = ...

... (3.10) = ...

... (3.11) = ...

... (3.12) = ...

... (3.13) = ...

... (3.14) = ...

... (3.15) = ...

... (3.16) = ...

... (3.17) = ...

... (3.18) = ...

... (3.19) = ...

... (3.20) = ...

... (3.21) = ...

... (3.22) = ...

... (3.23) = ...

... (3.24) = ...

... (3.25) = ...

... (3.26) = ...

... (3.27) = ...

... (3.28) = ...

... (3.29) = ...

... (3.30) = ...

... (3.31) = ...

... (3.32) = ...

... (3.33) = ...

... (3.34) = ...

... (3.35) = ...

... (3.36) = ...

... (3.37) = ...

... (3.38) = ...

... (3.39) = ...

... (3.40) = ...

... (3.41) = ...

... (3.42) = ...

... (3.43) = ...

... (3.44) = ...

... (3.45) = ...

... (3.46) = ...

... (3.47) = ...

... (3.48) = ...

... (3.49) = ...

... (3.50) = ...

... (3.51) = ...

... (3.52) = ...

... (3.53) = ...

... (3.54) = ...

... (3.55) = ...

... (3.56) = ...

... (3.57) = ...

... (3.58) = ...

... (3.59) = ...

... (3.60) = ...

... (3.61) = ...

... (3.62) = ...

... (3.63) = ...

... (3.64) = ...

... (3.65) = ...

... (3.66) = ...

... (3.67) = ...

... (3.68) = ...

... (3.69) = ...

... (3.70) = ...

... (3.71) = ...

... (3.72) = ...

... (3.73) = ...

... (3.74) = ...

... (3.75) = ...

... (3.76) = ...

... (3.77) = ...

... (3.78) = ...

... (3.79) = ...

... (3.80) = ...

... (3.81) = ...

... (3.82) = ...

... (3.83) = ...

... (3.84) = ...

... (3.85) = ...

... (3.86) = ...

... (3.87) = ...

... (3.88) = ...

... (3.89) = ...

... (3.90) = ...

... (3.91) = ...

... (3.92) = ...

... (3.93) = ...

... (3.94) = ...

... (3.95) = ...

... (3.96) = ...

... (3.97) = ...

... (3.98) = ...

... (3.99) = ...

... (4.00) = ...

# 6

---

## EXTENSIONS

---

The techniques of the previous chapters apply compile time analysis to static schedules for HSDF graphs that have no decision making at the dataflow graph level. In this chapter we consider graphs with data dependent control flow. Recall that atomic actors in an HSDF graph are allowed to perform data-dependent decision making within their body, as long as their input/output behaviour respects SDF semantics. We show how some of the ideas we explored previously can still be applied to dataflow graphs containing actors that display data-dependent firing patterns, and therefore are not SDF actors.

### 6.1 The Boolean Dataflow model

The **Boolean Dataflow (BDF)** model was proposed by Lee [Lee91] and was further developed by Buck [Buck93] for extending the SDF model to allow data-dependent control actors in the dataflow graph. BDF actors are allowed to contain a *control* input, and the number of tokens consumed and produced on the arcs of a BDF actors can be a two-valued function of a token consumed at the control input. Actors that follow SDF semantics, i.e. that consume and produce fixed number of tokens on their arcs, are clearly a subset of the set of allowed BDF actors (SDF actors simply do not have any control inputs). Two basic dynamic

actors in the token flow model are the SWITCH and SELECT actors shown in Fig. 6.1. The switch actor consumes one Boolean-valued control token and another input token; if the control token is TRUE, the input token is copied to the output labelled T, otherwise it is copied to the output labelled F. The SELECT actor performs the complementary operation; it reads an input token from its T input if the control token is TRUE, otherwise it reads from its F input; in either case, it copies the token to its output. Constructs such as conditionals and data-dependent itera-

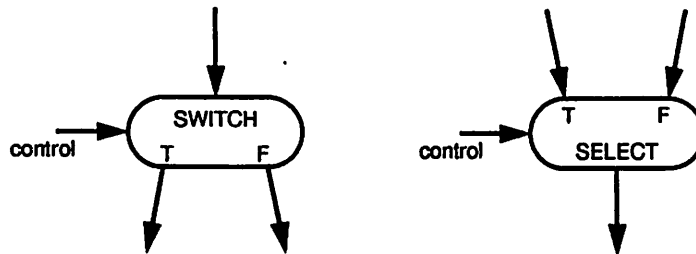


Figure 6.1. BDF actors SWITCH and SELECT

tions can easily be represented in a BDF graph, as illustrated in Fig. 6.2. The vertices A, B, C, etc. in Fig. 6.2 need not be atomic actors; they could also be arbitrary SDF graphs. A BDF graph allows SWITCH and SELECT actors to be connected in arbitrary topologies. Buck [Buck93] in fact shows that any Turing machine can be expressed as a BDF graph, and therefore the problems of determining whether such a graph deadlocks and whether it uses bounded memory are undecidable. Buck proposes heuristic solutions to these problems based on extensions of the techniques for SDF graphs to BDF model.

### 6.1.1 Scheduling

Buck presents techniques for statically scheduling BDF graphs on a single processor; his methods attempt to generate a sequential program without a dynamic scheduling mechanism, using `if-then-else` and `do-while` control constructs where required. Because of the inherent undecidability of determining deadlock behaviour and bounded memory usage, these techniques are not always



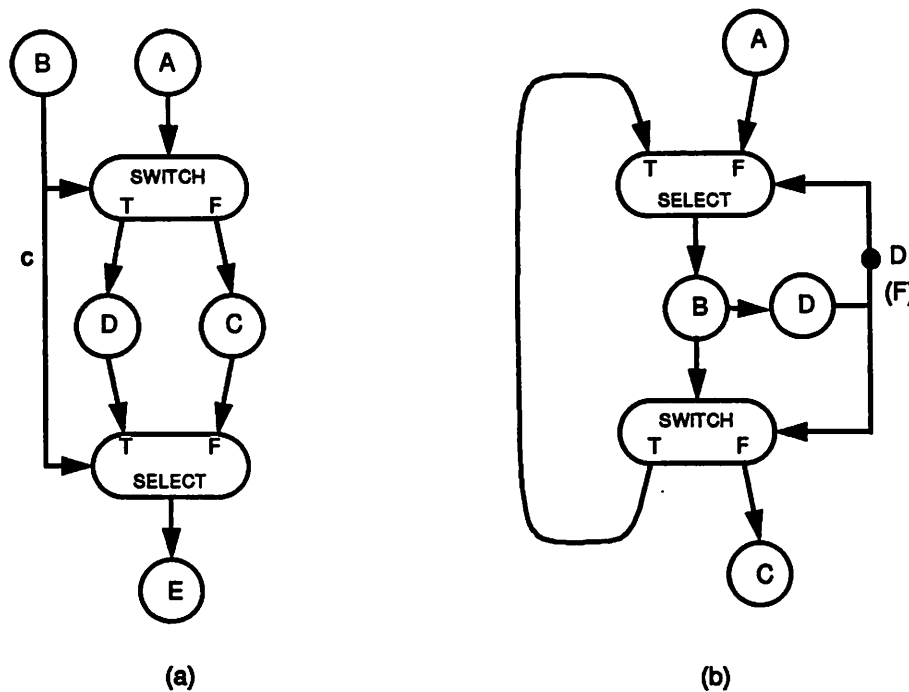


Figure 6.2. (a) Conditional (if-then-else) dataflow graph. The branch outcome is determined at run time by actor B. (b) Graph representing data-dependent iteration. The termination condition for the loop is determined by actor D.

guaranteed to generate a static schedule, even if one exists; a dynamically scheduled implementation, where a run time kernel decides which actors to fire, can be used when a static schedule cannot be found in a reasonable amount of time.

Automatic parallel scheduling of general BDF graphs is still an unsolved problem. A *naive* mechanism for scheduling graphs that contain SWITCH and SELECT actors is to generate an Acyclic Precedence Graph (APG), similar to the APG generated for SDF graphs discussed in section 1.2.1, for every possible assignment of the Boolean valued control tokens in the BDF graph. For example, the if-then-else graph in Fig. 6.2(a) could have two different APGs, shown in Fig.

6.3, and APGs thus obtained can be scheduled individually using a self-timed

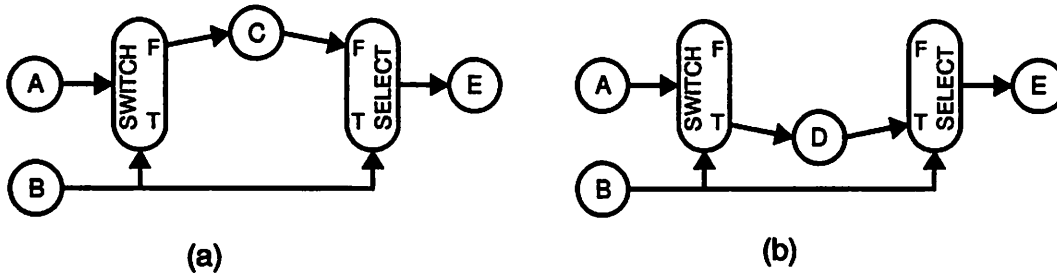


Figure 6.3. Acyclic precedence graphs corresponding to the if-then-else graph of Fig. 6.2. (a) corresponds to the TRUE assignment of the control token, (b) to the FALSE assignment.

strategy; each processor now gets several lists of actors, one list for each possible assignment of the control tokens. The problem with this approach is that for a graph with  $n$  different control tokens, there are  $2^n$  possible distinct APGs, each corresponding to each execution path in the graph. Such a set of APGs can be compactly represented using the so called Annotated Acyclic Precedence Graph (AAPG) of [Buck93] in which actors and arcs are annotated with conditions under which they exist in the graph. Buck uses the AAPG construct to determine whether a bounded-length uniprocessor schedule exists. In the case of multiprocessor scheduling, it is not clear how such an AAPG could be used to explore scheduling options for the different values that the control tokens could take, without explicitly enumerating all possible execution paths.

The main work in parallel scheduling of dataflow graphs that have dynamic actors has been the **Quasi-static scheduling** approach, first proposed by Lee [Lee88b] and extended by Ha [Ha92]. In this work, techniques have been developed that statically schedule standard dynamic constructs such as data-dependent conditionals, data-dependent iterations, and recursion. These constructs must be identified in a given dataflow graph, either manually or automatically, before Ha's techniques can be applied. These techniques make the simplifying assumption that the control tokens for different dynamic actors are independent of one another, and

that each control stream consists of tokens that take TRUE or FALSE values randomly and are independent and identically distributed (i.i.d.) according to statistics known at compile time. Such a quasi-static scheduling approach clearly does not handle a general BDF graph, although it is a good starting point for doing so.

Ha's quasi-static approach constructs a blocked schedule for one iteration of the dataflow graph. The dynamic constructs are scheduled in a hierarchical fashion; each dynamic construct is scheduled on a certain number of processors, and is then converted into a single node in the graph and is assigned a certain *execution profile*. A profile of a dynamic construct consists of the number of processors assigned to it, and the schedule of that construct on the assigned processors; the profile essentially defines the shape that a dynamic actor takes in the processor-time plane. When scheduling the remainder of the graph, the dynamic construct is treated as an atomic block, and its profile is used to determine how to schedule the remaining actors around it; the profile helps tiling actors in the processor-time plane with the objective of minimizing the overall schedule length. Such a hierarchical scheme effectively handles nested control constructs, e.g. nested conditionals.

One important aspect of quasi-static scheduling is determining execution profiles of dynamic constructs. Ha [Ha92] studies this problem in detail and shows how one can determine optimal profiles for constructs such as conditionals, data-dependent iteration constructs, and recursion, assuming certain statistics are known about the run time behaviour of these constructs.

We will consider only the conditional and the iteration construct here. We will assume that we are given a quasi-static schedule, obtained either manually or using Ha's techniques. We then explore how the techniques proposed in the previous chapters for multiprocessors that utilize a self-timed scheduling strategy apply when we implement a quasi-static schedule on a multiprocessor. First we propose an implementation of a quasi-static schedule on a shared memory multiprocessor, and then we show how we can implement the same program on the OMA architecture, using the hardware support provided in the OMA prototype for such an

implementation

## 6.2 Parallel implementation on shared memory machines

### 6.2.1 General strategy

A quasi-static schedule ensures that the pattern of processor availability is identical regardless of how the data-dependent construct executes at runtime; in the case of the conditional construct this means that irrespective of which branch is actually taken, the pattern of processor availability after the construct completes execution is the same. This has to be ensured by inserting idle time on processors when necessary. Fig. 6.4 shows a quasi-static schedule for a conditional construct. Maintaining the same pattern of processor availability allows static scheduling to proceed after the execution of the conditional; the data-dependent nature of the control construct can be ignored at that point. In Fig. 6.4 for example, the schedul-

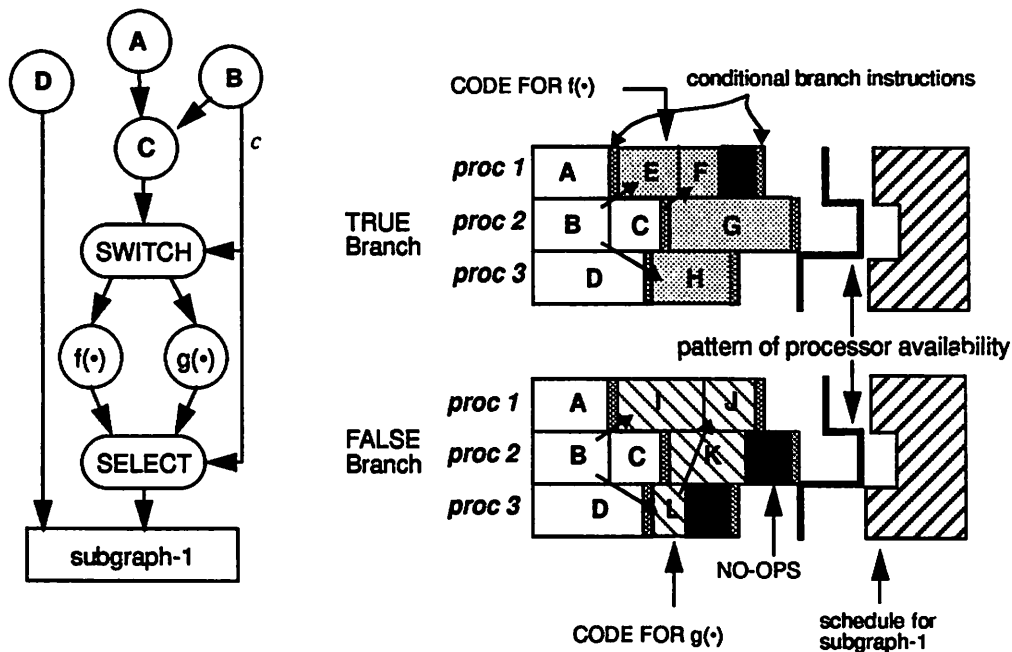


Figure 6.4. Quasi-static schedule for a conditional construct (adapted from [Lee88b])

ing of subgraph-1 can proceed independent of the conditional construct because the pattern of processor availability after this construct is the same independent of the branch outcome; note that “nops” (idle processor cycles) have been inserted to ensure this.

Multiprocessor implementation of a quasi-static schedule directly, however, implies enforcing global synchronization after each dynamic construct in order to ensure a particular pattern of processor availability. We therefore use a mechanism similar to the self-timed strategy; we first determine a quasi-static schedule using the methods of Lee and Ha, and then discard the timing information and the restrictions of maintaining a processor availability profile. Instead, we only retain the assignment of actors to processors, the order in which they execute, and also under what conditions on the Boolean tokens in the system the actor should execute. Synchronization between processors is done at run time whenever processors communicate. This scheme is analogous to constructing a self-timed schedule from a fully-static schedule, as discussed in section 1.2.2. Thus the quasi-static schedule of Fig. 6.4 can be implemented by the set of programs in Fig. 6.5, for the three processors. Here,  $\{r_{c1}, r_{c2}, r_1, r_2\}$  are the receive actors, and

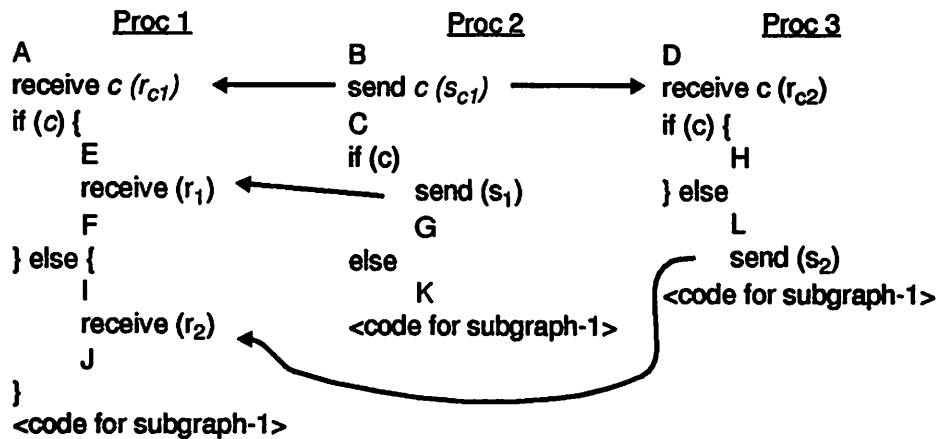


Figure 6.5. Programs on three processors for the quasi-static schedule of Fig. 6.4

$\{s_{c1}, s_1, s_2\}$  are the send actors. The subscript “c” refers to actors that communicate control tokens.

The main difference between such an implementation and the self-timed implementation we discussed in earlier chapters are the control tokens. Whenever a conditional construct is partitioned across more than one processor, the control token(s) that determine its behaviour must be broadcast to all the processors that execute that construct. Thus in Fig. 6.4 the value  $c$ , which is computed by Processor 2 (since the actor that produces  $c$  is assigned to Processor 2), must be broadcast to the other two processors. In a shared memory machine this broadcast can be implemented by allowing the processor that evaluates the control token (Processor 2 in our example) to write its value to a particular shared memory location preassigned at compile time; the processor will then update this location once for each iteration of the graph. Processors that require the value of a particular control token simply read that value from shared memory, and the processor that writes the value of the control token needs to do so only once. In this way actor executions can be conditioned upon the value of control tokens evaluated at run time. In the previous chapters we discussed synchronization associated with data transfer between processors. Synchronization checks must also be performed for the control tokens; the processor that writes the value of a token must not overwrite the shared memory location unless all processors requiring the value of that token have in fact read the shared memory location, and processors reading a control token must ascertain that the value they read corresponds to the current iteration rather than a previous iteration.

The need for broadcast of control tokens creates additional communication overhead that should ideally be taken into account during scheduling. The methods of Lee and Ha, and also prior research related to quasi-static scheduling that they refer to in their work, do not take this cost into account. Static multiprocessor scheduling applied to graphs with dynamic constructs taking costs of distributing control tokens into account is thus an interesting problem for further study.

## 6.2.2 Implementation on the OMA

Recall that the OMA architecture imposes an order in which shared memory is accessed by processors in the machine. This is done to implement the OT strategy, and is feasible because the pattern of processor communications in a self-timed schedule of an HSDF graph is in fact predictable. What happens when we want to run a program derived from a quasi-static schedule, such as the parallel program in Fig. 6.5, which was derived from the schedule in Fig. 6.4? Clearly, the order of processor accesses to shared memory is no longer predictable; it depends on the outcome of run time evaluation of the control token  $c$ . The quasi-static schedule of Fig. 6.4 specifies the schedules for the TRUE and FALSE branches of the conditional. If the value of  $c$  were always TRUE, then we can determine from the quasi-static schedule that the transaction order would be  $(s_{c1}, r_{c1}, r_{c2}, s_1, r_1, \langle \text{access order for subgraph-1} \rangle)$ , and if the value of  $c$  were always FALSE, the transaction order would be  $(s_{c1}, r_{c1}, r_{c2}, s_2, r_2, \langle \text{access order for subgraph-1} \rangle)$ . Note that writing the control token  $c$  once to shared memory is enough since the same shared location can

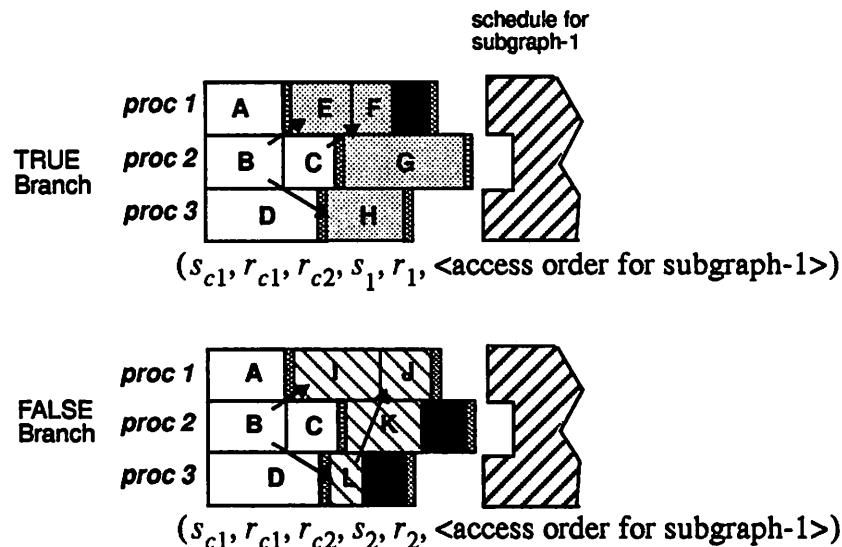


Figure 6.6. Transaction order corresponding to the TRUE and FALSE branches

be read by all processors requiring the value of  $c$ .

For the OMA architecture, our proposed strategy is to switch between these two access orders at run time. This is enabled by the preset feature of the transaction controller (Chapter 3, section 3.4.2). Recall that the transaction controller is implemented as a presettable schedule counter that addresses memory containing the processor IDs corresponding to the bus access order. To handle conditional constructs, we derive two bus access lists corresponding to each path in the program, and the processor that determines the branch condition (processor 2 in our example) forces the controller to switch between the access lists by loading the schedule counter with the appropriate value (address “7” in the bus access schedule of Fig. 6.7). Note from Fig. 6.7 that there are two points where the schedule counter can be set; one is at the completion of the TRUE branch, and the other is a jump into the FALSE branch. The branch into the FALSE path is best taken care of by processor 2, since it computes the value of the control token  $c$ , whereas the branch after the TRUE path (which bypasses the access list of the FALSE branch) is best taken care of by processor 1, since processor 1 already possesses the bus at the time when the counter needs to be loaded. The schedule counter load operations are easily incorporated into the sequential programs of processors 1 and 2.

The mechanism of switching between bus access orders works well when the number of control tokens is small. But if the number of such tokens is large, then this mechanism breaks down, even if we can efficiently compute a quasi-static schedule for the graph. To see why this is so, consider the graph in Fig. 6.8, which contains  $k$  conditional constructs in parallel paths going from the input to the output. The functions “ $f_i$ ” and “ $g_i$ ” are assumed to be subgraphs that are assigned to more than one processor. In Ha’s hierarchical scheduling approach, each conditional is scheduled independently; once scheduled, it is converted into an atomic node in the hierarchy, and a profile is assigned to it. Scheduling of the other conditional constructs can then proceed based on these profiles. Thus the scheduling complexity in terms of the number of parallel paths is  $O(k)$  if there are  $k$  parallel paths. If we implement the resulting quasi-static schedule in the



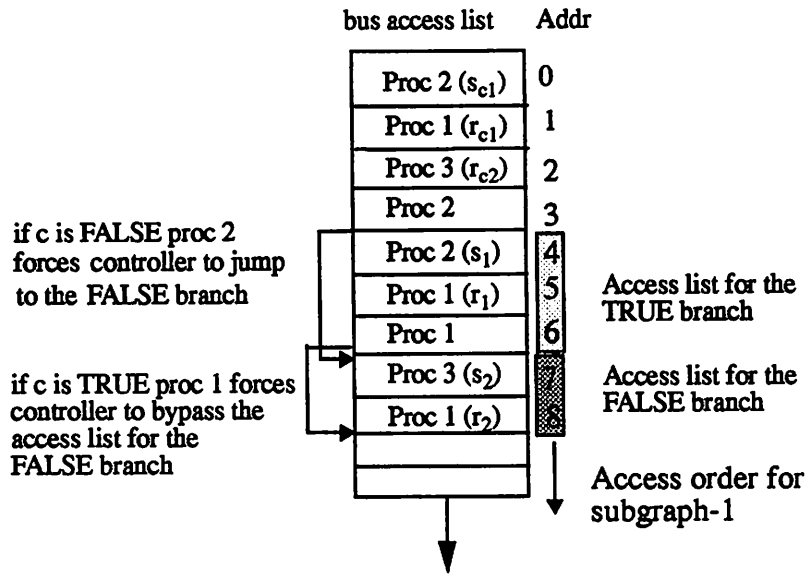


Figure 6.7. Bus access list that is stored in the schedule RAM for the quasi-static schedule of Fig. 6.6. Loading operation of the schedule counter conditioned on value of  $c$  is also shown.

manner stated in the previous section, and employ the OMA mechanism above, we would need one bus access list for every combination of the Booleans  $b_1, \dots, b_k$ . This is because each  $f_i$  and  $g_i$  will have its own associated bus access list, which then has to be combined with the bus access lists of all the other branches to yield one list. For example, if all Booleans  $b_i$  are true, then all the  $f_i$ 's are executed, and we get one access list. If  $b_1$  is TRUE, and  $b_2$  through  $b_k$  are FALSE, then  $g_1$  is executed, and  $f_2$  through  $f_k$  are executed. This corresponds to another bus access list. This implies  $2^k$  bus access lists for each of the combination of  $f_i$  and  $g_j$  that execute, i.e. for each possible execution path in the graph.

### 6.2.3 Improved mechanism

Although the idea of maintaining separate bus access lists is a simple mechanism for handling control constructs, it can sometimes be impractical, as in the example above. We propose an alternative mechanism based on *masking* that handles parallel conditional constructs more effectively.

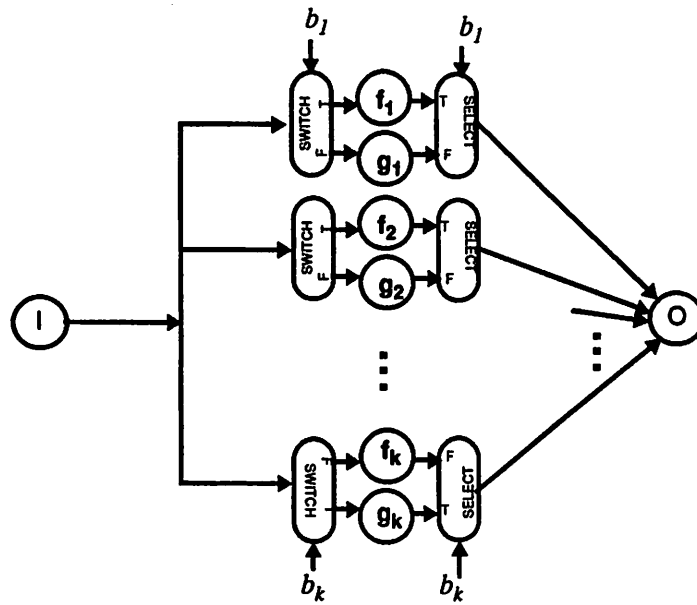


Figure 6.8. Conditional constructs in parallel paths

The main idea behind masking is to store an ID of a Boolean variable along with the processor ID in the bus access list. The Boolean ID determines whether a particular bus grant is “enabled.” This allows us to combine the access lists of all the nodes  $f_1$  through  $f_k$  and  $g_1$  through  $g_k$ . The bus grant corresponding to each  $f_i$  is tagged with the boolean ID of the corresponding  $b_i$ , and an additional bit indicates that the bus grant is to be enabled when  $b_i$  is TRUE. Similarly, each bus grant corresponding to the access list of  $g_i$  is tagged with the ID of  $b_i$ , and an additional bit indicates that the bus grant must be enabled only if the corresponding control token has a FALSE value. At runtime, the controller steps through the bus access list as before, but instead of simply granting the bus to the processor at the head of the list, it first checks that the control token corresponding to the Boolean ID field of the list is in its correct state. If it is in the correct state (i.e. it is TRUE for a bus grant corresponding to an  $f_i$  and FALSE for a bus grant corresponding to a  $g_i$ ), then the bus grant is performed, otherwise it is masked. Thus the run time values of the Booleans must be made available to the transaction controller for it to decide whether to mask a particular bus grant or not.

More generally, a particular bus grant should be enabled by a product

(AND) function of the Boolean variables in the dataflow graph, and the complement of these Booleans. Nested conditionals in parallel branches of the graph necessitate bus grants that are enabled by a product function; a similar need arises when bus grants must be reordered based on values of the Boolean variables. Thus, in general we need to implement an *annotated* bus access list of the form  $\{(c_1)ProcID_1, (c_2)ProcID_2, \dots\}$ ; each bus access is annotated with a Boolean valued condition  $c_i$ , indicating that the bus should be granted to the processor corresponding to  $ProcID_i$  when  $c_i$  evaluates to TRUE;  $c_i$  could be an arbitrary product function of the Booleans  $(b_1, b_2, \dots, b_n)$  in the system, and the complements of these Booleans (e.g.  $c_j = b_2 \cdot \bar{b}_4$ , where the bar over a variable indicates its complement).

This scheme is implemented as shown in Fig. 6.9. The schedule memory

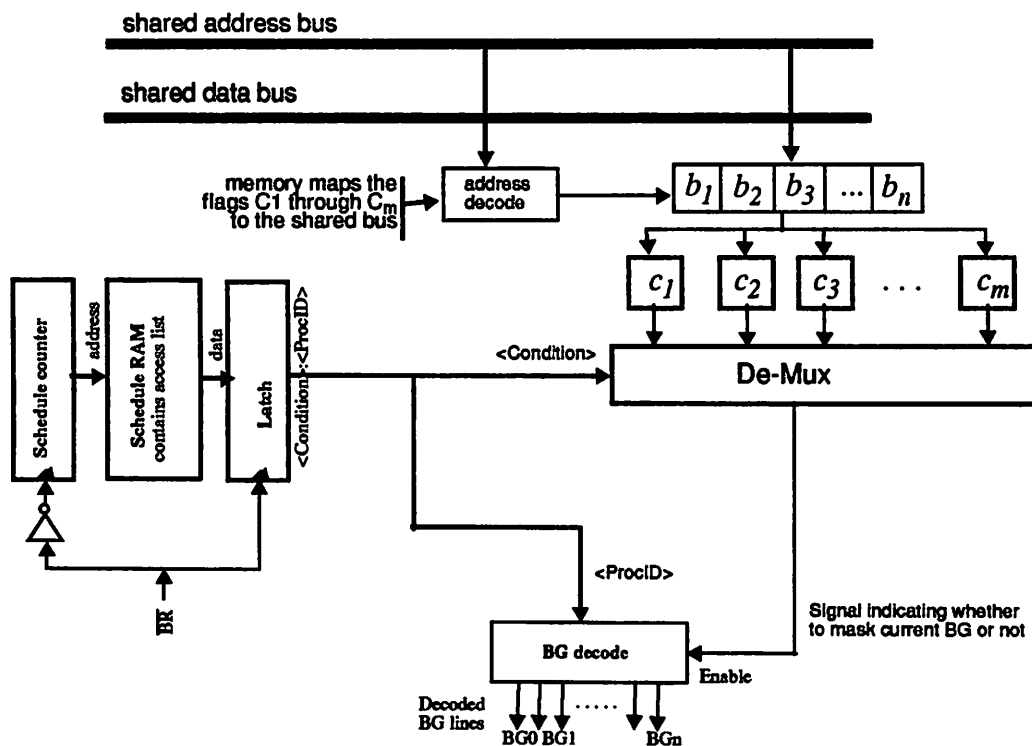


Figure 6.9. A bus access mechanism that selectively “masks” bus grants based on values of control tokens that are evaluated at run time

now contains two fields corresponding to each bus access: **<Condition>:<ProcID>**

instead of the <ProcID> field alone that we had before. The <Condition> field encodes a unique product  $c_i$  associated with that particular bus access. In the OMA prototype, we can use 3 bits for <ProcID>, and 5 bits for the <Condition> field. This would allow us to handle 8 processors and 32 product combinations of Booleans. There can be up to  $m = 3^n$  product terms in the worst case corresponding to  $n$  Booleans in the system, because each Boolean  $b_i$  could appear in the product term as itself, or its complement, or not at all (corresponding to a “don’t care”). It is unlikely that all  $3^n$  possible product terms will be required in practice; we therefore expect such a scheme to be practical. The necessary product terms ( $c_j$ ) can be implemented within the controller at compile time, based on the bus access pattern of the particular dynamic dataflow graph to be executed.

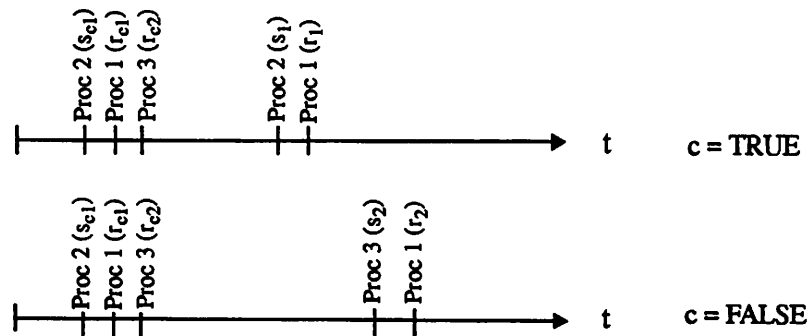
In Fig. 6.9, the flags  $b_1, b_2, \dots, b_n$ , are 1-bit memory elements (flip-flops) that are memory mapped to the shared bus, and store the values of the Boolean control tokens in the system. The processor that computes the value of each control token updates the corresponding  $b_i$  by writing to the shared memory location that maps to  $b_i$ . The product combinations  $c_1, c_2, \dots, c_n$ , are just AND functions of the  $b_i$ s and the complement of the  $b_i$ s, e.g.  $c_j$  could be  $b_2 \cdot \bar{b}_4$ . As the schedule counter steps through the bus access list, the bus grant is actually granted only if the condition corresponding to that access evaluates to TRUE; thus if the entry < $c_2$ ><Proc1> appears at the head of the bus access list, and  $c_2 = b_2 \cdot \bar{b}_4$ , then processor 1 receives a bus grant only if the control token  $b_2$  is TRUE and  $b_4$  is FALSE, otherwise the bus grant is masked and the schedule counter moves up to the next entry in the list.

This scheme can be incorporated into the transaction controller in our existing OMA architecture prototype, since the controller is implemented on an FPGA. The product terms  $c_1, c_2, \dots, c_n$  may be programmed into the FPGA at compile time; when we generate programs for the processors, we can also generate the *annotated* bus access list (a sequence of <Condition><Proc ID> entries), and a hardware description for the FPGA (in VHDL, say) that implements the required product terms.

## 6.2.4 Generating the annotated bus access list

Consider the problem of obtaining an annotated bus access list  $\{(c_1)ProcID_1, (c_2)ProcID_2, \dots\}$ , from which we can derive the sequence of  $\langle \text{Condition} \rangle \langle \text{Proc ID} \rangle$  entries for the mask-based transaction controller. A straightforward, even if inefficient, mechanism for obtaining such a list is to use enumeration; we simply enumerate all possible combinations of Booleans in the system ( $2^n$  combinations for  $n$  Booleans), and determine the bus access sequence (sequence of ProcID's) for each combination. Each combination corresponds to an execution path in the graph, and we can estimate the time of occurrence of bus accesses corresponding to each combination from the quasi-static schedule. For example, bus accesses corresponding to one schedule period of the two execution paths in the quasi-static schedule of Fig. 6.6 may be marked along the time axis as shown in Fig. 6.10 (we have ignored the bus access sequence corresponding to subgraph-1 to keep the illustration simple).

The bus access schedules for each of the combinations can now be collapsed into one annotated list, as in Fig. 6.10; the fact that accesses for each combination are ordered with respect to time allows us to enforce a global order on the



$\{(c) Proc2, (\bar{c}) Proc2, (c) Proc1, (\bar{c}) Proc1, (c) Proc3, (\bar{c}) Proc3,$  "annotated" list  
 $(c) Proc2, (c) Proc1, (\bar{c}) Proc3, (\bar{c}) Proc1\}$

Figure 6.10. Bus access lists and the annotated list corresponding to Fig. 6.6

accesses in the collapsed bus access list. The bus accesses in the collapsed list are annotated with their respective Boolean condition.

The collapsed list obtained above can be used as is in the masked controller scheme; however there is a potential for optimizing this list. Note, however, that the same transaction may appear in the access list corresponding to different Boolean combinations, because a particular Boolean token may be a “don’t care” for that bus access. For example, the first three bus accesses in Fig. 6.10 appear in both execution paths, because they are independent of the value of  $c$ . In the worst case a bus access that is independent of all Booleans will end up appearing in the bus access lists of all the Boolean combinations. If these bus accesses appear contiguously in the collapsed bus access sequence, we can combine them into one. For example, “ $(c) \text{ Proc2}, (\bar{c}) \text{ Proc2}$ ” in the annotated schedule of Fig. 6.10 can be combined into a single “Proc 2” entry, which is not conditioned on any control token. Consider another example: if we get contiguous entries “ $(b_1 \cdot \bar{b}_2) \text{ Proc3}$ ” and “ $(b_1 \cdot b_2) \text{ Proc3}$ ” in the collapsed list, we can replace the two entries with a single entry “ $(b_1) \text{ Proc3}$ ”.

More generally, if the collapsed list contains a contiguous segment of the form:

$$\{ \dots, (c_1) \text{ProcID}_k, (c_2) \text{ProcID}_k, (c_3) \text{ProcID}_k, \dots, (c_l) \text{ProcID}_k, \dots \},$$

we can write each of the contiguous segments as:

$$\{ \dots, (c_1 + c_2 + \dots + c_l) \text{ProcID}_k, \dots \},$$

where the bus grant condition is an expression  $(c_1 + c_2 + \dots + c_l)$ , which is a sum of products (SOP) function of the Booleans in the system. We can now apply 2-level minimization to determine a minimal representation of each of these expressions. Such 2-level minimization can be done by using a logic minimization tool such as ESPRESSO [Bray84], which simplifies a given SOP expression into an SOP representation with minimal number of product terms. Suppose the expression  $(c_1 + c_2 + \dots + c_l)$  can be minimized into another SOP expression

$(c_1' + c_2' + \dots + c_p')$ , where  $p < l$ . We can then replace the segment

$$\{ \dots, (c_1)ProcID_k, (c_2)ProcID_k, (c_3)ProcID_k, \dots, (c_l)ProcID_k, \dots \}$$

of the annotated bus access list with an equivalent segment of the form:

$$\{ \dots, (c_1')ProcID_k, (c_2')ProcID_k, (c_3')ProcID_k, \dots, (c_p')ProcID_k, \dots \} .$$

We can thus obtain a minimal set of contiguous appearances of a bus grant to the same processor.

Another optimization that can be performed is to combine annotated bus access lists with the switching mechanism of section 6.2.1. Suppose we have the following annotated bus access list:

$$\{ \dots, (b_1 \cdot \bar{b}_2)ProcID_i, (b_1 \cdot \bar{b}_3)ProcID_j, (b_1 \cdot b_4 \cdot b_5)ProcID_k, \dots \} .$$

Then, by “factoring”  $b_1$  out, we can equivalently write the above list as:

$$\{ \dots, (b_1) \{ (\bar{b}_2)ProcID_i, (\bar{b}_3)ProcID_j, (b_4 \cdot b_5)ProcID_k \}, \dots \} .$$

Now, we can skip over all the three accesses whenever the Boolean  $b_1$  is FALSE by loading the schedule counter and forcing it to increment its count by three, instead of evaluating each access separately, and skipping over each one individually. This strategy reduces overhead, because it costs an extra bus cycle to disable a bus access when a condition corresponding to that bus access evaluates to FALSE; by skipping over three bus accesses that we know are going to be disabled, we save three idle bus cycles. There is an added cost of one cycle for loading the schedule counter; the total savings in this example is therefore two bus cycles.

One of the problems with the above approach is that it involves explicit enumeration of all possible combinations of Booleans, the complexity of which limits the size of problems that can be tackled with this approach. An implicit mechanism for representing all possible execution paths is therefore desirable. One such mechanism is the use of Binary Decision Diagrams (BDDs), which have been used to efficiently represent and manipulate Boolean functions for the purpose of logic minimization [Bryant86]. BDDs have been used to compactly represent large

state spaces, and to perform operations implicitly over such state spaces when methods based on explicit techniques are infeasible. One difficulty we encountered in applying BDDs to our problem of representing execution paths is that it is not obvious how precedence and ordering constraints can be encoded in a BDD representation. The execution paths corresponding to the various Boolean combinations can be represented using a BDD, but it isn't clear how to represent the relative order between bus accesses corresponding to the different execution paths. We leave this as an area for future exploration.

### 6.3 Data-dependent iteration

A data-dependent iteration construct is shown in Fig. 6.2(b). A quasi-static schedule for such a construct may look like the one in Fig. 6.11. We are assuming

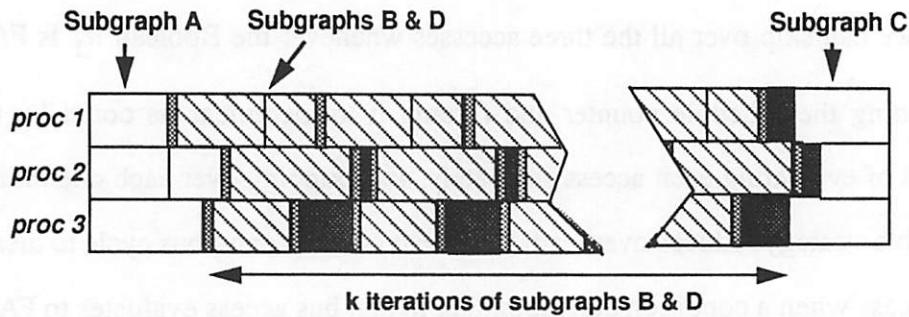


Figure 6.11. Quasi-static schedule for the data-dependent iteration graph of Fig. 6.2(b).

that A, B, C, and D of Fig. 6.2(b) are subgraphs rather than atomic actors.

Such a quasi-static schedule can also be implemented in a straightforward fashion on the OMA architecture, provided that the data-dependent construct spans all the processors in the system. The bus access schedule corresponding to the iterated subgraph is simply repeated until the iteration construct terminates. The processor responsible for determining when the iteration terminates can be made to force the schedule counter to loop back until the termination condition is reached.



This is shown in Fig. 6.12.

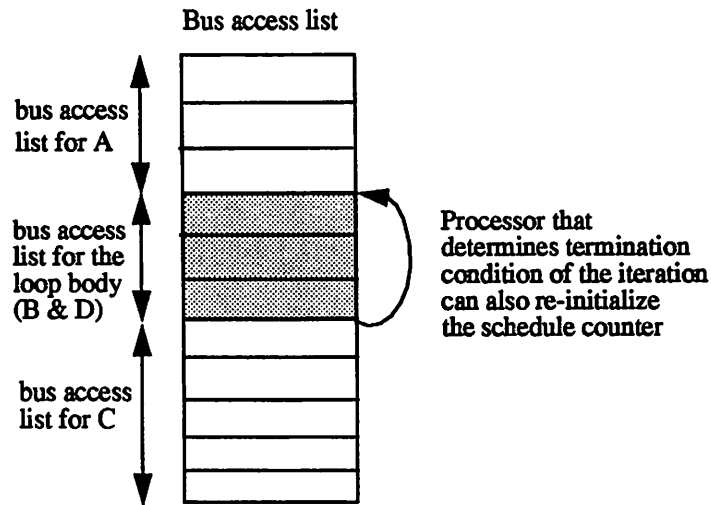


Figure 6.12. A possible access order list corresponding to the quasi-static schedule of Fig. 6.11.

## 6.4 Summary

This chapter dealt with extensions of the ordered transactions approach to graphs with data-dependent control flow. We briefly reviewed the Boolean Data-flow model, and the quasi-static approach to scheduling conditional and data-dependent iteration constructs. We then presented a scheme whereby the Ordered Memory Access board could be used when such control constructs are included in the dataflow graph. In this scheme, bus access schedules are computed for each set of values that the control tokens in the graph evaluate to, and the bus access controller is made to select between these lists at run time based on which set of values the control tokens actually take at any given time. This was also shown to be applicable to data-dependent iteration constructs. Such a scheme is feasible when the number of execution paths in the graph is small. We proposed another mechanism based on masking of bus accesses depending on run time values of control tokens, for handling the case when there are multiple conditional constructs in “parallel.”

# 7

---

## CONCLUSIONS AND FUTURE DIRECTIONS

---

In this thesis we explored techniques that minimize inter-processor communication and synchronization costs in statically scheduled multiprocessors for DSP. The main idea is that communication and synchronization in statically scheduled hardware is fairly predictable, and this predictability can be exploited to achieve our aims of low overhead parallel implementation at low hardware cost. The first technique we looked at was the ordered transactions strategy, where the idea is to predict the order of processor accesses to shared resources and enforce this order at run time. We applied this idea to a shared bus multiprocessor where the sequence of accesses to shared memory is pre-determined at compile time and enforced at run time by a controller implemented in hardware. We built a prototype of this architecture called the ordered memory access architecture, and demonstrated how we could achieve low overhead IPC at low hardware cost for the class of DSP applications that can be specified as SDF graphs, and for which good compile time estimates of execution times exist. We also introduced the IPC graph model for modeling self-timed schedules. This model was used to show that we can determine a particular transaction order such that enforcing this order at run time does not sacrifice performance when actual execution times of tasks are close to their compile time estimates. When actual running times differ from the compile time estimates, the computation performed is still correct, but the performance

(throughput) may be affected. We showed how such effects of run time variations in execution times on the throughput of a given schedule can be quantified.

The ordered transactions approach also extends to graphs that include constructs with data-dependent firing behaviour. We discussed how conditional constructs and data-dependent iteration constructs can be mapped to the OMA architecture, when the number of such control constructs is small — a reasonable assumption for most DSP algorithms.

Finally, we presented techniques for minimizing synchronization costs in a self-timed implementation that can be achieved by systematically manipulating the synchronization points in a given schedule; the IPC graph construct was used for this purpose. The techniques proposed include determining when certain synchronization points are redundant, transforming the IPC graph into a strongly connected graph, and then sizing buffers appropriately such that checks for buffer overflow by the sender can be eliminated. We also outlined a technique we call resynchronization, which introduces new synchronization points in the schedule with the objective of minimizing the overall synchronization cost.

The work presented in this thesis leads to several open problems and directions for further research.

Mapping a general BDF graph onto the OMA to make best use of our ability to switch between bus access schedules at run time is a topic that requires further study. Techniques for multiprocessor scheduling of BDF graphs could build upon the quasi-static scheduling approach, which restricts itself to certain types of dynamic constructs that need to be identified (for example as conditional constructs or data-dependent iterations) before scheduling can proceed. Assumptions regarding statistics of the Boolean tokens (e.g. the proportion of TRUE values that a control token assumes during the execution of the schedule) would be required for determining multiprocessor schedules for BDF graphs.

The OMA architecture applies the ordered transactions strategy to a shared bus multiprocessor. If the interprocessor communication bandwidth requirements for an application are higher than what a single shared bus can support, a more

elaborate interconnect, such as a crossbar or a mesh topology, may be required. If the processors in such a system run a self-timed schedule, the communication pattern is again periodic and we can predict this pattern at compile time. We can then determine the states that the crossbar in such a system cycles through or we can determine the sequence of settings for the switches in the mesh topology. The fact that we can determine this information should make it possible to simplify the hardware associated with these interconnect mechanisms, since the associated switches need not be configured at run time. How exactly this compile time information can be made use of for simplifying the hardware in such interconnects is an interesting problem for further study.

In the techniques we proposed in Chapter 5 for minimizing synchronization costs, no assumptions regarding bounds on execution times of actors in the graph were made. A direction for further work is to incorporate timing guarantees — for example, hard upper and lower execution time bounds, as Dietz, Zaafrani, and O’Keefe use in [Dietz92]; and handling of a mix of actors some of which have guaranteed execution time bounds, and some that have no such guarantees, as Filo, Ku, Coelho Jr., and De Micheli do in [Filo93]. Such guarantees could be used to detect situations in which data will always be available before it is needed for consumption by another processor.

Also, execution time guarantees can be used to compute tighter buffer size bounds. As a simple example, consider Fig. 7.1. Here, the analysis of section 5.8

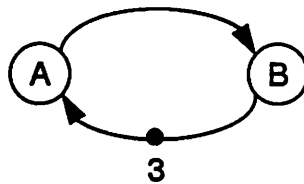


Figure 7.1. An example of how execution time guarantees can be used to reduce buffer size bounds.

yields a buffer size  $B_{fb}((A, B)) = 3$ , since 3 is the minimum path delay of a cycle that contains  $(A, B)$ . However, if  $t(A)$  and  $t(B)$ , the execution times of actors  $A$  and  $B$ , are guaranteed to be equal to the same constant, then it is easily verified that a buffer size of 1 will suffice for  $(A, B)$ . Systematically applying execution time guarantees to derive lower buffer size bounds appears to be a promising direction for further work.

Another interesting problem is applying the synchronization minimization techniques to graphs that contain dynamic constructs. Suppose we schedule a graph that contains dynamic constructs using a quasi-static approach, or a more general approach if one becomes available. Is it still possible to employ the synchronization optimization techniques we discussed in Chapter 5? The first step to take would be to obtain an IPC graph equivalent for the quasi-static schedule that has a representation for the control constructs that a processor may execute as a part of the quasi-static schedule. If we can show that the conditions we established for a synchronization operation to be redundant (in section 5.6) holds for all execution paths in the quasi-static schedule, then we could identify redundant synchronization points in the schedule. It may also be possible to extend the strongly-connect and resynchronization transformations to handle graphs containing conditional constructs; these issues require further investigation.

## REFERENCES

[Ack82]

W. B. Ackerman, "Data Flow Languages," *IEEE Computer Magazine*, Vol. 15, No. 2, February, 1982.

[Adam74]

T. L. Adam, K. M. Chandy and J. R. Dickson, "A Comparison of List Schedules for Parallel Processing Systems," *Communications of the ACM*, Vol. 17, No. 12, pp. 685-690, December 1974.

[Aik88]

A. Aiken, and A. Nicolau, "Optimal Loop Parallelization," *Proceedings of the SIGPLAN 88 Conference on Programming Language Design and Implementation*, 1988.

[Alle87]

R. Allen and K. Kennedy, "Automatic Transformation of Fortran Programs to Vector Form," *ACM Transactions on Programming Languages and Systems*, Vol. 9, No. 4, October, 1987.

[Amb192]

A. L. Ambler, M. M. Burnett, and B. A. Zimmerman, "Operational Versus Definitional: A Perspective on Programming Paradigms," *IEEE Computer Magazine*, Vol. 25, No. 9, September, 1992.

[Ariel91]

*User's Manual for the S-56X*, Ariel Corporation, Highland Park, New Jersey, 1991.

[Arvi90]

Arvind and R. S. Nikhil, "Executing a Program on the MIT Tagged-Token

Dataflow Architecture,” *IEEE Transactions on Computers*, Vol. C-39, No. 3, March, 1990.

[Arvi91]

Arvind, L. Bic, and T. Ungerer, “Evolution of Dataflow Computers,” *Advanced Topics in Dataflow Computing*, Prentice-Hall, 1991.

[Bacc92]

F. Baccelli, G. Cohen, G. J. Olsder, and J.-P. Quadrat, *Synchronization and Linearity*, John Wiley & Sons Inc., New York, pp. 103-154, 1992.

[Bacc92]

F. Baccelli and Z. Liu, “On a Class of Stochastic Recursive Sequences Arising in Queueing Theory,” *The Annals of Probability*, Vol. 20, No. 1, pp. 350-374.

[Barr91]

B. Barrera and E. A. Lee, “Multirate Signal Processing in Comdisco’s SPW,” *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Toronto, April, 1991.

[Ben91]

A. Benveniste and G. Berry, “The Synchronous Approach to Reactive and Real-Time Systems,” *Proceedings of the IEEE*, Vol. 79, No. 9, 1991, pp.1270-1282.

[Bhat95a]

S. S. Bhattacharyya, S. Sriram, and E. A. Lee, *Optimizing Synchronization in Multiprocessor Implementations of Iterative Dataflow Programs*, ERL Technical Report UCB/ERL M95/2, University of California, Berkeley, CA 94720, January 5, 1995.

[Bhat95b]

S. S. Bhattacharyya, S. Sriram, and E. A. Lee, *Resynchronization for Embedded Multiprocessors*, Draft XXX, to be made into an ERL memo.

[Bhat93]

S. S. Bhattacharyya and Edward A. Lee, "Scheduling Synchronous Data-flow Graphs for Efficient Looping," *Journal of VLSI Signal Processing*, No. 6, 1993.

[Bhat94]

S. S. Bhattacharyya and E. A. Lee, "Memory Management for Dataflow Programming of Multirate Signal Processing Algorithms," *IEEE Trans. on Signal Processing*, Vol. 42, No. 5, May 1994.

[Bier89]

J. Bier and E. A. Lee, "Frigg: A Simulation Environment for Multiprocessor DSP System Development", *Proceedings of the International Conference on Computer Design*, pp. 280-283, October, 1989, Boston, MA.

[Bier90]

J. Bier, S. Sriram and E. A. Lee, "A Class of Multiprocessor Architectures for Real-Time DSP," *VLSI DSP IV*, ed. H. Moscovitz, IEEE Press, November, 1990.

[Bils94]

G. Bilsen, M. Engels, R. Lauwereins and J. A. Peperstraete, "Static Scheduling of Multi-Rate and Cyclo-Static DSP Applications," *VLSI Signal Processing VII*, IEEE Press, 1994.

[Blaz87]

J. Blazewicz, "Selected Topics in Scheduling Theory," in *Surveys in Combinatorial Optimization*, North Holland Mathematica Studies, 1987.

[Bork88]



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

[Bray84]

R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, 1984.

[Bryant86]

R. E. Bryant, "Graph Based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers*, C-35(8), pp. 677-691, August 1986.

[Buck93]

J. T. Buck, *Scheduling Dynamic Dataflow Graphs with Bounded Memory using the Token Flow Model*, Ph. D. Thesis, Memorandum No. UCB/ERLM93/69, Electronics Research Laboratory, University of California at Berkeley, September, 1993.

[Buck94]

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*, January 1994.

[Cam92]

J. Campos and M. Silva, "Structural Techniques and Performance Bounds of Stochastic Petri Net Models," in *Advances in Petri Nets 1993*, pp. 325-349, edited by G. Rosenberg, Springer-Verlag, 1993.

[Chase84]

M. Chase, "A Pipelined Data Flow Architecture for Digital Signal Process-

ing: The NEC mPD7281," *IEEE Workshop on Signal Processing*, November 1984.

[Chao92]

L. F. Chao, and E. Sha, "Unfolding and Retiming Data-Flow DSP Programs for RISC Multiprocessor Scheduling," *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, April 1992.

[Chre83]

P. Chretienne, "Timed Event Graphs: A Complete Study of their Controlled Executions," *International Workshop on Timed Petri Nets*, July 1985.

[Cohen85]

G. Cohen, D. Dubois, J. Quadrat, "A Linear System Theoretic View of Discrete Event Processes and its use for Performance Evaluation in Manufacturing," *IEEE Transactions on Automatic Control*, March 1985.

[Corm92]

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

[Cun79]

R. Cunningham-Green, "Minimax Algebra," *Lecture Notes in Economics and Mathematical Systems*, Vol. 166, Springer-Verlag 1979.

[deGroot92]

S. M. H. de Groot, S. Gerez, and O. Herrmann, "Range-Chart-Guided Iterative Data-Flow Graph Scheduling," *IEEE Transactions on Circuits and Systems*, pp. 351-364, May 1992.

[DeMich94]

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

[Denn80]

J. B. Dennis, "Dataflow Supercomputers," *IEEE Computer Magazine*, Vol. 13, No. 11, November, 1980.

[Dietz92]

H. G. Dietz, A. Zaafrani, and M. T. O'Keefe, "Static Scheduling for Barrier MIMD Architectures," *Journal of Supercomputing*, Vol. 5, No. 4, 1992.

[Durr91]

R. Durrett, *Probability: Theory and Examples*, Wadsworth & Brooks/Cole Statistics/Probability Series, Pacific Grove, CA, 1991.

[Filo93]

D. Filo, D. C. Ku, C. N. Coelho Jr., and G. De Micheli, "Interface Optimization for Concurrent Systems Under Timing Constraints," *IEEE Transactions on Very Large Scale Integration*, Vol. 1, No. 3, September, 1993.

[Gajs94]

D. J. Gajski, F. Vahid, S. Narayan and J. Gong, *Specification and Design of Embedded Systems*, Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[Garey79]

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, 1979.

[Gasp92]

F. Gasperoni and Uwe Schweigelshohn, "Scheduling Loops on Parallel Processors: A Simple Algorithm with Close to Optimum Performance," *International Conference on Vector & Parallel Processors*, Sept. 1992.

[Gau91]

J. Gaudiot and L. Bic, *Advanced Topics in Data-flow Computing*, Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[Gov94]

R. Govindarajan, G. R. Gao, and P. Desai, "Minimizing Memory Requirements in Rate-Optimal Schedules," *Proceedings of the International Conference on Application Specific Array Processors*, San Francisco, August, 1994.

[Grin88]

C. M. Grinstead, "Cycle Lengths in  $A^{k^*}b^*$ ," *SIAM Journal on Matrix Analysis*, October 1988.

[Ha92]

S. Ha, *Compile Time Scheduling of Dataflow Program Graphs with Dynamic Constructs*, Ph. D. Thesis, Memorandum No. UCB/ERL M92/43, April 1992.

[Hal91]

N. Halbwachs, P. Caspi, P. Raymond and D. Pilaud, "The Synchronous Data Flow Programming Language LUSTRE," *Proceedings of the IEEE*, September 1991.

[Hal93]

N. Halbwachs, *Synchronous Programming of Reactive Systems*, Kluwer Academic Publishers, 1993.

[Hav91]

B. R. Haverkort, "Approximate Performability Analysis Using Generalized Stochastic Petri Nets," *Proceedings of the Fourth International Workshop on Petri Nets and Performance Models*, Melbourne, Australia, pp. 176-

185, 1991.

[Hu61]

T. C. Hu, "Parallel Sequencing and Assembly Line Problems," *Operations Research*, Vol. 9, 1961.

[Huis93]

J. A. Huisken et. al., "Synthesis of Synchronous Communication Hardware in a Multiprocessor Architecture," *Journal of VLSI Signal Processing*, Vol. 6, pp.289-299, 1993.

[Kala93]

A. Kalavade, and E. A. Lee, "A Hardware/Software Codesign Methodology for DSP Applications," *IEEE Design and Test*, September 1993, Vol. 10, No. 3, pp. 16-28.

[Karp66]

R. M. Karp and R. E. Miller, "Properties of a Model for Parallel Computations: Determinacy, Termination Queueing," *SIAM Journal of Applied Math.*, Vol. 14, No. 6, November, 1966.

[Karp78]

R. M. Karp, "A Characterization of the Minimum Cycle Mean in a Digraph," *Discrete Mathematics*, Vol. 23, 1978.

[Koh90]

W. Koh, *A Reconfigurable Multiprocessor System for DSP Behavioral Simulation*, Ph. D. Thesis, Memorandum No. UCB/ERL M90/53, Electronics Research Laboratory, University of California, Berkeley, June 1990.

[Koh75]

W. H. Kohler, "A Preliminary Evaluation of Critical Path Method for Scheduling Tasks on Multiprocessor Systems," *IEEE Transactions on*

*Computers*, pp. 1235-1238, December 1975.

[Kung87]

S. Y. Kung, P. S. Lewis, and S. C. Lo, "Performance Analysis and Optimization of VLSI Dataflow Arrays" *Journal of Parallel and Distributed Computing*, Vol. 4, pp. 592-618, 1987.

[Kung88]

S. Y. Kung, *VLSI array processors*, Englewood Cliffs, N. J., Prentice Hall, 1988.

[Lam88]

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

[Lam89]

M. Lam, *A Systolic Array Optimizing Compiler*, Kluwer Academic Publishers, Norwell, Massachusetts, 1989.

[Lamp86]

L. Lamport, "The Mutual Exclusion Problem: Part I and II," *Journal of the ACM*, Vol. 33, No. 2, pp. 313-348, April 1986.

[Laps91]

P. D. Lapsley, *Host Interface and Debugging of Dataflow DSP Systems*, M. S. Thesis, Electronics Research Laboratory, University of California, Berkeley, CA 94720, December, 1991.

[Lauw90]

R. Lauwereins, M. Engels, J.A. Peperstraete, E. Steegmans, and J. Van Ginderdeuren, "GRAPE: A CASE Tool for Digital Signal Parallel Processing," *IEEE ASSP Magazine*, Vol. 7, No. 2, April, 1990.

[Law76]

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

[Lee86]

E. A. Lee, *A Coupled Hardware and Software Architecture for Programmable DSPs*, Ph. D. Thesis, Department of EECS, University of California Berkeley, May 1986.

[Lee87]

E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Dataflow Programs for Digital Signal Processing," *IEEE Transactions on Computers*, February, 1987.

[Lee88a]

E. A. Lee, "Programmable DSP Architectures, Part I", *IEEE Acoustics, Speech, and Signal Processing Magazine*, October, 1988.

[Lee88b]

E. A. Lee, "Recurrences, Iteration, and Conditionals in Statically Scheduled Block Diagram Languages," *VLSI Signal Processing III*, IEEE, Press, 1988.

[Lee89]

E. A. Lee, and S. Ha, "Scheduling Strategies for Multiprocessor Real-Time DSP," *Globecom*, Dallas Texas, pp. 1279-1283, November 1989.

[Lee90]

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

[Lee91]

E. A. Lee, "Consistency in Dataflow Graphs", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 2, No. 2, April 1991.

[Lee93]

E. A. Lee, "Representing and Exploiting Data Parallelism Using Multidimensional Dataflow Diagrams," *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, Minneapolis, Vol. 1, pp. 453-456, April 1993.

[Lee95]

E. A. Lee and T. M. Parks, "Dataflow Process Networks," *Proceedings of the IEEE*, May 1995.

[Lei91]

C. E. Leiserson and J. B. Saxe, "Retiming Synchronous Circuitry," *Algorithmica*, Vol. 6, No. 1, pp. 5-35, 1991.

[Len92]

D. Lenoski, J. Laudon, K. Gharachorloo, W. D. Weber and J. Hennessey, "The Stanford Dash multiprocessor," *IEEE Computer*, March 1992.

[Lew81]

H. R. Lewis and C. H. Papadimitriou, *Elements of the Theory of Computation*, Prentice Hall, 1982.

[Liao94]

G. Liao, G. R. Gao, E. Altman, and V. K. Agarwal, *A Comparative Study of DSP Multiprocessor List Scheduling Heuristics*, technical report, School of Computer Science, McGill University, 1994.

[Liao95]

S. Liao, S. Devadas, K. Keutzer, S. Tjiang and A. Wang, "Code Optimization Techniques for Embedded DSP Microprocessors," *Proceedings of the*



*32nd Design Automation Conference, June 1995.*

[Li95]

Y. S. Li and S. Malik, "Performance Analysis of Embedded Software Using Implicit Path Enumeration," *Proceedings of the 32nd Design Automation Conference, June 1995.*

[Messer88]

D. G. Messerschmitt, "Breaking the Recursive Bottleneck," in *Performance Limits in Communication Theory and Practice*, J. K. Skwirzynski editor, Chapter 7, Kluwer Academic Publishers, 1988.

[Moll82]

Michael K. Molloy, "Performance Analysis Using Stochastic Petri Nets," *IEEE Transactions on Computers, Vol. c-31, No. 9, September 1982.*

[Moto89]

*DSP96002 IEEE Floating-Point Dual-Port Processor User's Manual*, Motorola Inc., 1989.

[Moto90]

*DSP96000ADS Application Development System Reference Manual*, Motorola Inc., 1990.

[Mouss92]

F. Moussavi and D. G. Messerschmitt, "Statistical Memory Management for Digital Signal Processing," *Proceedings of the 1992 IEEE International Symposium on Circuits and Systems, Vol. 2, pp. 1011-14, May 1992.*

[Mur89]

T. Murata, "Petri nets: Properties, Analysis, and Applications," *Proceedings of the IEEE, Vol. 77, pp. 39-58, January 1989.*

[Ols90]

G. J. Olsder, J. A. C. Resing, R. E. De Vries and M. S. Keane, "Discrete Event Systems with Stochastic Processing Times," *IEEE Transactions on Automatic Control*, March 1990, Vol.35, No.3, pp. 299-302.

[Ols89]

G. J. Oldser, "Performance Analysis of Data-driven Networks," *Systolic Array Processors; Contributions by Speakers at the International Conference on Systolic Arrays*; Edited by: J. McCanny, J. McWhiter, E. Swartzlander Jr., Prentice Hall, New York, 1989, pp. 33-41.

[Ous94]

J. K. Ousterhout, *An Introduction to Tcl and Tk*, Addison-Wesley Publishing, Redwood City, CA, 1994.

[Pap90]

G. M. Papadopoulos, "Monsoon: A Dataflow Computing Architecture Suitable for Intelligent Control," *Proceedings of the 5th IEEE International Symposium on Intelligent Control*, 1990.

[Parhi91]

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

[Patt90]

D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*, Morgan Kaufman Publishers, 1990.

[Peter81]

J. L. Peterson, *Petri Net Theory and the Modelling of Systems*, Prentice-Hall Inc., 1981.

[Pin95a]

J. Pino, S. Ha, E. A. Lee, and J. T. Buck, "Software Synthesis for DSP Using Ptolemy," *Journal of VLSI Signal Processing*, Vol. 9, No. 1, January, 1995.

[Pin95b]

J. L. Pino, S. S. Bhattacharyya and E. A. Lee, "A Hierarchical Multiprocessor Scheduling System for DSP Applications," to appear in *IEEE Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, CA, October 29 - November 1, 1995.

[Pow92]

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

[Prin91]

H. Printz, *Automatic Mapping of Large Signal Processing Systems to a Parallel Machine*, Ph.D. thesis, Memorandum CMU-CS-91-101, School of Computer Science, Carnegie Mellon University, May, 1991.

[Prin92]

H. Printz, "Compilation of Narrowband Spectral Detection Systems for Linear MIMD Machines," *Proceedings of the International Conference on Application Specific Array Processors*, Berkeley, August, 1992.

[Ptol94]

Ptolemy design group, UC Berkeley, *The Almagest*, UC Berkeley, 1994.

[Rab91]

J. M. Rabaey, C. Chu, P. Hoang, and M. Potkonjak, "Fast Prototyping of Datapath Intensive Architectures," *IEEE Design and Test of Computers*,

Vol. 8, No. 2, pp. 40-51, June 1991.

[Rajs94]

Sergio Rajsbaum and Mosha Sidi, "On the Performance of Synchronized Programs in Distributed Networks with Random Processing Times and Transmission Delays," *IEEE Transactions on Parallel and Distributed Systems*. Vol. 5, No. 9, September 1994.

[Ram80]

C. V. Ramamoorthy and G. S. Ho, "Performance Evaluation of Asynchronous Concurrent Systems using Petri Nets," *IEEE Transactions on Software Engineering*, Vol. SE-6, No. 5, pp. 440-449, September 1980.

[Ram72]

C. V. Ramamoorthy, K. M. Chandy, and M. J. Gonzalez, "Optimal Scheduling Strategies in Multiprocessor Systems," *IEEE Transactions on Computers*, Feb. 1972.

[Reit68]

R. Reiter, "Scheduling Parallel Computations", *Journal of the Association for Computing Machinery*, October 1968.

[Renf81]

M. Renfors and Y. Neuvo, "The Maximum Sampling Rate of Digital Filters Under Hardware Speed Constraints," *IEEE Transactions on Circuits and Systems*, CAS-28(3), March 1981.

[Ritz92]

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

[Sark89]

V. Sarkar, "Partitioning and Scheduling Parallel Programs for Multiprocessors," *Research Monographs in Parallel and Distributed Computing*, Pitman, London, 1989.

[Sha89]

P. L. Shaffer, "Minimization of Interprocessor Synchronization in Multiprocessors with Shared and Private Memory," *International Conference on Parallel Processing*, 1989.

[Schw85]

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

[Shen92]

N. Shenoy, R. K. Brayton, A. L. Sangiovanni-Vincentelli, "Graph algorithms for clock schedule optimization," in *1992 IEEE/ACM International Conference on Computer-Aided Design. Digest of Technical Papers* (Cat. No.92CH03183-1), Santa Clara, CA, pp. 132-6.

[Sih91]

G. C. Sih, *Multiprocessor Scheduling to account for Interprocessor Communication*, Ph. D. Thesis, Department of EECS, University of California Berkeley, April 1991.

[Stolz91]

A. Stolze, *A Real Time Large Vocabulary Connected Speech Recognition System*, Ph. D. Thesis, Department of EECS, University of California Berkeley, December 1991.

[Sriv92]

M. B. Srivastava, *Rapid-Prototyping of Hardware and Software in a Uni-*

*fied Framework*, Ph. D. Thesis, Memorandum No. UCB/ERL M92/67, Electronics Research Laboratory, University of California, Berkeley, June 1992.

[Thor86]

*Thor Tutorial*, VLSI CAD Group, Stanford University, 1986.

[Vai93]

P. P. Vaidyanathan, *Multirate Systems and Filter Banks*, Prentice Hall, 1993.

[Veig90]

M. Veiga, J. Parera and J. Santos, "Programming DSP Systems on Multiprocessor Architectures," *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Albuquerque, April 1990.

[Yao93]

L. Yao and C. M. Woodside, "Iterative Decomposition and Aggregation of Stochastic Marked Graph Petri Nets," in *Advances in Petri Nets 1993*, pp. 325-349, edited by G. Rosenberg, Springer-Verlag, 1993.

[Zaky89]

A. Zaky and P. Sadayappan, "Optimal Static Scheduling of Sequential Loops on Multiprocessors," *Proceedings of the International Conference on Parallel Processing*, Vol. 3, pp. 130-137, 1989.

[Zivo94a]

V. Zivojnovic, S. Ritz and H. Meyr, "Retiming of DSP programs for optimum vectorization," *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, April 1994.

[Zivo94b]

V. Zivojnovic, H. Koerner, and H. Meyr, "Multiprocessor Scheduling with

A-priori Node Assignment," *VLSI Signal Processing VII*, IEEE Press, 1994.

[Zivo95]

V. Zivojnovic, J. M. Velarde, C. Schlager and H. Meyer, "DSPSTONE: A DSP-Oriented Benchmarking Methodology," *Proceedings of ICSPAT*, 1995.