

Copyright © 1992, 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.

**COMPILING REAL-TIME DIGITAL SIGNAL  
PROCESSING APPLICATIONS ONTO  
MULTIPROCESSOR SYSTEMS**

by

Phu D. Hoang

Memorandum No. UCB/ERL M92/68

30 June 1992

**COMPILING REAL-TIME DIGITAL SIGNAL  
PROCESSING APPLICATIONS ONTO  
MULTIPROCESSOR SYSTEMS**

by

Phu D. Hoang

Memorandum No. UCB/ERL M92/68

30 June 1992

**ELECTRONICS RESEARCH LABORATORY**

College of Engineering  
University of California, Berkeley  
94720

**COMPILING REAL-TIME DIGITAL SIGNAL  
PROCESSING APPLICATIONS ONTO  
MULTIPROCESSOR SYSTEMS**

by

Phu D. Hoang

Memorandum No. UCB/ERL M92/68

30 June 1992

**ELECTRONICS RESEARCH LABORATORY**

College of Engineering  
University of California, Berkeley  
94720

# COMPILING REAL-TIME DIGITAL SIGNAL PROCESSING APPLICATIONS ONTO MULTIPROCESSOR SYSTEMS

by

**Phu D. Hoang**

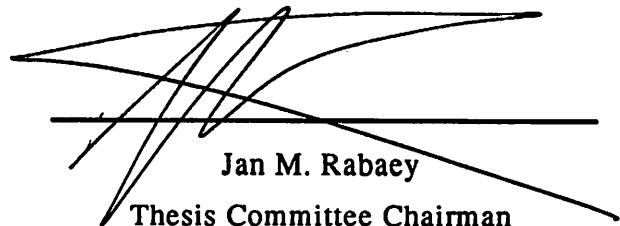
## ABSTRACT

The goal of this research is to develop a set of Computer-Aided Design (CAD) tools to support the real-time implementation of Digital Signal Processing (DSP) applications onto multiple programmable processors. The work has resulted in a complete DSP design environment, called McDAS, which can compile high level DSP applications directly down to parallel code for MIMD multiprocessors.

One of the major challenges of the research is the assignment and scheduling of tasks onto the processors in such a way as to maximize the throughput of the resultant implementation while considering interprocessor communication delays and resource constraints imposed by the target architecture. The scheduler in McDAS exploits pipelining, retiming, and parallel execution simultaneously, allowing the environment to efficiently support a wide range of applications with different types of concurrency. Users can invoke the scheduler with different architecture configurations to explore implementation trade-offs.

The code generator is similarly retargetable to different multiprocessor architectures as well as core processors. Data buffers and synchronizations are automatically inserted to ensure correct execution. The final implementation can be used for simulation speedup or real-time processing.

The results on a set of benchmarks demonstrate McDAS's ability to achieve near optimal speedups across a wide range of applications.



Jan M. Rabaey  
Thesis Committee Chairman

# ACKNOWLEDGEMENT

So many people have enriched my life during my years at Berkeley. First of all, I am indebted to my research advisor Jan Rabaey for his generous support and encouragement of my work, and for allowing me the freedom to be involved in other projects in addition to my main research. Of these ventures, working on the HYPER high level synthesis project has been the most rewarding. Thank you, Jan, for your advice, for your vision, for your unmatched programming style, for everything.

I would also like to thank Professors Edward Lee and Sadashiv Adiga, who served on my qualifying exam committee, as well as my thesis committee. Professor Lee, in particular, sparked my interest in DSP multiprocessor scheduling, and continued to provide invaluable guidance to me during our many discussions. Thank you, Edward, for all of your help.

My first two years, and my masters thesis, were with the Optimization CAD group, headed by Professor Polak. I would like to thank him for his help and guidance.

The BJgroup headed by Professors Brodersen and Rabaey was an excellent environment for research. I thank Professor Brodersen for sharing his deep knowledge and experience in VLSI Signal Processing. Special thanks to Miodrag Potkonjak, Paul Landman, Sean Huang, Lisa Guerra, Ingrid Verbauwhede, Wook Koh, Alfred Yeungk, and Mani Srivastava for their endless reviews of my work.

The friends that I have made at Berkeley will forever enrich my life. Miodrag Potkonjak was and will always be a friend, a teacher, and a brother. My first roommate, David Sze, was always there when I needed him. To him, my deepest appreciation. To my great friend John Coble: Hail to the Redskins!! Away from home, my Vietnamese friends at Berkeley were my family. Thank you, Quyen, Hung, Dinh, Tai, Khanh, Chris, Quan, and Johnny, for all the dinners, movies, and late-night card games.

To my best friend Quyen: All my warmth and love. Thank you for your patience, your forgiveness, your caring, and your belief in me. The last two years have been wonderful.

My brothers and sisters are a source of inspiration. Thank you each and every one of you for spoiling me all of these years.

Lastly, I would like to dedicate this thesis to my parents, who I love more than anything in this world. To my father, for his infinite love and sacrifice. To my mother: I am 1 year, 1 month, and 27 days late. I wish you were here to celebrate with me today.

---

# TABLE OF CONTENTS

---

<b>1 INTRODUCTION.....</b>	<b>1</b>
1.1 OVERVIEW .....	1
1.2 APPLICATION DOMAIN.....	2
1.3 CONTRIBUTION .....	4
1.3.1 Multiprocessor Scheduling.....	4
1.3.2 Design Environment.....	5
1.3.3 Estimation.....	6
1.4 PLAN OF THESIS .....	6
<b>2 BACKGROUND .....</b>	<b>9</b>
2.1 BASIC CONCEPTS .....	9
2.1.1 Performance.....	9
2.1.2 Concurrency .....	11
2.1.3 Granularity.....	14
2.2 MULTIPROCESSOR ARCHITECTURE .....	15
2.2.1 A Taxonomy .....	15
2.2.2 MIMD Computers .....	18
2.2.3 DSP Multiprocessor Systems .....	22
2.3 PARALLEL PROGRAMMING.....	23
2.3.1 Parallelizing Compilers .....	23
2.3.2 Parallel Languages & Block Diagrams .....	24
2.3.3 Multiprocessor Design Environments for DSP .....	27
2.4 MULTIPROCESSOR SCHEDULING .....	29
2.4.1 Classification .....	30
2.4.2 Complexity Analysis .....	33
2.4.3 Basic Multiprocessor Scheduling.....	33
2.4.4 Multiprocessor Scheduling in DSP .....	37
2.5 COMPARISON .....	40
2.6 SUMMARY .....	41
<b>3 THE McDAS ENVIRONMENT.....</b>	<b>43</b>



3.1	McDAS SYSTEM OVERVIEW .....	44
3.2	SILAGE .....	47
3.3	FLOWGRAPH DEFINITION .....	51
3.3.1	Flowgraph Model .....	52
3.3.2	Flowgraph Library .....	54
3.4	ARCHITECTURE DATABASE.....	55
3.5	SUMMARY.....	57
<b>4</b>	<b>SILAGE TO FLOWGRAPH TRANSLATION .....</b>	<b>59</b>
4.1	BASIC TRANSLATION.....	59
4.1.1	Handling of Delayed Signals.....	60
4.1.2	Generation of Arrays .....	61
4.2	CDFG OPTIMIZATION .....	63
4.3	MULTIRATE APPLICATIONS.....	64
4.3.1	Introduction .....	64
4.3.2	Multirate Transformation.....	67
4.4	SUMMARY.....	72
<b>5</b>	<b>MODEL OF COMPUTATION .....</b>	<b>73</b>
5.1	A MOTIVATING EXAMPLE .....	74
5.2	COMPUTATION MODEL .....	76
5.2.1	A Multiprocessor Schedule .....	78
5.2.2	Evaluation of a Schedule .....	81
5.3	ARCHITECTURAL SUPPORT.....	82
5.3.1	Processor Characteristics.....	83
5.3.2	Multiprocessor Topology.....	83
5.3.3	Interprocessor Communication .....	84
5.4	ESTIMATING COMPUTATION TIMES & MEMORY REQUIREMENTS.....	86
5.4.1	Operator Benchmarking .....	86
5.4.2	Model Construction .....	93
5.4.3	Limitation of the Technique .....	95
5.4.4	Memory Estimation .....	96
5.5	ESTIMATING COMMUNICATION DELAYS .....	97
5.5.1	Time Slot Model.....	97
5.6	SUMMARY.....	101
<b>6</b>	<b>SCHEDULING .....</b>	<b>103</b>
6.1	PROBLEM DEFINITION .....	104

6. 1. 1	Problem Formulation.....	104
6. 1. 2	Previous Approaches.....	105
6. 1. 3	Our Scheduling Strategy.....	106
6.2	SCHEDULING FOR FIXED THROUGHPUT .....	107
6. 2. 1	Definitions .....	107
6. 2. 2	The Scheduling Appeal: Intuitive Description .....	108
6. 2. 3	Node Scheduling .....	110
6. 2. 4	Complexity Analysis .....	115
6.3	PATH MERGING .....	116
6. 3. 1	Problem Definition .....	116
6. 3. 2	Path Merging Algorithm.....	118
6. 3. 3	Complexity Analysis .....	122
6.4	RETIMING .....	123
6. 4. 1	Problem Definition .....	123
6. 4. 2	Retiming Algorithm.....	126
6. 4. 3	Complexity Analysis .....	128
6.5	NODE DECOMPOSITION.....	129
6. 5. 1	Bottleneck Node Decomposition.....	129
6. 5. 2	Critical Cycle Decomposition .....	130
6.6	SCHEDULING FOR MAXIMUM THROUGHPUT .....	131
6. 6. 1	Bounded Search Heuristic .....	131
6. 6. 2	Complexity Analysis .....	134
6.7	SUMMARY.....	134
<b>7</b>	<b>CODE GENERATION.....</b>	<b>137</b>
7.1	OVERVIEW .....	138
7.2	MEMORY MAPPING .....	140
7. 2. 1	The FIFO Communication Model .....	140
7. 2. 2	Local Synchronization.....	141
7. 2. 3	Shared Memory Implementation.....	143
7. 2. 3. 1	Centralized Shared Memory .....	143
7. 2. 3. 2	Distributed Shared Memory .....	144
7. 2. 4	Message-Passing Implementation .....	148
7.3	CODE EMISSION.....	148
7. 3. 1	Circular Buffering.....	148
7. 3. 1. 1	Interprocessor Communication .....	149
7. 3. 1. 2	State Variables.....	150
7. 3. 2	C Code Emission .....	152
7. 3. 3	Floating-Point & Fixed-Point Simulation .....	154
7. 3. 4	DSP Code Emission .....	156
7.4	SUMMARY.....	159

<b>8 SCHEDULING RESULTS</b> .....	<b>161</b>
8.1 TARGET ARCHITECTURES .....	161
8.1.1 The Sequent Symmetry Multiprocessor .....	161
8.1.2 The SMART Multiprocessor .....	163
8.2 RESULTS .....	165
8.2.1 Scheduling a Histogram Computation on the Sequent Multiprocessor.....	165
8.2.2 Scheduling a Cordic Computation on the SMART Multiprocessor.....	170
8.2.3 Scheduling Different Applications .....	174
8.2.4 Scheduling Applications with Global Recursions.....	177
8.3 SUMMARY.....	179
<b>9 CONCLUSION</b> .....	<b>181</b>
9.1 SUMMARY.....	181
9.2 FUTURE RESEARCH .....	183
9.2.1 Data Dependency Analysis.....	184
9.2.2 DSP Code Generation.....	184
9.2.3 Computation and Memory Estimation .....	185
9.2.4 Scheduling for Heterogeneous System.....	186
9.2.5 Scheduling Data-dependent Computations .....	187
9.2.6 Loop Transformations .....	188
9.3 SUMMARY.....	191
<b>10 REFERENCES</b> .....	<b>193</b>
<b>Appendix A: Flowgraph Implementation</b> .....	<b>205</b>
A.1 Flowgraph Structure .....	207
A.1.1 AFL Format .....	208
A.1.2 OCT Format.....	210
A.2 C Data Structure.....	210
A.3 A Sample AFL Flowgraph.....	213
<b>Appendix B: Silage To Flowgraph Implementation</b> .....	<b>219</b>
B.1 Silage Frontend .....	219
B.2 CDFG Generator.....	221
B.2.1 The Algorithm .....	221
B.2.2 The Data Structure.....	225
<b>Appendix C: Code Generation Results</b> .....	<b>227</b>

C.1 Histogram Silage Code .....	228
C.2 Histogram C Code .....	230
<b>Appendix D: McDAS User's Manual.....</b>	<b>243</b>
D.1 McDAS Compilation Manager .....	244
D.2 Silage To Flowgraph Translator .....	246
D.3 Scheduler for Sequent.....	248
D.4 Scheduler for SMART .....	250
D.5 Scheduler for Ideal Multiprocessor.....	252
D.6 Code Generator for Sequent .....	254
D.7 Silage Syntax .....	256

# INTRODUCTION

## 1.1 OVERVIEW

In recent years, a significant improvement in the computing power of programmable digital signal processors has been observed. New advances in architecture and technology have enable DSP processors to achieve throughputs up to 16.7 MIPS and 50 MFLOPS [Mot90]. Their high-speed performance, programmability, and low cost have already made them the ideal implementation medium in a number of real-time applications such as speech detection [Dau87] and speech encoding [Alr86]. Unfortunately, we have concurrently experienced an even greater increase in the computational requirements of DSP applications. For instance, a computation rate of 1 GFLOPS is typical for High Definition Television (HDTV) applications [Fre89]. In addition, the applications themselves are also becoming increasingly more complex, utilizing nested loop structures or multi-dimensional vector computations. Examples of this can be found in Code-Excited Linear Prediction coders [Cam90], CCITT Standard Visual Telephony [CCI89], and JPEG and MPEG image compression algorithms [Wal89]. Currently, the only means to meet the high throughput demands of these applications is with special purpose hardware, which can be quite expensive and time consuming to build at the prototyping stage.

Given the success of the DSP processor, one approach to obtaining a greater computational power while maintaining a rapid prototyping capability is to employ multiple DSP processors working in parallel. As an example, a system of 20 Motorola DSP96002 DSP's can yield a peak throughput rate of 334 MIPS and 1 GFLOPS [Mot90]. Already we can see a number of academic and industrial DSP multiprocessor projects. These will be reviewed in Section 2.1.3.

The major obstacle to the prevalent use of these multiprocessor systems, however, has been the absence of an adequate Computer Aided Design (CAD) environment to help system designers quickly design, simulate, and prototype their applications. In this thesis, we present a DSP design environment, called McDAS, which can generate efficient code for MIMD multiprocessors given a behavioral input description. The description is *architecture-independent* in that the designer does not have to tailor his specification to comply to a particular target architecture or execution scheme. The partitioning and scheduling of the application onto the processors, as well as the code generation, are completely automated. This allows the designer to devote all of his effort to designing and optimizing the application itself.

In the remainder of the chapter, the domain of DSP applications supported by McDAS is characterized, and arguments are given to justify why multiprocessors are good target implementations. A summary of the contributions of the thesis is presented, followed by an outline of the remainder chapters of the thesis.

## 1.2 APPLICATION DOMAIN

Our applications consists of medium to large-grain, synchronous digital signal processing systems. The term "medium to large-grain" indicates that the number of operations in the application is several ( $> 2$ ) orders of magnitude greater than the number of processors available. Applications which do not fit this category are termed

fine-grain. The inherent overhead in programmable processors often precludes a high throughput implementation of fine-grain applications.

The term “synchronous” means that the amount of input samples consumed by each task in the application, and the amount of output samples generated, are known at compile time and invariable at run time. This fixed execution pattern allows a multiprocessor scheduler to produce a schedule at compile time, eliminating the run time scheduling overhead. Asynchronous systems, on the other hand, allows the production or consumption of samples in tasks to depend on the value of some data. This yields an execution pattern that is unpredictable, making high quality compile-time scheduling difficult. These applications will not be addressed in this thesis.

Medium to large-grain synchronous systems cover the majority of the common signal processing applications. These include filters, digital audio, speech processing, telecommunications, robotics, sonar, radar, and image processing. A close examination of the nature of the computations involved reveals that almost all applications contain some amount of concurrency, and most contain a substantial amount. For instance, all DSP applications are executed in an infinite time loop, giving rise to *temporal concurrency* which can be exploited by pipelining (see Section 2.1). In addition, many exhibit *spatial concurrency* which is amenable to parallel execution. The concurrency may not be easily detected however, as it can exist at different levels of granularity. For spatial concurrency at a large granularity or block level, there can be parallel tasks operating on the same data. At a fine granularity level, there can be operations performed on each element of a vector or matrix in parallel. Similarly, temporal concurrency can exist at the block level, or lower, such as between iterations of a serial loop.

## 1.3 CONTRIBUTION

We are interested in an implementation medium that can be used to quickly prototype a wide range of DSP applications. Employing multiple DSP processors in parallel is an attractive option in terms of cost, design time, and performance. The low cost and programmability of a DSP processor makes it an ideal processing element, and the abundance of available concurrency in DSP applications makes leads naturally to parallel processing.

While these powerful DSP multiprocessor engines are attractive, they are seldom used in the DSP community due to the lack of software tools to support the automatic scheduling and compilation of the input program onto the multiple processors. Currently, users of these machines have to partition their applications by hand, usually without any regard for optimization. The necessary interprocessor communication and synchronization are then determined, and finally, code is handwritten for each processor. This is a painstaking and error-prone process which is also likely to be suboptimal. The contribution of this thesis, hence, is a DSP design environment to automate this process.

### 1.3.1 Multiprocessor Scheduling

A key component in the system is the multiprocessor scheduling algorithm. The goal of the algorithm is to find a mapping of tasks onto processors in such a way as to maximize the throughput of the resultant implementation. All types of concurrency execution are employed to achieve the speedup. This includes pipelining, retiming, and parallel execution. Furthermore, the scheduler can traverse the application to any level of granularity, allowing concurrency detection to take place at a granularity level suitable with the available hardware resources. For example, for multiprocessor systems with few number of processors, the concurrency exploited is at a large



granularity. For systems with a large number of processors, the concurrency is exploited at a finer granularity level to fully utilize all the available processors. By considering granularity in conjunction with concurrency, the algorithm is able to efficiently schedule a wide range of DSP applications.

A scheduling algorithm is only useful if it is able to consider constraints imposed by the target architecture, specifically, the number of available processors and the amount of available memory in each processor. The scheduler uses these parameters as bounds to prune the search space to yield only feasible schedules. In addition, interprocessor communication delays can often take a significant portion of the overall execution time, and is therefore tightly integrated in the scheduling process to yield a high quality solution. The ability to take as inputs architectural descriptions allows the scheduler to be retargetable to different multiprocessor systems.

### **1. 3. 2 Design Environment**

The McDAS environment augments the scheduler with a set of parsing and code generation tools to facilitate the prototyping process. The input textual description is parsed into a flowgraph representation, where concurrency is exposed explicitly. The flowgraph is hierarchical, structuring the application into many levels of granularity. The flowgraph serves as a central database on which all tools interact. This clean interface makes the system very modular, and allows new tools to be easily integrated. To schedule, the user only has to enter the processor count and the topology of the architecture and invoke the scheduler. Different implementations can be entered and scheduled to explore the design space. This is aided with a history mechanism for easy backtracking. Each scheduling result can be displayed with graphical tools showing processor assignment and utilization, as well as memory and bus usage. Finally, once a schedule is determined, code can be generated for each processor. The code generator supports functional simulation and real-time implementation. For simulation, C code is

emitted with bit-true or floating point data types to allow for the assessment of quantization and truncation effects. The code generator is similarly designed to be easily adaptable to different memory architectures and different processor instruction sets.

### **1.3.3 Estimation**

For the scheduling algorithm to perform well, accurate estimates of the computation and memory costs of the tasks are vital. A methodology for estimating computation times and memory requirements of operations is developed. The technique relies on benchmarking a target architecture with a set of programs to obtain execution times and memory usage of primitive operations. These values are then accumulated systematically to obtain estimates for large tasks. Results demonstrate that the technique is able to yield estimates to within 5% of the measured values.

A detail model of the interprocessor communication process has also been developed. Each communication is explicitly scheduled on the appropriate bus or buses to take into account delays due to bus congestion. This technique enables the scheduler to accurately estimate the arrival time of a data to a processor given the time of the transfer, the source processor, and the state of the routing network. This parameter is critical to the scheduler in deciding which node-processor assignment is optimal.

## **1.4 PLAN OF THESIS**

The remainder of the thesis is composed of eight chapters, organized as follows:

In Chapter 2, background material and previous work on parallel computation, multiprocessor architectures, multiprocessor compilers and scheduling theory are presented.

In Chapter 3, the McDAS DSP design environment is described in detail, and an overview of the entire compilation process is given. We will describe the input language Silage, as well as the hierarchical flowgraph format. Details are given as how certain hierarchical constructs such as function calls, loops, and conditionals are represented in the flowgraph. Finally, the parameters necessary to characterize a target architecture is presented.

In Chapter 4, we describe our front-end parser which translates a Silage textual description into a flowgraph. The organization of the program is presented, along with its features. The strategy for deriving data dependencies between operations, especially between array accesses and loop iterations, is outlined. A set of standard compiler optimizations which is incorporated into the parser is described. These include such transformations as dead-code elimination, common subexpression elimination, and manifest expression evaluation. Finally, a transformation to convert a multirate flowgraph to a single rate flowgraph via node clustering is presented.

In Chapter 5, the model of parallel computation in the McDAS environment is given. First, an example is given to motivate the model. The computation model is then presented, and a *multiprocessor schedule* is defined and interpreted. Thirdly, the architecture and the interprocessor communication model are discussed. Next, techniques for estimating computation times and memory requirements of DSP tasks are presented. We verify our estimations with actual measurements and discuss the limitation of the approach. Finally, the strategy for estimating interprocessor communication delay is presented.

In Chapter 6, we present our scheduling algorithm under two performance objectives: a) Given a fixed available sample period, determine the minimum number of processors needed, b) Given a fixed amount of processors, determine the fastest throughput implementation. The details of the node-processor assignment strategy is described, emphasizing how it can simultaneously consider pipelining and parallel

execution. The algorithm is then extended to perform retiming when flowgraph cycles are present. The granularity issue is addressed with the node decomposition flowgraph transformation. We show how node decomposition combined with pipelining and parallelism allow our scheduler to exploit block level parallelism, data parallelism, block level pipelining, and loop pipelining, all in a unified manner. The results of the scheduling are shown for a wide set of examples.

In Chapter 7, the code generation strategy is introduced. The first phase is the memory mapper phase, which allocates buffers for interprocessor communication and determines local and global synchronizations. We discuss how different memory architectures affect the interprocessor communication strategy. The organization of the code emitter is then described, detailing our implementation of buffers, I/O, and sample delays. Finally, a mechanism for performing fixed-point and floating-point simulations from the same C code is presented.

In Chapter 8, we describe two multiprocessor systems which have been targeted by McDAS. The first system is the Sequent system, a shared-bus multiprocessor machine composed of 14 PE's. The second system is the SMART system, a configurable bus machine composed of 10 PE's. Results obtained from scheduling different applications on both architectures are analyzed and compared.

Finally, in Chapter 9, we conclude the thesis, and point out directions for future research.

# BACKGROUND

In this chapter, the previous work in multiprocessor architecture, parallel languages and compilers, and multiprocessor scheduling theory is presented and analyzed. In Section 2.1, a number of key concepts and terminologies in parallel computing is introduced. In Section 2.2, a classification of computers is presented and their target applications described. The section ends with a detail look at multiprocessor systems for DSP applications. In Section 2.3, the software aspects of multiprocessing is treated, including parallel languages and compiler systems. Again, the section ends with a discussion on multiprocessor software systems for DSP. Section 2.4 reviews the research efforts in the area of multiprocessor scheduling, concentrating on those useful for DSP. Finally, the McDAS environment and its scheduling algorithm are compared with other systems in Section 2.5.

## 2.1 BASIC CONCEPTS

### 2.1.1 Performance

In parallel computation, multiple processing units are employed to achieve a higher computational performance over a single processor. The performance gain can be a reduction in *latency*, defined as the time elapsed between the arrival of an input

sample, and the availability of the corresponding output, or an increase in *throughput*, defined as the rate at which the system can process incoming input data.

The performance gain is measured by the speedup over a single processor implementation. There are two types of speedup: *Execution speedup* or *Latency speedup*, and *throughput speedup*. Execution speedup on  $n$  processors is given as the execution time of an application on one processor over the execution time of the same application on  $n$  processors. Throughput speedup is given as the increase in the rate at which a system can process incoming data. The maximum achievable execution or throughput speedup for  $n$  processors is  $n$ .

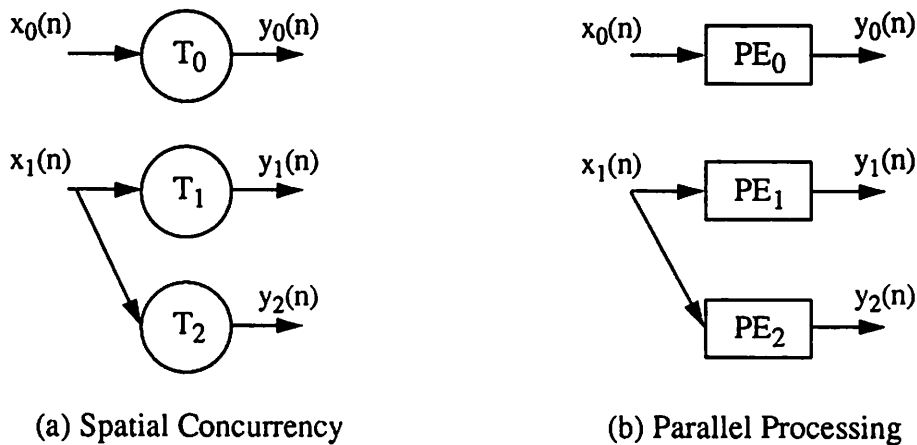
Two criteria which can indirectly affect the speedup of an implementation are the *load balancing* and the *interprocessor communication overhead*. Load balancing tells how even the distribution of work is across the processors. When the load is unbalanced, lightly loaded processors sit idle waiting for the heavily loaded processors to finish. The lightly loaded processors are inefficiently used, resulting in poor speedup. Other measures equivalent to load balancing are maximizing processor utilization and minimizing idle time.

Interprocessor communication results when processors need to exchange data. The time devoted to these transfers can be substantial, especially in applications that are communication intensive. To minimize communication overhead, tasks that communicate heavily are put in the same processor, eliminating the communication. Note that this goal tries to cluster tasks together, while load balancing attempts to disperse tasks to different processors. A good schedule must consider both criteria to be effective. Another overhead which can arise is the *synchronization overhead*. Processors usually need to synchronize when they communicate, and thus minimizing communication also indirectly minimizes synchronization.

In this thesis, the performance speedup will be the main criteria for evaluating the quality of a multiprocessor implementation.

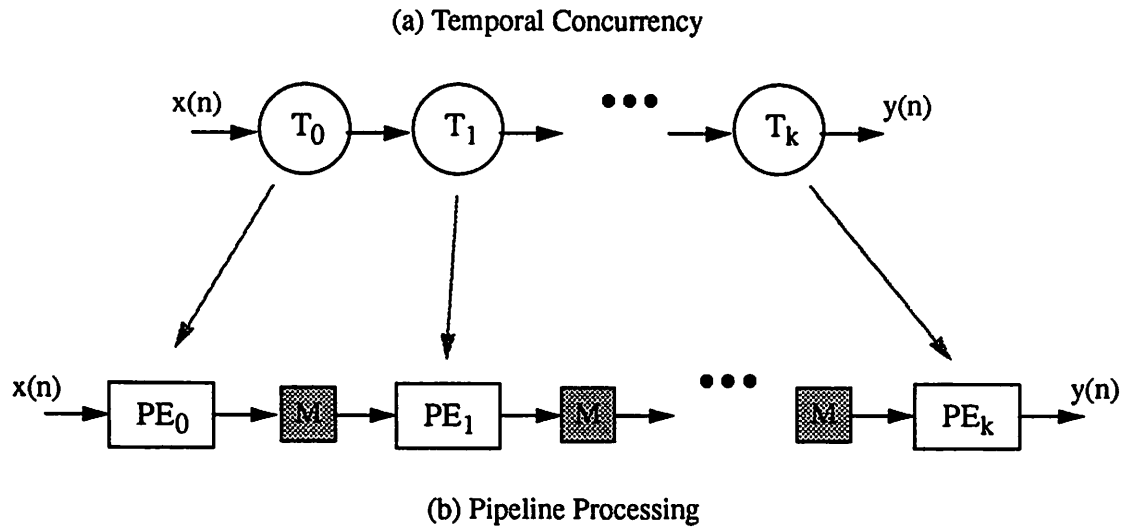
## 2.1.2 Concurrency

The performance gain is obtained by exploiting the concurrency available in the application. There are two types of concurrency available, each giving rise to a corresponding concurrent processing methodology. There is *spatial concurrency (parallelism)* (Figure 2.1a), where there are tasks which can be executed by several processors simultaneously without affecting the resultant output. This methodology is called parallel processing (Figure 2.1b). There is *temporal concurrency* (Figure 2.2a), where there is a chain of tasks which is embodied in an infinite time loop. Concurrent processing of these tasks involves dividing the chain into stages, with every stage handling results obtained from the previous stage. This methodology is called pipelining or pipelined processing, and is illustrated in Figure 2.2b. Pipeline processing



**FIGURE 2.1 : Parallelism and Parallel Processing**

is possible in DSP applications due to the inherent nature of signal processing to repeat the same computation to each sample of the input stream. Note that for pipelining to work, buffer memories must be inserted between the stages to store intermediate values.

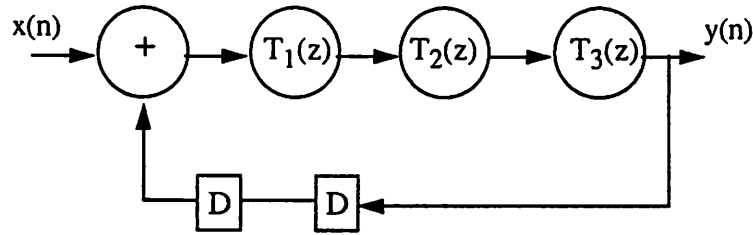


**FIGURE 2.2 : Pipeline Concurrency and Pipeline Processing**

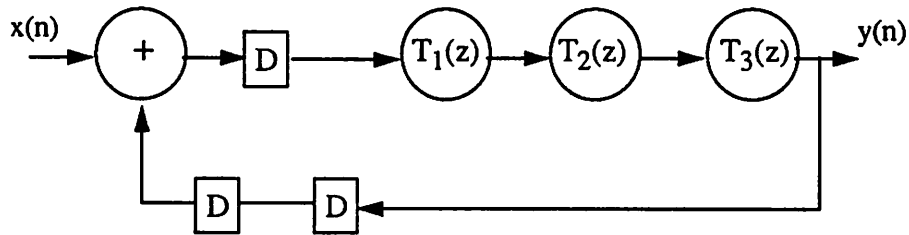
Algorithmically, each pipelining operation corresponds to an insertion of a sample delay into the computation. Thus, in a computation pipelined into  $k$  stages, each output sample  $y(n)$  corresponds to the resultant output of the input sample  $x(n-k)$ . In the  $Z$  domain, the output is  $Y(z) = T_k(z) \cdots T_1(z) z^{-k} X(z)$ . The  $z^{-k}$  latency is called *sample latency*. In general, exploiting parallelism reduces latency, and exploiting pipelining increases the throughput. The simultaneous application of both parallel and pipeline processing can significantly improve both the latency and the throughput of a system.

Some applications may possess *feedback* or *recursion*, that is, when the processing of a sample is dependent of the resultant processing of a previous sample. This shows up in a flow graph as a cycle with one or more sample delays, as shown in Figure 2.3(a). Inside a cycle, pipelining can alter the functionality of the algorithm, which is undesirable. Let  $T(z) = T_3(z)T_2(z)T_1(z)$ . In Figure 2.3(a), the  $Z$  domain output is  $Y(z) = T(z) [z^{-2} Y(z) + X(z)]$ . The output for Figure 2.3(b) is  $Y(z) = T(z) [z^{-3} Y(z) + z^{-1} X(z)]$ , which is different from the output in Figure 2.3(a). One technique which allows us to effectively pipeline the cycle while maintaining the correct functionality is the *retiming* transformation. Retiming involves the rearranging of delays within cycles to achieve better performance. It has been used extensively in optimizing circuit

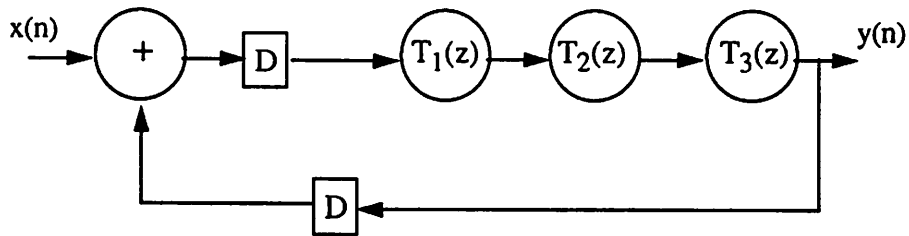




(a) Flow graph with feedback



(b) Delay insertion changes the functionality



(c) Retiming retains same algorithm

**FIGURE 2.3 : Retiming**

performance [Lei83], but is of great value in signal processing as well. Figure 2.3(c) shows a retiming of the flow graph. The output is  $Y(z) = T(z) [z^{-1} X(z) + z^{-2} Y(z)]$ . This is the same computation as in Figure 2.3(a) if the input stream is delayed by a sample. The use of pipelining, parallelism, and retiming exploits all types of concurrency available in DSP applications.

### 2. 1. 3 Granularity

The concurrency in DSP applications can exist at all levels of granularity, from the largest block level granularity to the finest data level granularity. At the block level, spatial concurrency gives rise to *block level parallelism*, where there are independent tasks simultaneously processing their input data. At the data level, spatial concurrency takes the form of *data parallelism* or *data partitioning*, where identical operations are applied to each member of the vector or a matrix. Temporal concurrency at the block level is amenable to *block level pipelining*, while at a lower loop level granularity, *loop pipelining* or *loop winding*[Gir87] can be employed. The ability to exploit different types of concurrency at different granularity levels allows the consideration of all of the above techniques in a unified manner. It should be noted that combining data parallelism with retiming can improve the throughput of recursive applications. This will be expanded further in later sections.

The optimal granularity level exploited by a scheduler should be dictated by the number of processors present and the communication overhead. The greater the number of processors available, the smaller the granularity level should be. This guarantees that enough concurrency is exhibited to be used by all the processors. Beyond a certain granularity however, the large number of operations present may significantly slow down the scheduler. Furthermore, the interprocessor communication cost begins to play a dominant role in the overall execution time. In particular, it may take less time to execute two nodes in one processor than to spread them to two processors and suffer the communication overhead. This *saturation effect* has been observed by a number of researchers [Chu80] [Kri87]. In [Sar89] Sarkar attempts to capture this trade-off by assigning a cost to a granularity level equal to the maximum of the flow graph critical path and the flow graph communication overhead. If the granularity is too fine, the overhead term will be large, causing the cost to be large. If the granularity is too coarse, the critical path term will be large, also causing the cost to

be large. The cost will be minimum at an optimal intermediate granularity. The overhead term is calculated as the sum of the scheduling overhead of a node, over all nodes in the flow graph, plus the input and output communication overhead. A similar technique was used by McCreary and Gill [McC89]. They first cluster nodes into a hierarchy of *clans*, defined as a group of nodes with the same input and output nodes. Traversing the clan hierarchy bottom up, they determine whether it is cheaper to execute the clan in one processor or not. If it is, the nodes in the clan are clustered. The cost function is the execution time of the nodes, as well as the input and output communications.

These work are significant as they recognize the importance of granularity in multiprocessor scheduling. However, their approach separates the granularity determination and the scheduling itself. As a result, their modelling of the communication costs is done without knowing in detail the activities on the bus. We believe that granularity determination is best done in conjunction with the scheduling process, in an iterative manner. In this way, the scheduler can decide to change the granularity of the flow graph based on previous scheduling results.

## **2.2 MULTIPROCESSOR ARCHITECTURE**

### **2.2.1 A Taxonomy**

There has been a rapid growth in the number of proposed and constructed architectures over the past 10 years. Flynn [Fly72] originally proposed a taxonomy to organize computers based on their processing of instructions and data. There are four classifications:

1. SISD - Single Instruction, Single Data
2. SIMD - Single Instruction, Multiple Data

3. MISD - Multiple Instruction, Single Data
4. MIMD - Multiple Instruction, Multiple Data

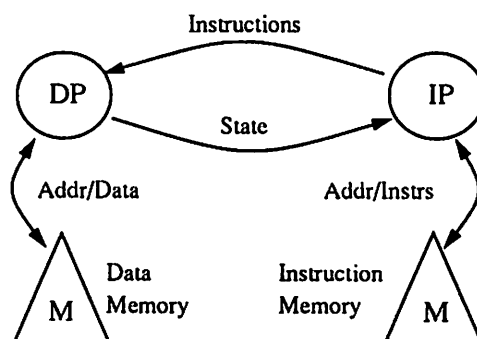
SISD architectures represent the standard von Neumann organization of most uniprocessor-based computers today. SIMD architectures incorporate an array of processing elements all executing the same instructions from a central controller. These are used to process elements of arrays in parallel. The SIMD technique can be found in vector or array processors of today's supercomputers [Nag84][Che84]. A number of SIMD computers have also been built to tackle problems with a massive amount of parallelism such as image and video processing, computer graphics, database, and simulations. Perhaps the most well known SIMD computer is the Connection Machine from Thinking Machines Inc.[Thi87], which can be configured with 65, 536 processors. Execution on MISD architectures would involve having multiple simultaneous instructions on the same piece of data. Pipeline machines are often mentioned to be of this style. The last category describes MIMD machines, which we commonly refer to as multiprocessor computers. These computers contain processors, each with its own independent controller. This enables the processors to execute different instructions on different data.

Recently, Skillicorn [Ski88] presented an extension to Flynn's taxonomy to classify the growing number of multiprocessor architectures more discriminantly. It is a two-level hierarchy in which the upper level classifies architectures based on the number of processors for data and for instructions, and the interconnection between them. A lower level can be used to distinguish variants even more precisely; it is based on a state machine view of the processors. In the taxonomy, there are four types of functional units from which any abstract machine can be constructed. These are:

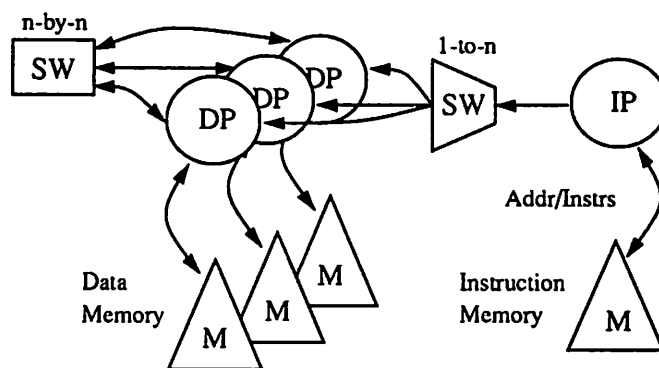
1. IP - An instruction processor to process instructions
2. DP - A data processor to process data
3. M - A memory hierarchy to store data
4. SW - A switch to provide connectivity between other functional units

There are three different forms of switches. A 1-to-n switch connects 1 unit to n devices of another set of units. An n-to-n switch connects the  $i$ th unit of one set of functional units to the  $i$ th unit of another. This is equivalent to a 1-to-1 switch replicated  $n$  times. An n-by-n switch connects any device of one set of functional units to any other device of a second set.

With these definitions, a large number of architectures can be described precisely. Figure 2.4a shows the arrangement of a uniprocessor machine, while Figure 2.4b shows the abstract machine for an SIMD machine.



(a) Von Neumann SISD architecture



(b) SIMD architecture

**FIGURE 2.4 : Skillicorn's Taxonomy for Computer Architectures**

Skillicorn's classification allows for a much more detail discrimination of the large number of MIMD architectures that are available. These range from the familiar

shared memory and message passing computers to the exotic graph reduction [Dar81] and dataflow [Gur80] machines. In this thesis, we will be concern only with the first two architectures.

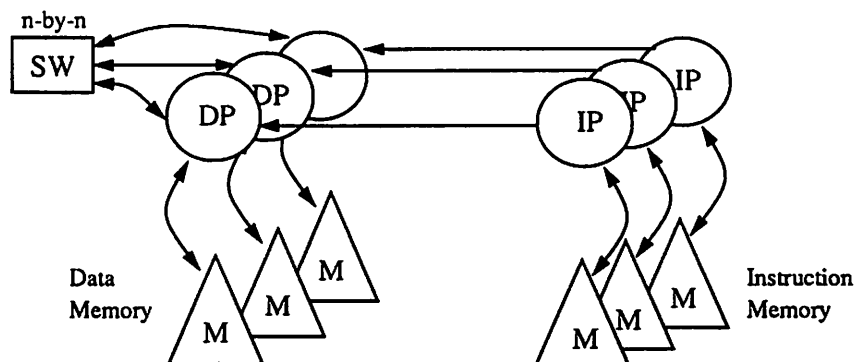
## **2. 2. 2 MIMD Computers**

The ability for each processor of an MIMD machine to autonomously work on a separate part of the problem makes MIMD machines much more general than SIMD machines. Unfortunately, the generality comes with a cost. Cooperating to solve a problem usually requires a tight interaction among the processors. SIMD machines achieve this via the common instruction flow. Since MIMD processors are independently controlled, communications and synchronizations among the processors are necessary. Extra time and resources must be reserved for these operations. The more processors that are used, the more the overhead incurred. Beyond a certain point, the addition of more processors may even decrease the performance due to the excessive overhead. This phenomenon is the saturation effect alluded to earlier. Since multiprocessing makes sense only when the speedup achieved outweighs the overhead paid in communications, MIMD machines tend to exploit parallelism at a coarser granularity than SIMD machines. As a result, the number of processors in MIMD systems (2-100) is usually much less than SIMD systems (100-100,000). With fewer processors, each core processor can afford to be much more powerful. There are two common methods for processors in MIMD computers to exchange data, via message passing and through shared memory. This gives rise to two classifications for MIMD computers: Multicomputers and multiprocessors.

### **2. 2. 2. 1 Multicomputers**

A multicomputer system consists of a number of processors that communicate asynchronously by sending and receiving messages across a network. There are no global or shared system resources. Each processor has its own private address space, its

own local memory and hardware support to transmitting and receiving messages. Using Skillicorn's classification, the multicomputer has a structure as shown in Figure 2.5.

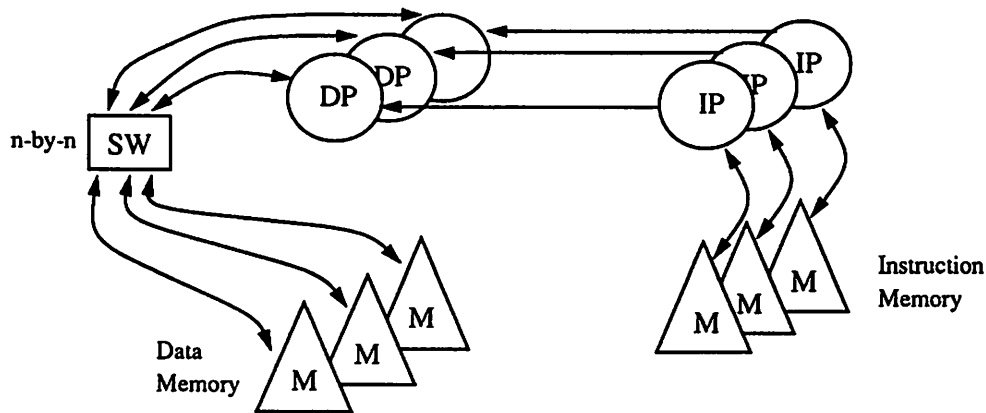


**FIGURE 2.5 : An Abstract machine for a multicomputer**

The send and receive constructs not only perform the communication, they also synchronize the data transfer to ensure the receive process only starts after the send process has completed. The processor interconnection can range from a set of connected buses to an elaborate nationwide network. Messages may be broken up into packets and routed through the network to the destination processor. The most well-known multicomputer is the Intel iPSC system [Int86]. In this system, processors are connected in a hypercube topology. A hypercube of dimension  $d$  has  $2^d$  processors, each one in direct link with  $d$  neighbors. Other topologies include the ring, tree, and mesh. Multicomputers are attractive because their interconnection and communication strategy can handle a large number of physically distributed processors, enabling a large user group and fault-tolerance. However, the cost of constructing and routing messages at run-time can be substantial. Multicomputers are also known as *loosely-coupled systems* or *distributed systems*.

### 2. 2. 2. 2 Multiprocessors

A multiprocessor system consists of a number of processors which communicate through shared resources. The programmer sees a single address space that all processors can access. To communicate, the sender writes the data to a specific memory location and the receiver reads the data from the location. This process must be done carefully to ensure correct data transfer. Firstly, the sender and the receiver must agree at compile-time what memory location will be used to implement the writing and reading. Secondly, they must synchronize so that the receiver processor only reads the data after the sender has completed the write. Multiprocessors are attractive because of their similarity to the uniprocessor from the programmer's point of view. Multiprocessors are also called *tightly-coupled systems* to reflect their resource sharing. An abstract multiprocessor machine is shown in Figure 2.6.



**FIGURE 2.6 : An Abstract machine for a multiprocessor**

Multiprocessors can be further divided into two groups depending on whether the shared memory is centralized or distributed. In a centralized memory multiprocessor system, the single address space is also realized by a single memory space. If many processors shared the same data, only one copy of the data needs to be present in memory. The biggest drawback of a centralized memory is the memory access contention by the processors. Reads and writes to the same location must be queued,



slowing down the system's performance. To alleviate this problem, caching can be used to reduce accesses to the central memory. However, the presence of caches in a multiprocessor system introduces the problem of *cache coherence*, which is, how to make sure a read always delivers the most recent value. Various techniques to solve this problem can be found in [Arc86][Rav83][Pap84]. One example of a centralized memory multiprocessor system is the Symmetry computer from Sequent Computer Systems [Ost86]. The Symmetry system has up to 30 Intel 30386 processors, connected by a single shared 64-bit wide bus. Each processor has a floating point co-processor and a 64KB write back cache. The memory system can have up to 6 controllers with up to 240MB total main memory.

Extending the caching idea one more step, we get the distributed memory multiprocessor system. Here, the single address space is divided into sections, with each section assigned to a different processor. The shared memory is distributed so that the memory implementing a processor's section is physically located next to the processor. This locally resident memory effectively behaves as the processor's cache in that reading from this memory does not access any shared resources. However, since the memory is shared memory, the source processor can directly write data into the destination processor's section of memory. Thus, one possible distributed memory communication scheme is to have all writes go through the network, and have all reads be local. This *global-write-local-read* communication scheme can dramatically reduce the number of network accesses. The trade-off is that for broadcast data, all destination processors must have its own copy of the data in its section of memory. Figure 2.7 shows the global-write-local-read scheme on a distributed memory multiprocessor. This scheme can be found the distributed memory architecture of the SMART multiprocessor from U.C. Berkeley [Koh89].

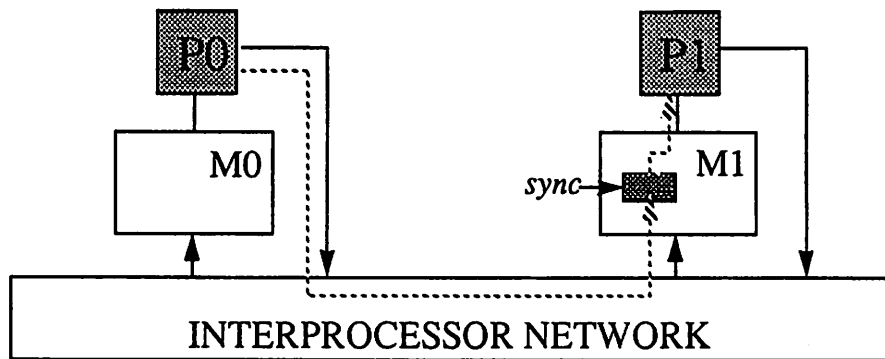


FIGURE 2.7 : Global-write-local-read Communication Scheme

### 2. 2. 3 DSP Multiprocessor Systems

DSP Multiprocessors are multiprocessors constructed from DSP processors. These specialized processors are designed to maximize throughput in data intensive real-time applications. As such, they boast features like two or more concurrent memory accesses in one cycle, a floating point multiply-accumulate operation in one cycle, and zero-overhead hardware looping[Att88]. To maintain the high throughput support, DSP multiprocessors are designed to allow efficient interprocessor communication. Over the years, many DSP multiprocessor systems have been proposed or built. A few will be described here as a representative sample. One such system is the DADO2 DSP Multiprocessor system developed at AT&T Bell Laboratories for performing speech recognition [Sto85]. The DADO2 is composed of 15 processing elements (PE) connected in a binary tree topology. Each PE is comprised of an Intel 8751 processor, an AT&T DSP32 signal processor, and 64 Kbytes of local memory. The machine is used to implement various Dynamic Time Warping algorithms [Sak78] to perform the pattern matching necessary in speech recognition. Another system is the family of DSP multiprocessors designed at Georgia Tech [Bar91]. The first generation DSMP-I is composed of 16 AM29325 processors connected in a 2-dimensional grid. Each processor is capable of 5MFLOPS. The DSMP-II consists of 2 to 16 AT&T DSP32 processors connected in a ring. Each processor has a peak rate of 8MFLOPS. The DSMP-III currently under design would contain 16 to 128 TI TMS320C40 processors

connected in a 3-dimensional grid. Each C40 processor is capable of 40 MFLOPS and has 6 communication ports. Lee and Bier at UC Berkeley proposed a multiprocessor architecture MOMA, for Maintains Ordered Memory Accesses [Lee90]. The special feature of this architecture is a central controller, which grants access to the shared memory bus in a prespecified, compiler-determined order. This guarantees synchronization, since whenever two processors must synchronize on a shared location, the writer is always granted control before the reader. One high performance multiprocessor machine which finds application in DSP is the WARP systolic machine from Carnegie Mellon University [Ann87]. It is also a linear array of 10 processor cell, each equipped with a large local data memory and capable executing at 10MFLOPS. The cells are connected in a ring architecture where each cell can transfer up to 80 Mbytes/sec to its neighbor cell. The large memory and I/O bandwidth enables the WARP machine to efficient support fine-grain data-parallelism.

## **2.3 PARALLEL PROGRAMMING**

While programming a uniprocessor machine has become standard practice, programming a multiprocessor system is still new and complex. The most efficient algorithms on serial machines may no longer be the most efficient for parallel machines. The programmer must now decompose the algorithm into parallel parts, map these onto the processors, layout the memory organization, and schedule the interprocessor communications. Even worse, these parallel programs often must undergo a major change or even be rewritten from scratch in order to run on a different multiprocessor.

### **2.3.1 Parallelizing Compilers**

To alleviate this burden, attempts have been made to design compilers that map existing sequential programs onto vector computers or multiprocessors. Most

supercomputer manufacturers such as Cray[Che84], Hitachi[Nag84] provide such compilers. In addition, there are a number of experimental work from third-party vendors and universities[Kuc84][Dav86][All87]. Perhaps the most notable work is the University of Illinois's Parafrese [Kuc84] and Stanford's SUIF projects [May91], which detects parallelism from Fortran programs and generates parallel code. This is done as a 3-phase process. The first phase involves an extensive data dependency analysis of the program to build a data dependency graph. The second phase applies various optimizations which can exploit the architecture. For computers with vector instruction sets, vector optimization or *vectorization* is appropriate. For multiprocessors, concurrent constructs are derived. Some optimizations available are scalar expansion, loop interchanging, fission by name, loop fusion, loop collapsing, and strip mining [Pad86]. Finally, the last phase generates the parallel code.

The hardest phase of the process is the data dependence detection, which translates into an integer programming problem [May91]. Since there are no efficient algorithm known for solving integer programming problems [Kan87], many compilers assume a data dependence whenever the dependence cannot be determined in a reasonable time. For special case inputs, efficient algorithms are available to determine exact dependence [Li90][May91].

- While the performance of these compilers are improving, they rest solely on their ability to extract parallelism from the program, which can be severely limited by the semantics of the language used and the possible unnecessary sequential organization of the code.

### **2.3.2 Parallel Languages & Block Diagrams**

Most programs for uniprocessors were written using Fortran, C, Pascal, or other conventional imperative languages. These languages were designed to let programmers manipulate data stored in the memory of a von Neumann computer. This

programming style translates a computational task, which may contain a large amount of concurrency, into a series of sequential memory fetches, stores and arithmetic operations. This artificial sequential ordering, termed the “von Neumann bottleneck” by Backus[Bac78], makes these languages unattractive for parallel programming. In addition, the close interaction of the language to the underlying memory storage allows routines to modify memory in very subtle ways. The use of “call-by-reference” allows a routine to modify parameters in the calling program. Also, most languages allow global variables that can be modified by any routine at any time. These side effects may inadvertently establish data dependencies between routines which were not intended.

Applicative (also called functional or data flow) languages have been proposed as a paradigm for parallel programming. It is an attempt to capture the flavor of a graphical flow graph in a linear textual form. In data flow, a program is interpreted as a flowgraph, with nodes and edges. Nodes represent instances of functions and arcs represent the flow of data between nodes. Data flow languages sequence program actions by a simple data availability firing rule: When a node’s inputs (arguments) are available, the function associated with the node can be fired, i.e. applied to the arguments. After firing, the results are available at the node’s outputs, and these may allow other nodes to fire. Thus, a node’s outputs are only dependent on the node’s inputs. There is no side effect to obscure data dependency. The main advantage resulting from this is that nodes which are not connected by a directed path are independent and can be executed in parallel. To enforce the data flow semantic, these languages possess properties such as a single assignment convention, a pass-by-value function call, an absence of global variables, a lack of history sensitivity, and others[Ack82]. Two examples of data flow languages include Val from MIT[Ack79] and Id from UC Irvine[Arv78]. These were developed in conjunction with the data flow computer projects at the respective universities. Silage[Hil85] is an applicative language developed especially for specifying DSP systems. It is the input language for the McDAS environment and will be described in more detail in a later chapter.

LUSTRE [Roc91] is a synchronous data flow language used to describe hardware controlled by a global clock. It has been used for high level synthesis in conjunction with the ESTEREL language [Ber91] used to describe controllers. Sprite [Kro92] combines applicative and functional constructs of applicative languages with operational constructs of sequential languages for high level synthesis. For real-time programming, the SIGNAL functional language [Gue91] is based on synchronous multiple-clocked flows of data and events. Still others such as Lucid[Ash77] and FP[Bac78] were designed to possess certain mathematical properties of functional application which make them amenable to program verification.

Instead of working on textual languages which represent flow graphs, some designers have chosen to work on the signal flow graphs themselves. Flow graphs or block diagrams have been used by engineers for years to represent their systems, whether it be in VLSI, parallel computation, DSP, or robotics. Recently, flow graphs are finding use as a tool for specification of computations, especially in the digital signal processing and VLSI design domain. In such a graphical design system, the user constructs a design by connecting functional blocks together into a signal flow graph, using a graphical schematic editor. The blocks can have arbitrary functionality and may be defined in a standard library or by the user. Each block has associated with it code that implements the necessary processing functions. In the field of VLSI design, these graphical design systems are abundant. Examples can be found in the OCT environment at UC Berkeley[Har86], the System Architect's Workbench at Carnegie Mellon University[Wal87], and commercial tools from CAD vendors like Synopsys, Valid, Xilinx, etc.

It is very interesting to note that there are many models of computations which can be expressed with the same block diagram representation. These include dataflow, time-driven, control/data flow, and hybrid dynamical systems, all with very different execution semantics[Lee89a]. A block-diagram description is thus only complete when

a model of computation is included. Furthermore, in constructing the flow graphs, subtle semantic inconsistencies between parts of the flow graph can be inadvertently created such as nodes with different rate inputs for example. In [Lee89b], Lee shows how these inconsistencies can lead to deadlocks and unbounded memory requirements. In this thesis, we are primarily interested in a special case of data flow, called *synchronous data flow* (SDF)[Lee87b]. In synchronous data flow, the number of data samples produced and consumed by every node on each firing is known at compile time. For signal processing, most applications fall into this category. Knowing the data rates permits the scheduling of synchronous data flow graphs onto multiprocessors to occur at compile time, eliminating the run-time scheduling overhead. SDF also allows inconsistencies to be found at compile time[Lee89b].

Many block diagram based DSP design systems for uniprocessors have been developed. These include Blosim[Mes84], BOSS[Sha87], and Gospl[Cov87]. All of these systems contain a library of common DSP blocks such as filters, FFT, equalizers, decimators, etc. as well as arithmetic blocks such as adders, magnitude, log, etc. Associated with each block are its simulation code and real-time code. The simulation code is usually implemented in a high level language with emphasis on portability and user interface, whereas real-time code is more concerned with exploiting hardware, efficiency and throughput. Program construction is done by concatenating the code associated with each block. Due to the explicit concurrency exposed in flow graph descriptions, the extension to multiprocessor implementation was inevitable. In the next section, we concentrate on design systems for multiprocessor digital signal processing.

### **2. 3. 3 Multiprocessor Design Environments for DSP**

In this section, we discuss some multiprocessor design environments for DSP users. By design environment, we mean a software system which aid the user in

developing real-time DSP applications, from algorithmic design and simulation to implementation on real-time hardware.

Most multiprocessor DSP design environments currently available are block-diagram based. Examples of these systems include the Gabriel system from UC Berkeley[Lee89c], the Block Diagram Compiler (BDC) from Lincoln Labs[Zis87], the ZC compiler from Carnegie Mellon[Pri91], the cyclo-static scheduler from Georgia Institute of Technology[Sch85], and the Signal Processing WorkSystem-MultiProx (SPW-MP) system from Comdisco Systems, Inc. The block diagram construction processor is similar to uniprocessor design systems. After the block-diagram is created, the application is scheduled onto the target multiprocessor using a multiprocessor scheduling algorithm. Once a schedule is found, code is generated for each processor based on the processor assignments. Extra code is inserted to perform interprocessor communication and synchronization. Each program can then be downloaded to its corresponding processor to be executed.

There are a few multiprocessor DSP design environments, including McDAS, which allow users to specify their descriptions using a DSP applicative language. Textual descriptions are not as illustrative as a graphical interface, but allows for more flexibility and cleaner specification of hierarchical structures such as iterations, recursion, etc. They also prohibit inconsistencies through well-defined syntax. These systems will have to initially translate the textual program into a flow graph description to expose concurrency. After the scheduling is done, code is generated using standard compiler's code generation techniques. In the past, the code synthesized from block-diagram systems were considered superior to these compiler generated code due to their hand-optimized libraries. With the recent advances in DSP compilers technology however, the advantage is rapidly diminishing [Tex92][Gen89]. Furthermore, library blocks with pre-defined code cannot be split into smaller tasks, and must be considered by the scheduler as atomic entities. This prevents the scheduler from exploiting



concurrency inside these blocks, restricting the smallest exploitable granularity level to the size of the largest block. In flow graphs built from textual descriptions however, there is no such artificial granularity boundary. The scheduler is free to exploit whatever granularity level it desires. Finally, many block diagram systems describe their library blocks with sequential code such as C or Fortran. This leads to a discrepancy between the data flow semantical model at the top level and the control flow semantical model of the blocks. While it is possible to define an interface semantic policy, it is implementation dependent and not portable. A completely textual description ensures the same semantical model at all levels of hierarchy.

The major difference among all of these systems, whether graphical or textual, lie in the multiprocessor scheduling strategy. In the next section, we review the current approaches.

## 2.4 MULTIPROCESSOR SCHEDULING

Multiprocessor scheduling consists of assigning tasks to processors, specifying the order in which the tasks are executed on each processor, and specifying the time at which they begin execution. There is a myriad of scheduling algorithms, with all different kinds of assumptions and approaches. The basic problem can be stated as follows. Consider a set of  $P$  processors, and a directed graph  $G = (N, E)$ .  $N = \{n_1, n_2, \dots, n_k\}$  are the nodes of  $G$ , and  $E = \{e_1, e_2, \dots, e_m\}$  are the edges of  $G$ . The nodes represent computational tasks, and the edges define a relation on these tasks.  $n_i \rightarrow n_j$  implies there is an arc from  $n_i$  to  $n_j$ , and that the task  $n_i$  must finish before  $n_j$  can begin. Associated with each task  $n_i$  is a execution time, or node weight  $w_i$ . Beyond this problem definition, different assumptions are made and different goals are pursued, leading to a number of different classification of scheduling techniques.

## 2.4.1 Classification

We discuss here the numerous approaches to classifying multiprocessor scheduling techniques. This is not meant to be exhaustive, but is meant to provide a better appreciation of the different scheduling applications.

### 2.4.1.1 Preemptive vs. Non-preemptive

In non-preemptive schedulers, once a processor is allocated to a task, it executes the task to completion. Preemptive schedulers allow a processor to halt execution of a task to begin processing a new task. The interrupted task is continued at a later time, either by the original processor or a different one. Preemption requires run-time task switching but may lead to better load balancing. Non-preemption has no run-time overhead, but performance can be affected by the task size.

### 2.4.1.2 Dynamic vs. Static

A scheduling taxonomy was introduced by Lee[Lee89d] to classify schedulers according to whether three components of the scheduling task, being processor assignment, order assignment, and time assignment, are to be performed at run-time or at compile time. Figure 2.8 shows the properties of the four classes. A Fully dynamic

	Node-Assignment	Node-Ordering	Node-Timing
fully dynamic	run-time	run-time	run-time
static-assignment	compile-time	run-time	run-time
self-timed	compile-time	compile-time	run-time
fully static	compile-time	compile-time	compile-time

**FIGURE 2.8 : Scheduling taxonomy by Lee.**

---

scheduler performs all operations at run-time[Bok81]. When all inputs to a node are available, the node is assigned to an idle processor, which executes it. A static assignment scheduler determines which node is assigned which processor at compile-time, usually base on interprocessor communication costs[Sto77][Bok81]. The exact order and timing is determined at run-time depending on which input data to which node is available. A fully static scheduler determines everything at compile-time, so there is no run-time scheduling and synchronization overhead. However, in order to determine the exact time to execute nodes, the exact execution time of each node must be known. Since this is rarely possible in a real life environment, we can decide the order of the node execution of each processor at compile time, and let the processor execute the nodes when possible. This is the approach of a self-timed scheduler. Self-timed schedulers allow the execution time of nodes to be non-exact and uses synchronization to ensure correct execution. Lee concluded that for signal processing, self-timed scheduling is the most attractive, although any fully static scheduling algorithm can be converted to a self-timed algorithm if synchronization costs are addressed.

### **2. 4. 1. 3 Single-execution vs. Iterated-execution**

Most applications outside of signal processing assume that the application is to be executed once on some input set. For these, only spatial concurrency is available, and the standard scheduling objective is to minimize the execution time of the application. In signal processing, applications are executed indefinitely on a stream of input samples. Hence, both spatial and temporal concurrency are available. If we repeat the minimum execution time schedule for each sample, we obtain a minimum latency implementation. More often however, the goal is to maximize the system throughput, with latency being a secondary objective. The approach above does not exploit the available temporal concurrency to increase throughput through pipelining. With both

concurrency, we can employ both pipelining and parallel processing to achieve the throughput speedup.

#### **2. 4. 1. 4 Variable Granularity vs. Fixed Granularity**

Most scheduling algorithms to date assume the granularity of the given flow graph is fixed. They do not know about and therefore will not exploit any potential concurrency that may exist inside the nodes in the flow graph. In the case where the granularity is too coarse for the given number of processors, there will not be enough concurrency to fill all the processors, resulting in poor processor utilization. Schedulers which can accept hierarchical flow graphs and can traverse different granularity levels do not run into this difficulty.

#### **2. 4. 1. 5 Others**

There are still many other criteria which separate scheduling algorithms from one another. One criteria is whether or not communication overhead is taken into account. Most earlier scheduling algorithms do not consider communication overhead, yielding results which are often unusable. With the types of communication intensive applications that can exist, it is no longer feasible to ignore this important criteria. Another classification is whether the scheduler assumes a finite resource limit or an infinite one. Some scheduling techniques use as many processors as necessary to achieve their goals. Others have no consideration for memory usage limits. Even if these techniques are good, they would need to be modified to be practical. Finally, other key points of a scheduling algorithm is whether it can accept different types of processors in the system, whether it can efficiently handle special types of tasks such as data-dependent loops and conditionals, whether it can configure the architecture topology as it schedules, and so on.

In our application of real-time signal processing, the performance constraint forbids any excessive run-time overhead. As a result, preemptive and dynamic

scheduling must be ruled out. The scheduling algorithms that we will discuss in the remainder of the section are all non-preemptive and static (fully static or self-timed) algorithms. In order to perform the scheduling at compile-time, we must model the events which will take place at run-time, and make scheduling decisions based on them. As a result, static algorithms are much more complex than their dynamic counterparts. A measure of algorithm complexity is briefly introduced in the next section.

## 2.4.2 Complexity Analysis

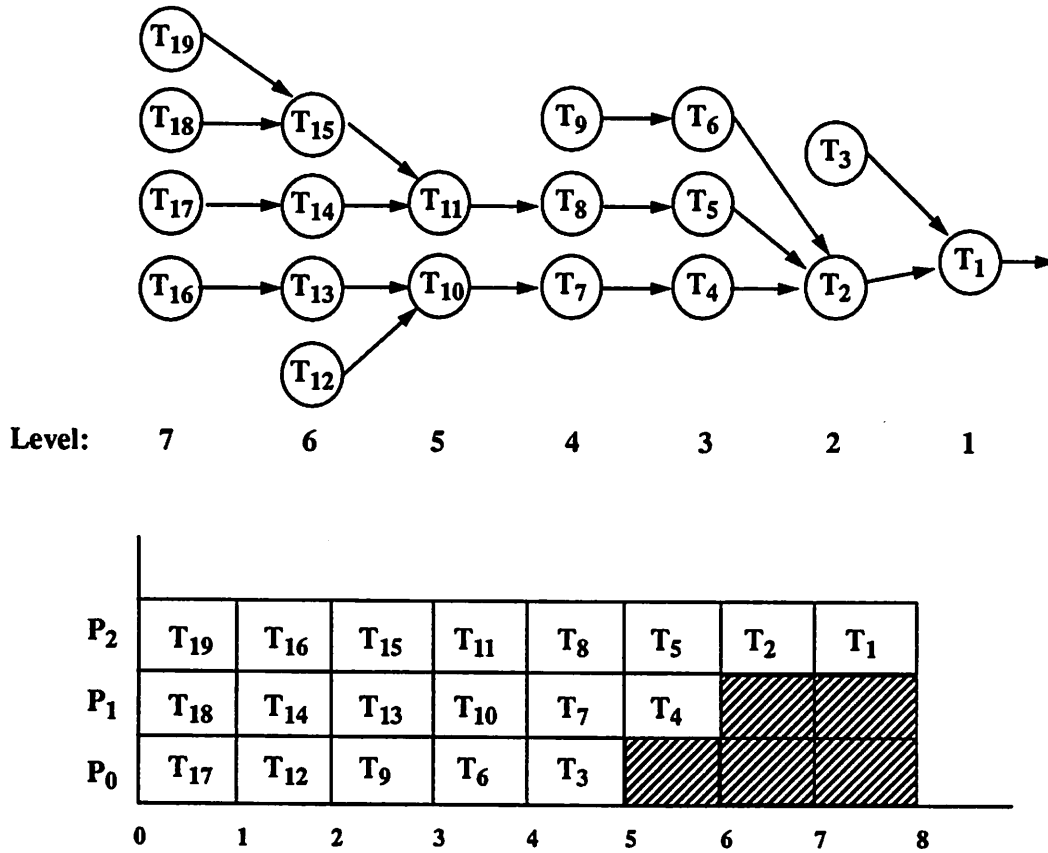
A key issue in the study of multiprocessor scheduling is the amount of computation time needed to find a suitable schedule. In computer science, an algorithm is said to be *efficient* if it requires an amount of time that is bounded by a polynomial expression of its input size. An *inefficient* algorithm is one which essentially requires an enumeration of all possible solutions before the best solution can be selected. Solutions of this type can be characterized by algorithms whose running times are exponential in the input size. These algorithms belong in a large family of seemingly intractable problems called *NP-complete* problems. Discussions on NP-completeness is discussed in more details in [Coo71][Kar72][Gar79]. For our purpose, it is only necessary to know that most of the problems of interest in multiprocessor scheduling are NP-complete [Len78][Ull75]. Since it is infeasible to enumerate all possible schedules in a reasonable time on present day computers, it is necessary to use heuristics to find a close to optimal solution to our scheduling problem. All techniques discussed below use some form of heuristic to arrive at their solutions.

## 2.4.3 Basic Multiprocessor Scheduling

Multiprocessor scheduling has its roots in management science and operations research studies, which are mainly concerned with assigning jobs to resources in the most efficient manner. If jobs are equated to programs and resources are equated to

processors, then the extension is apparent. A good overview of multiprocessor scheduling can be found in [Cof76] and [Gon77].

Perhaps the most frequently cited reference in multiprocessor scheduling is the work by T.S. Hu [Hu61] which presents a solution to the basic scheduling problem. Hu assumes all nodes have equal execution times, the precedence relationship is in form of a tree, and no communication overhead. The Hu algorithm labels each node with a *level* equal to the longest distance from the node to completion. The algorithm then schedules nodes with the highest levels first. An illustration of Hu's algorithm is shown in Figure 2.9. When trying to minimize execution time with the assumptions above, this



**FIGURE 2.9 : Hu's algorithm showing optimal schedule on 3 processors**

algorithm is actually optimal. However, relaxing the assumptions to arbitrary execution times and general precedence, the problem becomes NP-complete, and algorithms using the same technique above as heuristics became known as *critical path methods* or

HLFETs (Highest Levels First with Estimated Times) [Ada74] [Koh75]. In these methods, the level concept above generalizes to the critical path length. The nodes are sorted in non-increasing order and put into a list. The *list scheduling* method [Cof72] is then used to implement the processor assignment. A list scheduler traverses the list and assigns any idle processor to the next element on the list. Many other priority heuristics were developed to be used in a list scheduler. These include: HLFET, HLFNET (Highest Levels First with No Estimated Times), SCFET (Smallest Colevels First with Estimated Times), Random, and so on [Gra69]. There is evidence to suggest that HLFET performs quite well as a heuristic to minimize execution time [Ada74]. Theoretical performance bounds on these list scheduling heuristics are discussed in [Gra69].

In an attempt to consider communication delays, algorithms were developed which can model the overhead of data transfers between processors. The earliest work were static assignment scheduling algorithms headed by Stone [Sto77]. The algorithm uses network flow to group nodes to processors to minimize the overall execution and communication costs. Stone's solution is optimal for two processors, but the formulation becomes computationally intractable for more than three processors. Continuing developments can be found in [Bho81]. The main problem with static assignment algorithms is their lack of consideration for the data precedence constraints. Thus a minimum cost assignment may not yield a minimum execution time. These algorithms find the most use in a heterogeneous multiprocessor environment.

Another class of algorithms attempt to extend list scheduling to consider communication delays. These include the work from Yu [Yu84], Hwang [Hwa89], and Sih [Sih89]. The key lies in the fact that communication delay is characterized by the starting time of the communication, the source processor, the destination processor, and the routing path. For each candidate node to be scheduled, the earliest time it can start execution on a particular processor is determined. This is possible because all of its predecessor nodes have already been scheduled, and the communication delay from the

predecessor processors to the present processor can be calculated. Thus, to pick the next candidate node to schedule, in addition to its critical path, these methods can look at the earliest starting times of nodes to decide. By scheduling the communications on the routing resources themselves, Sih [Sih89] is able to model bus contention as well. These methods present the most accurate modelling of communications to date. Their main drawback, as pointed out in [Sih89], is that single pass list scheduling approaches are too greedy. At each scheduling instance, they only see nodes that are ready to be scheduled. this lack of global vision often prevents them from making the best scheduling decisions.

Finally, a class of scheduling algorithms have emerged, including the algorithm to be introduced in this thesis, which takes an iterative clustering approach to scheduling. The iterative refinement allows the algorithm to improve on the last scheduling step by concentrating on the most troubling spot of the last schedule. This allows these algorithms to consider the flow graph globally, yielding an improved solution over the previous local minimization techniques. Yu [Yu84] proposed an algorithm which clusters nodes with heavy communications together. The goal is to minimize the critical path of the flowgraph, which in turns, minimize the completion time. This algorithm assumes there are as many processors as needed to execute each cluster. Sarkar [Sar89] also clusters nodes to minimize the scheduling length, but then uses a list scheduling algorithm to perform processor assignment. Sih's clustering algorithm is more elaborate [Sih90]. He starts with all the leaf nodes and group those which communicate heavily into clusters. These clusters are then grouped into larger clusters and so on to the top. He then proceeds to decluster by traversing the hierarchy just built to expose concurrency. At each declustering step, he performs list scheduling on the flow graph. This allows the algorithm to take into account the architecture and arrive at the right level of granularity effectively.

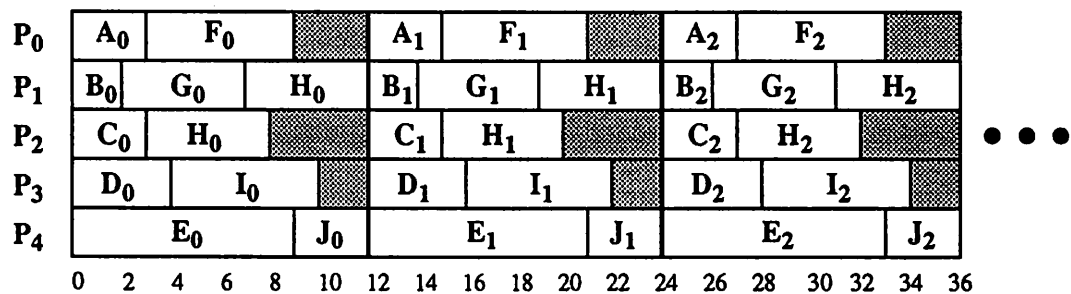


After reviewing the literature, it can be seen that the main difference between many multiprocessing scheduling problems lies in the problem formulation themselves, that is, in the various assumptions that are made. For instance, one formulation may ignore communication delays, another may not consider resource constraints, while another may assume a fix granularity. Most scheduling formulations, we found, do not exploit all aspects available to them in their respective applications. This has been especially true in signal processing.

## 2.4.4 Multiprocessor Scheduling in DSP

As a target application for multiprocessor scheduling, real-time signal processing is both a blessing and a curse. Signal processing allows temporal concurrency to be exploited in addition to spatial concurrency. On the other hand, the performance constraint requires non-preemptive, compile-time scheduling which are more difficult to construct.

One popular approach is to use the single-execution scheduling algorithms detailed in the last section for signal processing as well. In this case, the same schedule is repeated for each input sample, as shown in Figure 2.10. This class of schedulers will



**FIGURE 2.10 : Periodic Schedule built from single-execution schedule**

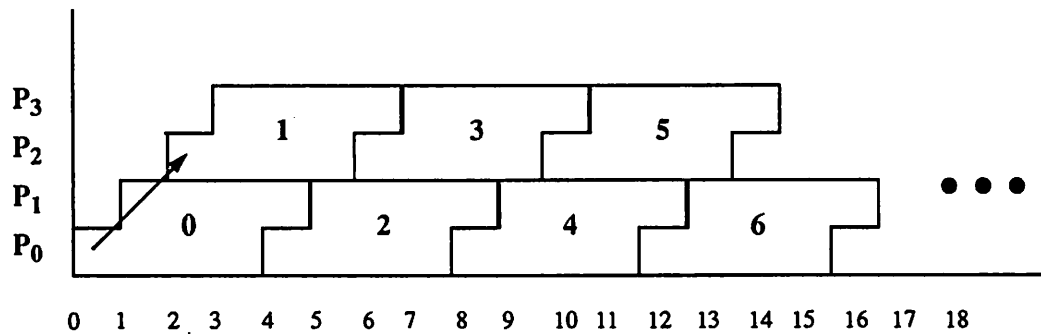
minimize the latency of the system, in the sense that the time elapsed between an input and the corresponding output is optimized. However, because the temporal concurrency

is not exploited, these algorithms do not perform as well as far as throughput speedup is concerned, especially in flow graphs with little spatial concurrency. The Gabriel [Lee89c], BDC [Zis87], and ZC [Pri91] systems introduced earlier all use this approach. They all consider communication delays and resource constraints. The ZC scheduler, in addition, has a limited capability to exploit granularity. Nodes which contain repetitive computations are classified as either parallel, systolic, or serial. When a parallel node is scheduled, for instance, the scheduler can spread the parallel computations across the processors. It is limited in that only one level of granularity is considered.

An opposite approach is taken by Bokhari [Bok88] and Hamada [Ham92]. These algorithms consider only temporal concurrency to achieve speedup. They model the application as a chain of tasks to be mapped onto a linear array multiprocessor architecture. The goal is to pipeline the chain to achieve the maximum throughput. Hamada also treats heterogenous DSP processors. Although many signal processing applications are chain-like, there is usually a lot of parallelism inside the tasks which are not being exploited.

Exploiting the fact that signal processing applications contain both spatial and temporal concurrency, Schwartz and Barnwell [Sch85] introduced the cyclo-static multiprocessor realization paradigm. Cyclo-static scheduling was developed to address single-sample-rate, fine grain, recursive DSP applications. The method computes optimal performance and hardware bounds for the application, and uses these as delimiters to the search space. The bounds computed are maximum throughput rate, minimum number of required processors, minimum latency delay, and minimum communication delay. An exhaustive search of the processor and time space is performed to find the optimal solution. The solution obtained by cyclo-static scheduling is a schedule which repeats periodically. In contrast to other periodic schedules, where the schedule of an iteration can be represented as the schedule of the previous iteration,

shifted in time by one iteration period, cyclo static schedules represent the schedule of an iteration by the schedule of the previous iteration, shifted in time by one iteration period *and* shifted in processor space by a fixed displacement. This shift information is conveyed by a *principal lattice vector*  $v \in P \times T$ . For example, a vector  $v = (2,2)$  says the schedule for the next iteration should be replicated two processors up and 2 time unit over. An example of such a cyclo-static schedule is shown in Figure 2.11.



**FIGURE 2.11 : Cyclo static schedule with  $v = (2,2)$**

Special cases and extensions of cyclo static scheduling have also been investigated as a means to narrow the search even further. These include Skewed Single Instruction Multiple Data (SSIMD) [Lee85], Fully-Static Rate Optimal [Par89], Parallel Skewed Single Instruction Multiple Data (PSSIMD) and Generalized Parallel Skewed Single Instruction Multiple Data (GPSSIMD)[For88]. While the cyclo-static paradigm is very powerful, the original formulation had only theoretical value. Communication delays were not considered, resources constraints were not handled, only fine grain single sample rate applications were addressed, and the exhaustive search for optimal solutions was exponential in complexity. However, a number of implementations based on cyclo-static scheduling are currently underway at Georgia Tech [Cur92][Gel92]. These employ heuristics to find a processor and time mapping which are suboptimal but can take into account communication delays and resource constraints.

## 2.5 COMPARISON

In this section, the McDAS design environment and its scheduling algorithm is compared with the systems examined earlier. The goal is to point out the major differences which exist and why they are present.

Unlike most systems which uses a block-diagram input specification, McDAS accepts a textual description of the DSP application. One advantage over a graphical input scheme is the increased flexibility and ease of description. This is especially true when dealing with hierarchical constructs such as loops, recursions, and conditionals, which are difficult to represent in graphical languages. Another advantage is the flowgraph generated from the input textual description is able to represent the application at all levels of granularity. This allows a scheduler to traverse different granularity levels to exploit concurrency. Finally, the semantic models of the tasks at different levels of hierarchy are completely consistent, unlike a block-diagram description, which may use different models at different hierarchy levels.

A key feature of McDAS is its ability to retarget the scheduler and code generator to different target machines. Of all the previous environments analyzed, only the Gabriel system [Lee89c] is likewise. The use of topology libraries and user-defined resource constraints allows the designer to quickly prototype and assess different realizations. Perhaps just as significant is McDAS's modular software design. The flow graph representation provides a central repository, on which all tools interact. Many optimizing flowgraph transformations such as common subexpression, dead-code elimination, loop unrolling, or multirate transformations were implemented and augmented to the design process. Other tools can be incorporated just as easily with the flowgraph interface.

The main difference however, lies in the scheduling algorithm. Like most schedulers examined, our scheduler is non-preemptive, static, of polynomial complexity, and considers communication delays and resource availability. Unlike them, it exploits both temporal and spatial concurrency to achieve a speedup. Furthermore, it can traverse different levels of granularity to find the concurrency, making it insensitive to the problem organization. This allows the scheduler to yield efficient multiprocessor realizations to a wide range of DSP applications with different types of concurrency.

## **2.6 SUMMARY**

In this chapter, we reviewed some fundamental terminologies in parallel processing and discussed previous work in three related areas -- multiprocessor architectures, parallel programming environments, and multiprocessor scheduling. A comparison of the McDAS environment and its scheduling algorithm to the existing work is then presented.

The main contribution of the McDAS system is its ability to consider and exploit all aspects of the multiprocessing scheduling problem, including concurrency, granularity, interprocessor communications, and resource constraints. This was achieved by providing a flowgraph representation which can concisely capture both concurrency and granularity information, and a scheduler which can optimally exploit them.

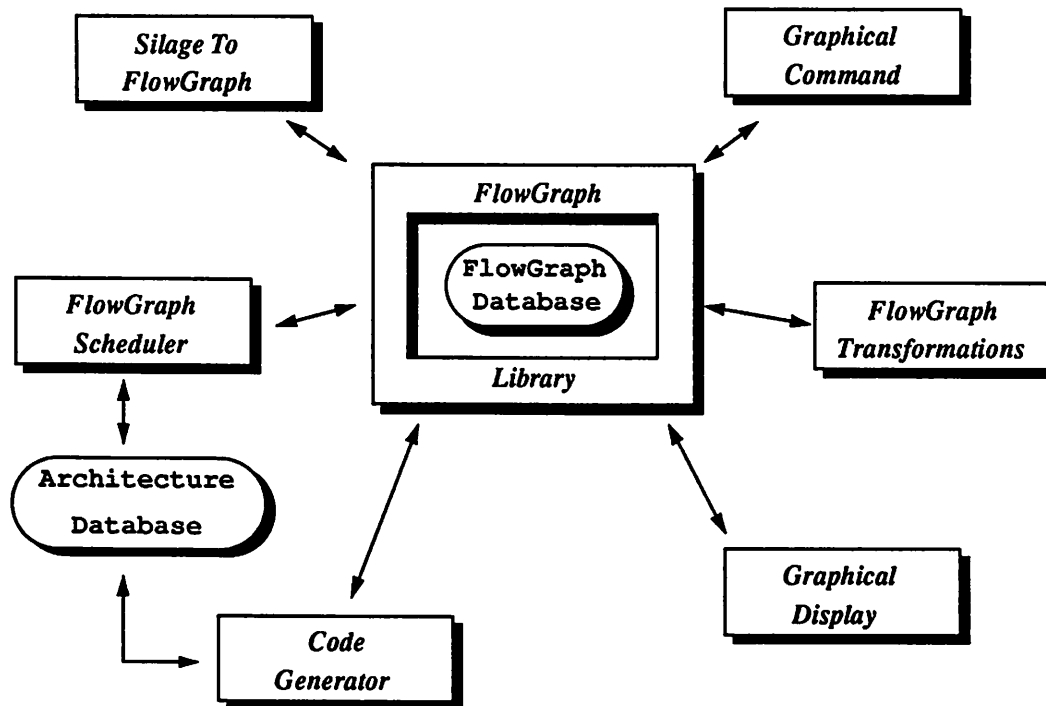
# 3

## THE McDAS ENVIRONMENT

In this chapter, the McDAS DSP design environment is presented. Section 3.1 begins with an overview of the software organization of the environment. It then explains the compilation process, from the input language Silage down to the DSP code. Section 3.2 describes the McDAS user interface, and discusses in detail the features of each tool. In Section 3.3, the language Silage is described, and specific features which make it an attractive description language for DSP are highlighted. Section 3.4 examines the hierarchical flowgraph representation. Details are given as how certain constructs such as function calls, loops, and conditionals are represented in the flowgraph. A library which supports the reading, writing, and restructuring of the flowgraph is described. Finally, Section 3.5 documents the information necessary to describe a target architecture, including the topology and the processor element.

### 3.1 McDAS SYSTEM OVERVIEW

An overview of the McDAS environment is shown in Figure 3.1.



**FIGURE 3.1 : McDAS system overview**

The system is composed of six modules operating on a centralized flowgraph database. The input is described using Silage, a signal-flow language developed especially for DSP specification [Hil85]. The modules interact with each other by reading and/or writing to the flowgraph database. Accessing the database as well as performing common operations on it are facilitated by the *Flowgraph Library*, which is linked into all tools. This *modularity of design* and extensive library support allows new modules to be incorporated easily.

The compiler environment is retargetable to different multiprocessor implementations. This is accomplished with the aid of an architecture database, which is linked to both the *Flowgraph Scheduler* and the *Code Generator*. A set of graphic

tools allow for the displaying of the essential statistics of an implementation on a given architecture. In particular, processor utilizations, bus congestions and memory requirements can be plotted to facilitate user interaction and feedback.

The design flow through McDAS is shown in Figure 3.2. The user begins with a Silage description of his application. The description is translated into a flowgraph description, and a number of architecture-independent flowgraph transformations are automatically applied to remove any dead or redundant operations. To schedule the flowgraph, the user inputs the number of processors, chooses a desired architecture topology from the database, and invokes the scheduler. Once finished, the scheduler decorates the flowgraph with the processor assignments and scheduling order. The decorated flowgraph is now called the "scheduled flowgraph". The scheduling results showing the speedup, load balancing, processor assignments, and communication costs can be displayed at this time. If not satisfied with the result, the user can select a new architecture and re-schedule. Once an acceptable solution is found, the code generator can be invoked. Currently, C code is generated which can perform floating point or bit-true simulations. This allows an algorithm designer to verify functionality, optimize application parameters, and assess the effects of finite word-length implementations. For real-time implementation on DSP processors, DSP assembly code can be generated by compiling the C code.

The Graphical Command module provides the graphical interface between the user and McDAS. Here, all commands to input data, invoke tools, display results are available to the user via menus and buttons. A sample screen is shown in Figure 3.3.

Other graphical display tools currently available include a flowgraph schematic display tool and a schedule display tool. A screen dump of a 5th order IIR filter schematics, with the feedback broken at the delay nodes, is shown in Figure 3.4. A multiprocessor schedule for 4 processors, in the form of a Gantt chart, is shown in Figure 3.5.



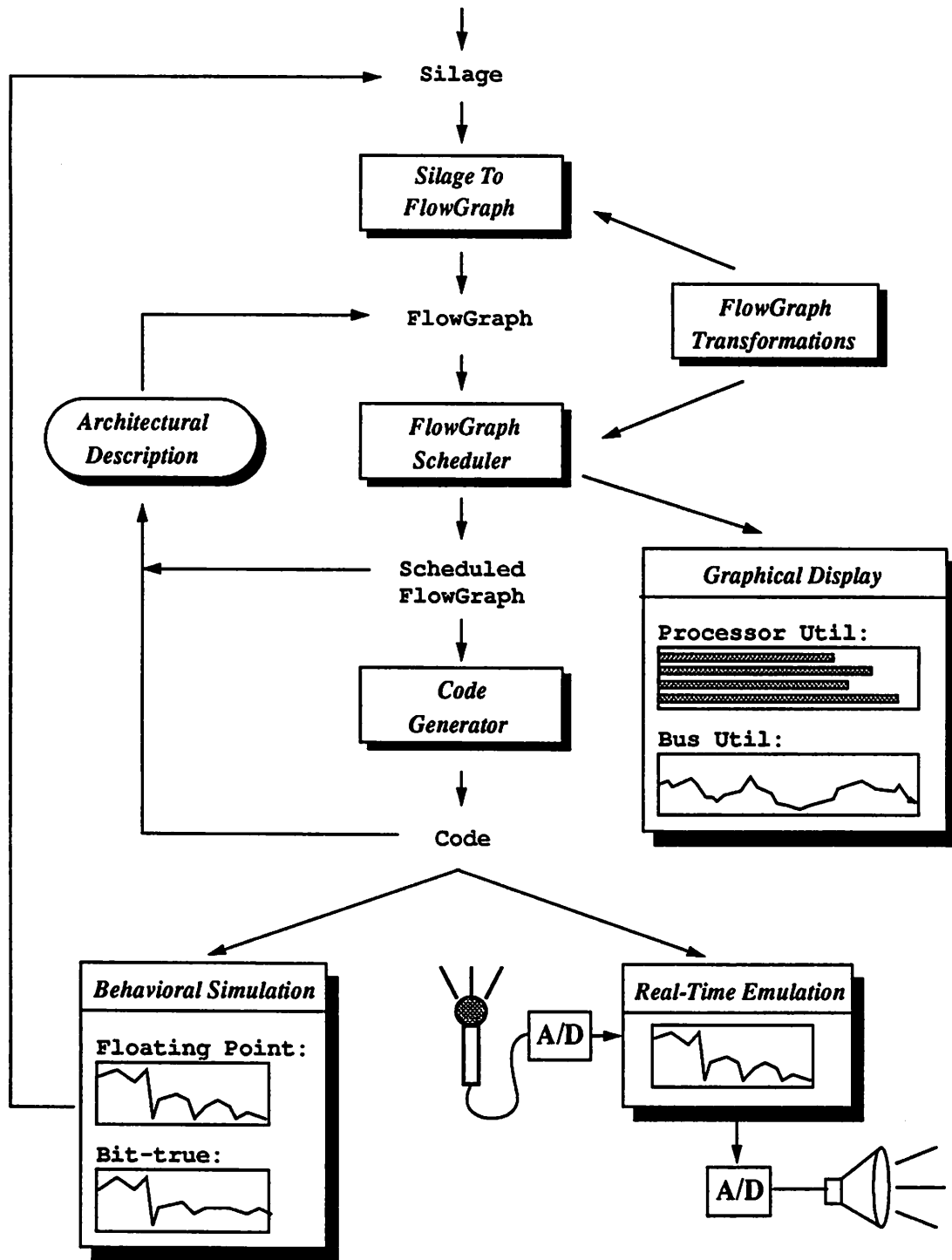


FIGURE 3.2 : McDAS Design Flow

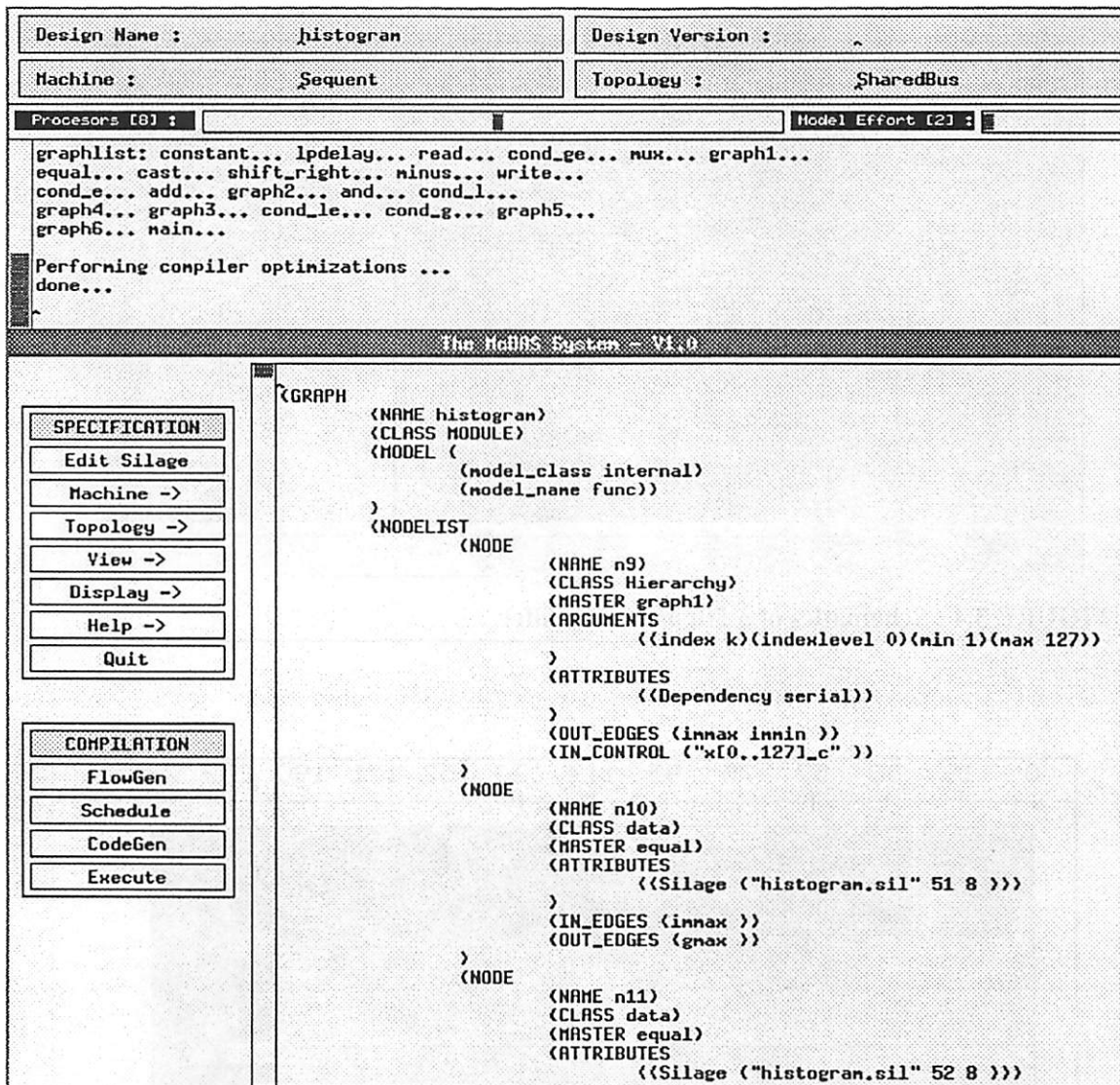


FIGURE 3.3 : McDAS Graphical User Interface

## 3.2 SILAGE

A specification language must provide to the user the primitives of the application domain. In DSP, system designers are very comfortable with a *graphical representation* of DSP algorithms. The semantics implied is known as *data-flow semantics*, in which the emphasis is on the paths followed by the inputs and intermediate results of a computation, rather than the sequence of imperative operations

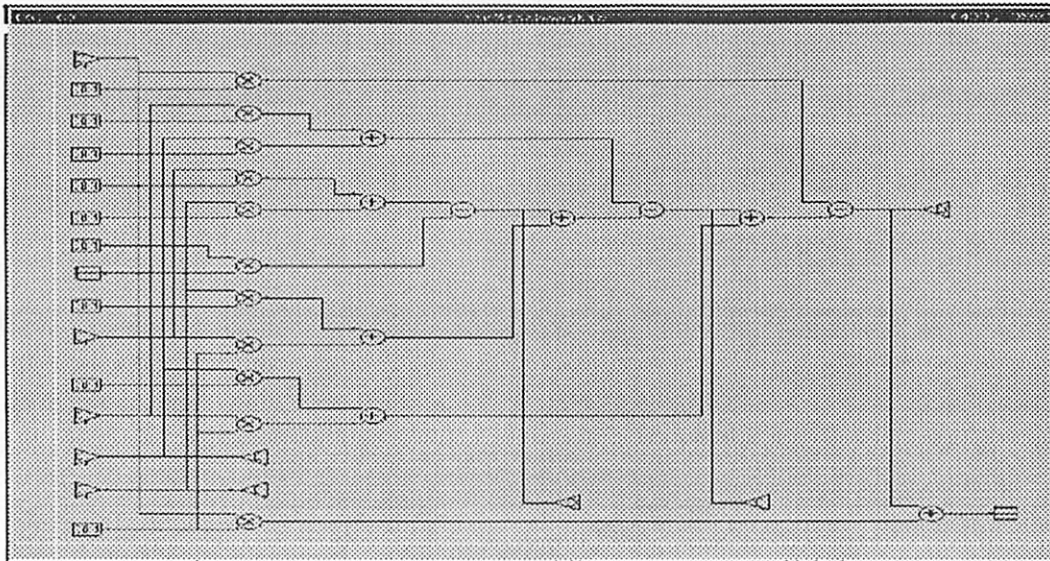


FIGURE 3.4 : Schematics of a 5th order IIR filter

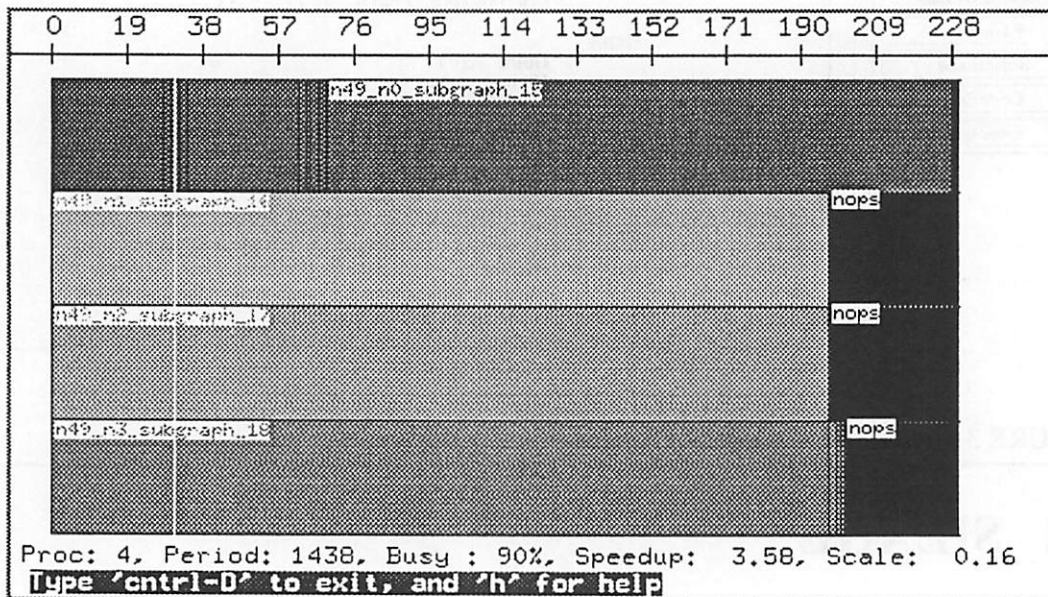


FIGURE 3.5 : Gantt Chart showing processor

performed on memory locations -- the *control-flow* semantics. The data-flow representation is not only intuitive to the designer, but also explicitly exposes potential parallelism. These two features make it the ideal specification language for synthesizing multiprocessor implementations of DSP applications, and explain the

popularity of block-diagram interfaces in a number of DSP design environments discussed previously. While this format is useful for describing traditional signal processing algorithms such as filters and modulators, signal processing algorithms have steadily grown more and more complex, requiring powerful constructs such as iterations and conditionals. The introduction of these operators to pure data-flow graphical representations has been very awkward. Textual languages such as Silage allow the addition of these structures to a data flow environment in a natural and concise manner.

Silage is an applicative language designed specifically for DSP specification. Being applicative, Silage captures the data flow concept in a linear, textual form. To support DSP, it provides as primitives certain data types and operations intrinsic to sample stream processing. An example Silage program of an 5th order IIR filter is shown in Figure 3.6. All variables or *signals* in Silage denote infinite streams of values. These signals can have integer, floating point, or fix-point data types. A fix-point type  $\text{Fix}\langle 32, 10 \rangle$  means a word length of 32 bits, with 10 bits of binary fraction. This allows the user to express precise requirements on signal accuracy, and monitor the affects of quantization and truncation. A past sample of a signal can be accessed using a Silage primitive operation '@'. For example,  $X@2$  represents the value of signal X, 2 samples earlier. These delay signals can be initialized using the '@@' operator. The signals can have different sampling rates and can be synchronous or asynchronous.

A Silage program consists of an unordered sequence of definitions of signals and of functions which are applied to these signals. In the IIR example, three functions: IIR, Biquad, and FirstOrder are defined. A signal can only be defined once, making statements like  $X = X + 1$  illegal. This *single assignment* semantic allows for a simple translation to a flowgraph format. To handle repetitive and conditional operations, Silage supports iterations and conditional expressions, as shown in Figure 3.7. A finite iteration construct allow groups of definitions to be described succinctly using an index

```

#define word fix<32,10>
#define Coef11 0.015437
..
func IIR (In: word) Out: word =
begin
  Tmp1 = Biquad(In, Coef11, Coef12, Coef13, Coef14);
  Tmp2 = Biquad(Tmp1, Coef21, Coef22, Coef23, Coef24);
  Out = FirstOrder(Tmp2, Coef31, Coef32);
end;

func Biquad (in, a1, a2, b1, b2 : word) out: word =
begin
  state@@1 = 0.0;
  state@@2 = 0.0;
  state = in - (a1 * state@1) + (a2 * state@2);
  out = state + (b1 * state@1) + (b2 * state@2);
end;

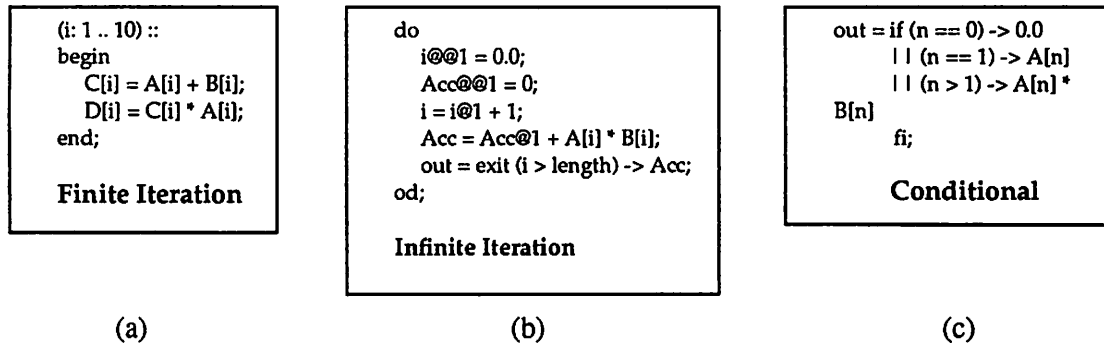
func FirstOrder (in, a1, b1) out: word =
begin
  state@1 = 0.0;
  state = in - (a1 * state@1);
  out = state + (b1 * state@1);
end;

```

**FIGURE 3.6 : Silage description of an 5th order IIR**

variable which is enumerated a known number of times. Thus, Figure 3.7a is a shorthand notation for defining  $C[0]$ ,  $D[0]$ ,  $C[1]$ , ...,  $D[10]$  separately. The implied control structure can serve as a hint to the CAD tools, which may or may not exploit it. An infinite iteration construct (Figure 3.7b), on the other hand, is a construct used to iterate a computation until a data dependent condition is met. This can be interpreted as a nesting of a separate Silage domain inside the construct. The '@' is used in this situation to refer to a value in the previous iteration of a signal in this inner domain. The *exit* clause defines the termination condition as well as the output signal of the domain. Finally, Silage supports a conditional operation to choose from among a set of signals depending on a boolean condition (Figure 3.7c). Other important features of Silage include vector operations and operations to switch sample rates, multiplex data streams, etc. The discussion on the multirate operations is deferred to Section 4.4, where a flowgraph transformation to translate a multirate description to a single rate description is introduced. Although relatively new, Silage has proven its effectiveness. A number of DSP design environments have already been built around this language,

from compiler systems [Gen89] to high level synthesis systems of custom ICs [Rab91] and field-programmable data paths [Che92].



**FIGURE 3.7 : Silage description of Iterations and Conditionals**

### 3.3 FLOWGRAPH DEFINITION

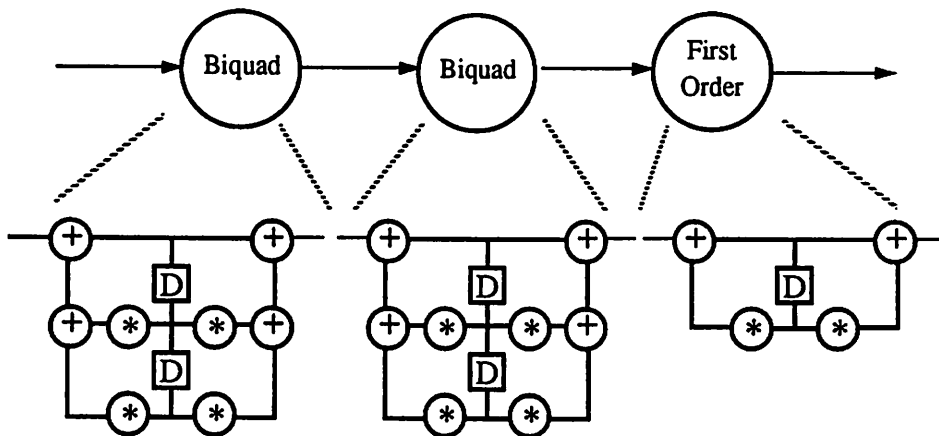
The central database to which all tools interact is represented as a hierarchical data/control flowgraph (CDFG). The nodes in the CDFG represent data operations, while the data edges represent the flow of data from the source node to the destination node. In addition, control edges can be used to enforce dependency constraints between independent nodes. Aside from the standard arithmetic operations, the CDFG allows a number of macro control-flow operations such as iterations and conditionals. With these constructs, we obtain a hierarchical graph whose subgraphs represent bodies of these iterations and conditionals. The subgraph contracts to a single node at the next level. This hierarchical representation has the advantages of compactness and descriptiveness, as it dramatically reduces the number of effective nodes as compared to a flattened flowgraph. Furthermore, the macro control-flow of the algorithm is retained, preserving the structural hints from the designer which may lead to more efficient code generation. More details on the implementation of the CDFG is given in Appendix A.

### 3.3.1 Flowgraph Model

This section describes the semantics of the flowgraph representation, which are based on the Silage language. Besides describing the primitive operators, we will also discuss the representation of constructs such as conditionals, iterations, and arrays.

#### Primitive Operators and Function calls

Each primitive operation in Silage has a corresponding primitive node in the CDFG. A function call is represented as a hierarchical node, whose subgraph points to the function body. The interpretation of these constructs is straightforward. Figure 3.8 shows the flowgraph representation of an 5th order IIR Silage description presented earlier.

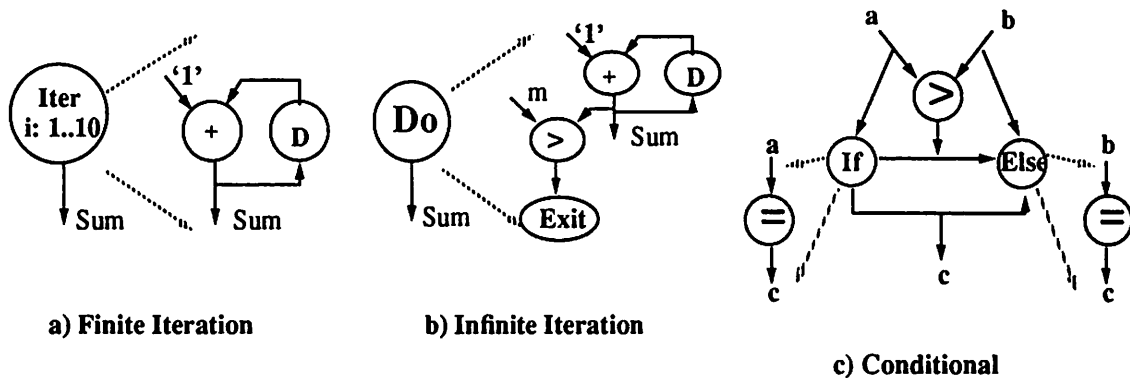


**FIGURE 3.8 : CDFG description of 5th order IIR**

#### Iterators and Conditionals

Control operators such as iterators and conditionals are hard to represent in a flowgraph. We propose an elegant scheme, based on hierarchy: An iteration or conditional is represented as a single node at the invocation level. This node is an instance of a subgraph representing the body of the iteration or conditional. Information about the iteration, such as the index and the bounds are passed as parameters to the hierarchical node. The flowgraph representations of both the finite and infinite iteration

are shown in Figure 3.9a and Figure 3.9b. There are two ways to represent the conditional operator. The first way uses the standard data-flow approach, using a multiplexer to select between two results. This representation requires both cases to be evaluated. The CDFG also allows a control conditional statement representation, where only one of the cases will be executed at run-time. The flowgraph for this construct is shown in Figure 3.9c. Notice that the signal *c* has two definitions! This still satisfies the single assignment rule since the two definitions are mutually exclusive.



**FIGURE 3.9 : CDFG representation of Iterations and Conditionals**

### Arrays

In a CDFG, arrays are represented as background memory, and all array references are interpreted as memory reads and writes. Control edges are introduced to establish precedences between array reads and writes to make sure that we only read an element after it has been assigned a value. Two nodes, *read* and *write*, are introduced to operate on arrays, as shown in Figure 3.10. A read node has two types of inputs: Control dependency edges from other nodes, and data edges holding the index values of the element to be accessed. The control edges originate from the nodes which produce the desired array elements. The node has two mutually exclusive outputs, a data edge and a control edge, depending on whether a single element or a vector is accessed. A write node has index input edges and two mutually exclusive data input edges: An input data edge or control edge, depending on whether a single element or a vector is stored.



It has one output control edge: A control precedence edge to any subsequent read nodes which want to access its data. To simplify the representation, it is possible to store the



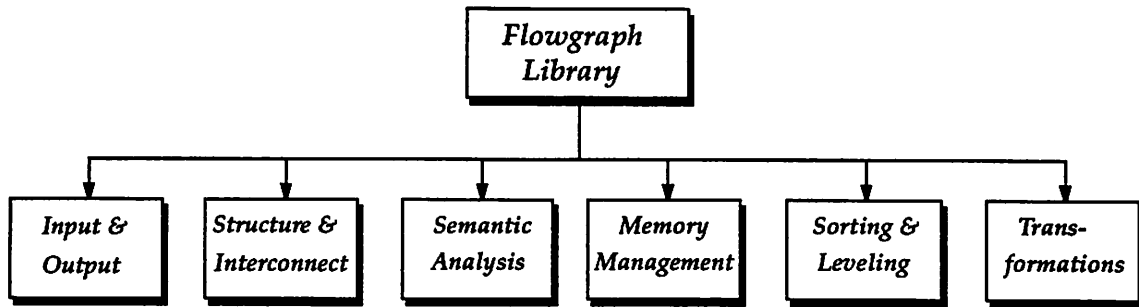
**FIGURE 3.10 : Read and Write nodes in CDFG**

array indices as arguments to the nodes if the indices are constants. In this case, no index edges are needed. Note that for non-constant array indices, in order to establish the correct relationship between different read and write nodes of the same array, careful analysis these indices is mandatory. We will expand on this task in Chapter 4.

### 3.3.2 Flowgraph Library

A flowgraph library based on the CDFG has been developed. It provides the support routines to facilitate the integration of new tools into the McDAS environment. The library is completely generic and has been used to support a number of other environments such as HYPER [Rab91], CADDI [Che92], and Aloha [Sun91]. It is composed of 6 sub-libraries, as shown in Figure 3.11.

The *Input & Output* library contains routines to translate the flowgraph from the AFL format to the OCT format (see Appendix A), and vice versa. It also has routines to read in and dump out AFL text files. This is the main mechanism for the tools to pass their results to one another. The *Structure & Interconnect* library contains all routines to create, delete, copy, and modify flowgraph objects such as graphs, nodes, edges, arguments, and attributes. All routines to connect and disconnect objects are also



**FIGURE 3.11 : Flowgraph Library Organization**

---

available here. It is the largest and most often use library. The *Semantic Analysis* library performs semantic checking on the flowgraph, such as deducing and enforcing edge types as well as resolving node and edge references. The *Memory Management* library is responsible for efficiently managing the allocation and de-allocation of memory used in creating flowgraph objects. The *Sorting & Leveling* library contains routines to topological order the flowgraph, to traverse the flowgraph in depth-first and breath-first order, to level the flowgraph from input and output, and to find the critical paths of a flowgraph. These operations are often used in many graph algorithms in scheduling and high level synthesis. Finally, the flowgraph *Transformation* library contains a number of flowgraph transformations to remove constant nodes, transform cast nodes, transform input and output nodes, etc.

## 3.4 ARCHITECTURE DATABASE

The architecture database contains a description of the different types of possible architectures that McDAS will map to. It interacts with the scheduler and code generator to customize the synthesis process to a particular target machine. Each description characterizes the core processor, the number of processors, the interconnection topology, and the memory layout of the target architecture. The content of each description includes:

1. Computation time and memory requirement of each primitive instruction of the processor such as multiply, add, shift, etc.
2. Computation time overhead and memory requirement overhead of control constructs such as function calls, loop jump and tests, etc.
3. Instruction set of the processor.
4. Program and data memory size for the processor.
5. Distance between processors in terms of bus hops, as well as the data routing paths.
6. Time required to send one unit of data across 1 bus hop.
7. The memory layout of the architecture.
8. The interprocessor communication and synchronization protocols.

The first four items characterize the processor, while the last four items describe the multiprocessor composition. From this data, it is possible for the scheduler to estimate computation times and memory requirements of nodes as well as the communication delays in data transfers. The code generator also uses the memory layout and the communication protocols to generate code for interprocessor communication and synchronization. Currently, an architecture description has components which can be modified on the fly from the McDAS command window (such as the number of available processors) and components which are compiled along with the scheduling and code generation front-ends (such as header files defining instruction costs, routines to calculate communication delays for each topology, and routines to emit code for a specific processor). Users who want to build their own architecture description can modify the header files and routines to reflect their architecture. The customized header files and routines are then compiled with the generic scheduling or code generation front-end to yield customized schedulers and code generators. These

are then available as options to the user from the McDAS command window. While it is possible to standardize these interface routines so that a textual description of the instruction set and interconnect topology can be read in on the fly, it is doubtful whether a generic code generation strategy can have enough intelligence to generate code efficient enough for real-time implementation.

### **3.5 SUMMARY**

An overview of the McDAS compiler environment has been presented, showing the components of the system as well as the design flow of the compiler. Special attention is given to the description of the Silage input language and the control/data flowgraph (CDFG) representation.

The system provides a complete synthesis path from Silage to executable code on the target machine. Both the scheduler and code generator can be customized with a header file and a few interfacing routines. A number of different versions are available to the user, allowing rapid implementations on different architectures. Users can easily construct and include new architecture descriptions.

The hierarchical CDFG structure allows all the concurrency of an application, from the finest to the coarsest level of granularity, to be represented in a concise manner. The powerful hierarchical constructs such as function calls, iterations, and conditionals permits the CDFG to effectively support a wide range of complex DSP applications. In addition, the availability of the CDFG library greatly facilitate the incorporation of new tools to the environment.

# 4

## SILAGE TO FLOWGRAPH TRANSLATION

The *Silage To Flowgraph* module translates a Silage program into a CDFG. Section 4.1 discusses the basic translation process from Silage to CDFG. Particular attention is given to the handling of arrays and data dependency. Section 4.2 discusses a number of flowgraph optimizing transformations which are automatically applied to the resultant CDFG. Finally, Section 4.3 discusses multirate DSP systems: how they are represented in the CDFG, and how they are processed by the Silage To Flowgraph module. In particular, the section introduces a graph clustering transformation to reduce a multirate CDFG into a single rate CDFG.

### 4.1 BASIC TRANSLATION

The basic Silage to CDFG translation is performed in two phases: The Silage front end, and the CDFG generation. At the time of the development of the Silage To Flowgraph module, there were two existing Silage compilers: One was the Silage simulator [Sch88] developed at IMEC Laboratory for the Cathedral II high level synthesis project; another one was the Silage compiler developed to generate RL code for the Kappa Architecture project [Wan88]. In an effort not to duplicate the time consuming parsing work, the Silage front end was derived from the IMEC Silage simulator. It parses the Silage text and builds the necessary data structures for the

construction of a CDFG. The CDFG generator then makes a 1-pass traversal of this data structure and constructs a hierarchical CDFG representation of the Silage program. Each Silage function definition gives rise to a CDFG graph, with each Silage primitive operator generating a corresponding CDFG leaf node. A Silage function call results in a hierarchical link between the calling node and the corresponding subgraph, while an iteration construct links the iteration node to the iteration body subgraph. As the nodes and edges are generated, the position of the Silage code (line number, character number, filename) is copied over for error reporting. More details on the implementation of the basic Silage to CDFG translation process is described in Appendix B.

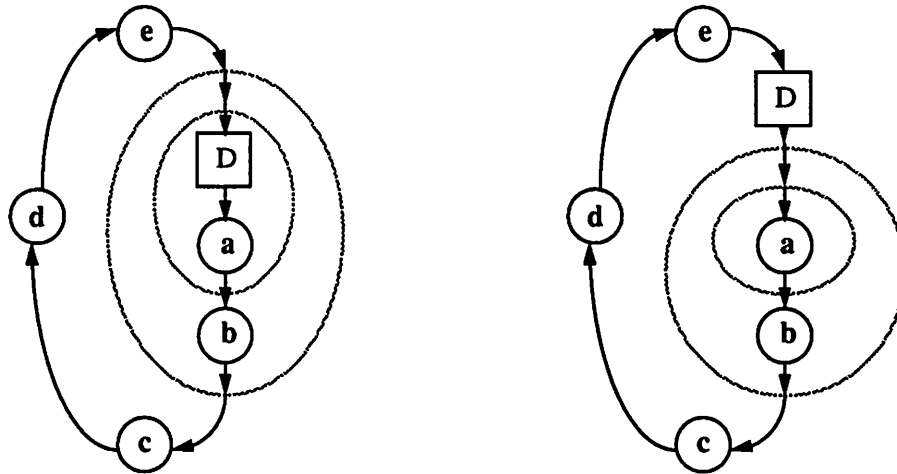
### 4. 1. 1 Handling of Delayed Signals

In Silage, a delayed signal is expressed using the '@' operator. If the delayed signal is needed to define the original signal, a recursion or *cycle* is formed. This recursive relationship must be derived from the definition of a signal and its delayed value, as shown in the following example Silage code:

```
func Biquad(in, a1, a2, b1, b2: word) out word =
begin
    state@@1 = 0.0;
    state@@2 = 0.0;
    state = in - (a1 * state@1) + (a2 * state@2);
    out = state + (b1 * state@1) + (b2 * state@2);
end;
```

In a flowgraph formulation, the recursion is expressed explicitly, using delay nodes and feedback edges. Often, a delayed signal is used before the signal itself is defined. Hence, in the CDFG construction, delayed signals must be tagged so that the CDFG generator can know to complete the cycle once the primary signals are encountered. This must be done even when the use of the delayed signal and the definition of the signal occur at different hierarchical levels. In this case, the cycle is promoted to the highest hierarchy possible so that tools which traverse the CDFG top-

down may see the cycle at the earliest time. This process is shown in Figure 4.1 This is



**FIGURE 4.1 : Delay Promotion**

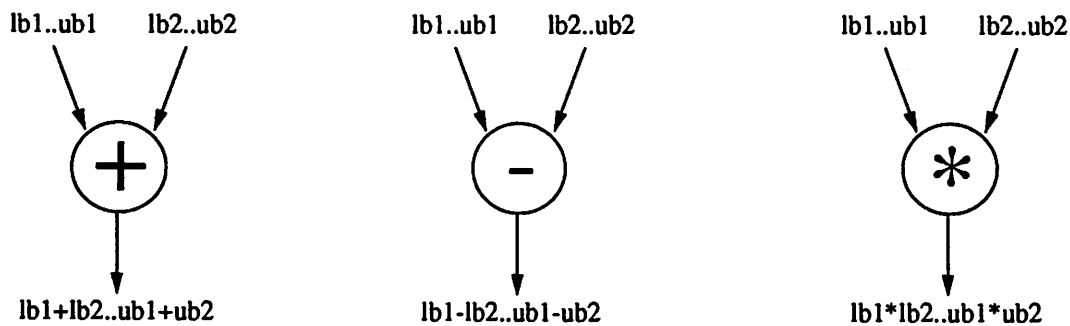
necessary so that there never exists a cycle without delay at any level of granularity.

Finally, it is desirable to build a delay line in the CDFG if the Silage description contains references to consecutive sample delays of a signal. This is present for instance, in the Silage description of the Biquad filter shown above. To accomplish this, upon encountering a delay signal such as `state@2` for instance, the CDFG Generator works backwards to find the input edge. It checks if `state@2` is present, then `state@1`, and finally `state`. The delay line is built one by one until the primary signal is found.

### 4.1.2 Generation of Arrays

As discussed in Section 3.3, array definitions and usage in Silage are implemented as reads and writes in the CDFG. To enforce correct data dependency, control edges are introduced to order the read and write nodes so that a read operation can only occur *after* the write operation which produces the data. This entails a precise analysis of the ranges of the indices on the array to enforce data dependency. To read element `A[i][j+k]` for example, it is necessary to determine what the range of indices `i`,

$j$  and  $k$  covers. Only when this is known will a correct data dependency be established between this read operation and all of the previous write operations to array  $A$ . Fortunately, Silage requires that an array index be manifest, that is, its value or range of values must be computable at compile time. This means that all read and write index edges, even if they are expressions, must have index ranges which can be computed by the compiler. This is accomplished by tagging all edges which are manifest, and propagating their values across operators whenever possible. This computation for a number of standard arithmetic operations is shown in Figure 4.2. Here, the  $lb..ub$



**FIGURE 4.2 : Manifest Value Propagation**

represents the lower and upper bound of an index. Using this technique, all index edges of read and write operations will be declared as manifest. Dependency checking now reduces to determining whether the index range of the read node *intersect* the index range of any of the write nodes to the same array.

Using  $lb..ub$  to represent an index range can be a gross simplification in some cases. For example, if we want to read to every 3rd element of an array, the index set would be  $\{0,3,6,9,12,\dots\}$ . This is abstracted to  $0..12$ , even though not all index values are needed. This scheme ensures sufficient data dependency for correct execution, but may include more dependency than necessary. As discussed in section 2.3, an exact dependency analysis would require the CDFG generator to solve a number of integer programming problems, which are exponential in complexity. Since this is not practical, the approximation as outlined above is adopted. For all of the applications that we have



tested so far, this implementation has been sufficient. However, if no extraneous dependency can be tolerated, more powerful dependency analysis techniques can be examined. This issue is addressed in more detail in chapter 9, where future research directions are discussed.

## 4.2 CDFG OPTIMIZATION

A number of flowgraph transformations have been implemented to perform optimizations on the CDFG. These transformations retain the functionality of the flowgraph but reorganize its structure to ease implementation, improve execution time, create extra concurrency, or reduce memory requirements. Flowgraph transformations can be classified into two groups: *Architecture-independent* transformations and *architecture-dependent* transformations. Architecture-independent transformations are those transformations which improve a flowgraph realization without relying on the knowledge of the target architecture. Based largely on classical compiler optimization theory, they restructure the flowgraph to remove redundant computations, increase parallelism, or replace costly computations with equivalent but less costly ones. As the architecture is not yet chosen at this point, only the architecture-independent transformations are applied. They include: Equal node removal, cast node removal, manifest expressions, common subexpression elimination, and dead-code elimination.

The first transformation aims to remove the 'equate' operation, which is redundant in data flow. The second transformation removes cast operations if the input and output data are of the same type. The third optimization collapses manifest calculations down to a single edge. The optimization takes care of algebraic identities such as addition or subtraction by 0, multiplication or division by 1, shifting by 0, or any other compile-time computable expressions. The next transformation, common subexpression elimination, attempts to remove nodes which are computing the same

thing. To do this, the graph is ordered from input to output, and for each input edge, we scan its output nodes and check for identical operations. If any two nodes are the same, we remove one of them. The ordered traversal allows entire common graph blocks to be detected and removed. Finally, the dead-code elimination transformation removes all computations whose results are not used. To do this, we scan the graph from output to input, and remove nodes whose output edges are unconnected.

The optimizations above are the implementation independent transformations available in McDAS. Many other transformations have not been implemented as it was not the focus of the thesis. Their addition to the transformations library, however, will further increase the effectiveness of the environment.

## 4.3 MULTIRATE APPLICATIONS

Some DSP systems may include a number of subsystems working at different sampling rates. Silage supports a number of operations for the re-sampling and time-multiplexing of signals. These operations map directly to their corresponding nodes in the CDFG format. However, to implement these operations, it is necessary to translate them into more basic operations. This transformation is discussed in this section. It is applied to the CDFG before the optimization step to allow the CDFG optimizer to work on its result. Before discussing how this is performed, we give a brief description of the multirate operators as defined by the CDFG policy. A detailed discussion of multirate digital signal processing can be found in [Cro83][Jac90].

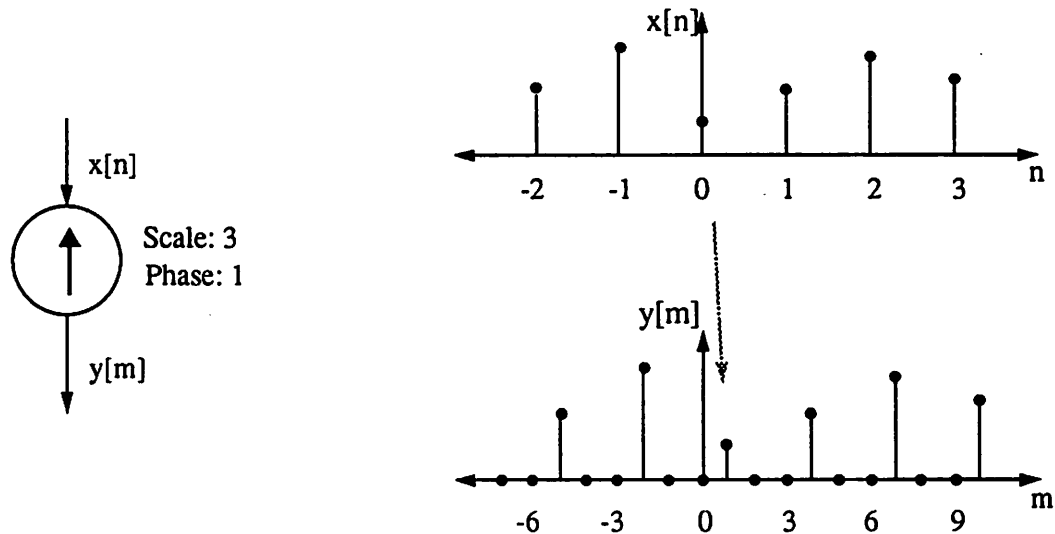
### 4.3.1 Introduction

Each signal in a synchronous data flow system has associated with it an *index* set to enumerate its samples. The index set is defined by two parameters: A *repetition period* and a *phase*. The repetition period represents the interval between successive

samples. In synchronous systems, this period is constant for a given signal, and in single-rate systems, it is constant for all signals in the system. In multirate systems, the signals can have repetition periods which are rational multiples of each other. The phase of the signal determines the offset of the index set relative to an arbitrary origin 0. The phase is expressed in terms of an integer value less than the repetition period. With these two parameters, the CDFG library supports four multirate operations:

- **Upsampling**

The upsampling function modifies the index set of the original signal  $x$  to give a new signal  $y$  with a shorter repetition period, or alternatively, a faster repetition or sampling rate. The new rate is an integer factor *scale* faster than the original rate. Values between the old values are filled with zeroes. The new phase can be offset by an integer value *phase*. The offset is with respect to the new higher rate signal. Figure 4.3

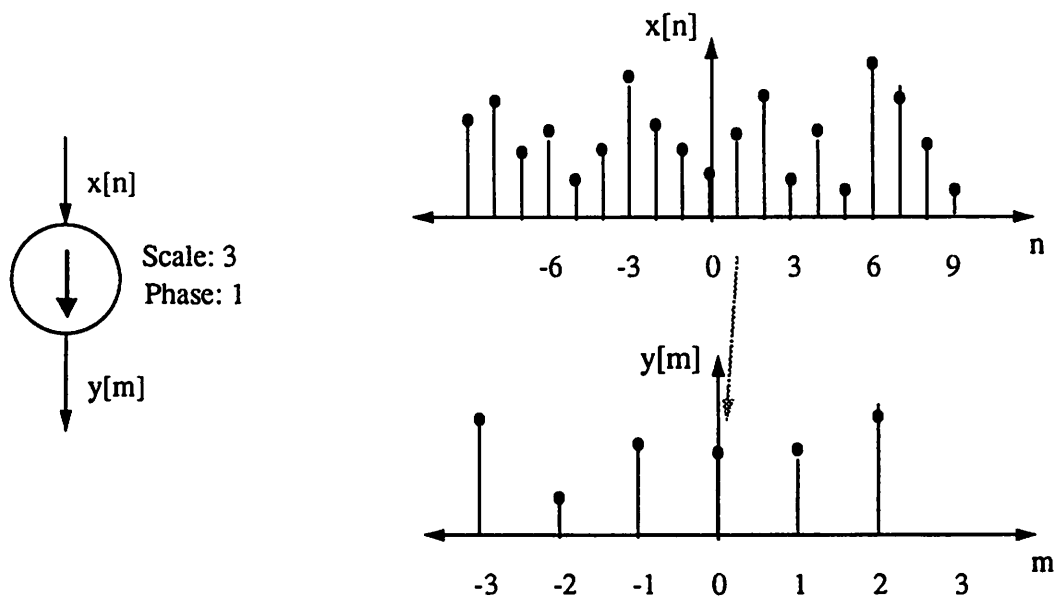


**FIGURE 4.3 : UpSampling with scale = 3 and phase = 1**

shows the upsampling of a sequence with scale = 3, phase = 1.

- **Downsampling**

The downsampling function modifies the index set of the original signal  $x$  to give a new signal  $y$  with a slower repetition rate. The new rate is an integer factor *scale* slower than the original rate. The new phase is offset by an integer *phase* offset. The offset is with respect to the old higher rate signal. Figure 4.4 shows the downsampling of a sequence with  $scale = 3$ ,  $phase = 1$ .

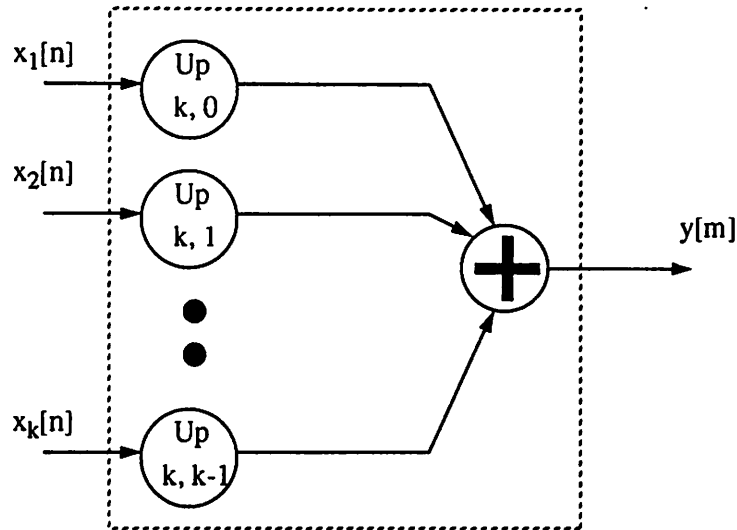


**FIGURE 4.4 : DownSampling with scale = 3 and phase = 1**

- **Time Domain Multiplexing**

The time multiplexing function takes  $k$  input signals  $x_1, x_2, \dots, x_k$ , each with the same sampling rate  $F_s$ , and results in a single output signal  $y$  with a sample rate  $k \cdot F_s$ . The samples of the output signal are defined as  $y[n] = \{x_1[0], x_2[0], \dots, x_k[0], x_1[1], x_2[1], \dots, x_k[1], \dots\}$ . This operation is actually composed of  $k$  upsampling operations with the appropriate phase factor, followed by a summation. Figure 4.5 shows this operation. The parameter of the Upsampling node is *scale*, *phase*.

- **Time Domain Demultiplexing**

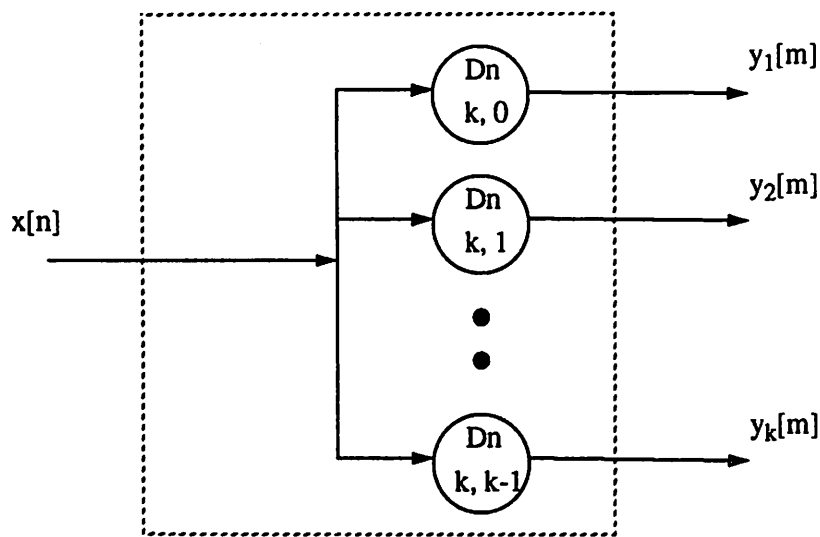


**FIGURE 4.5 : Time Multiplexing**

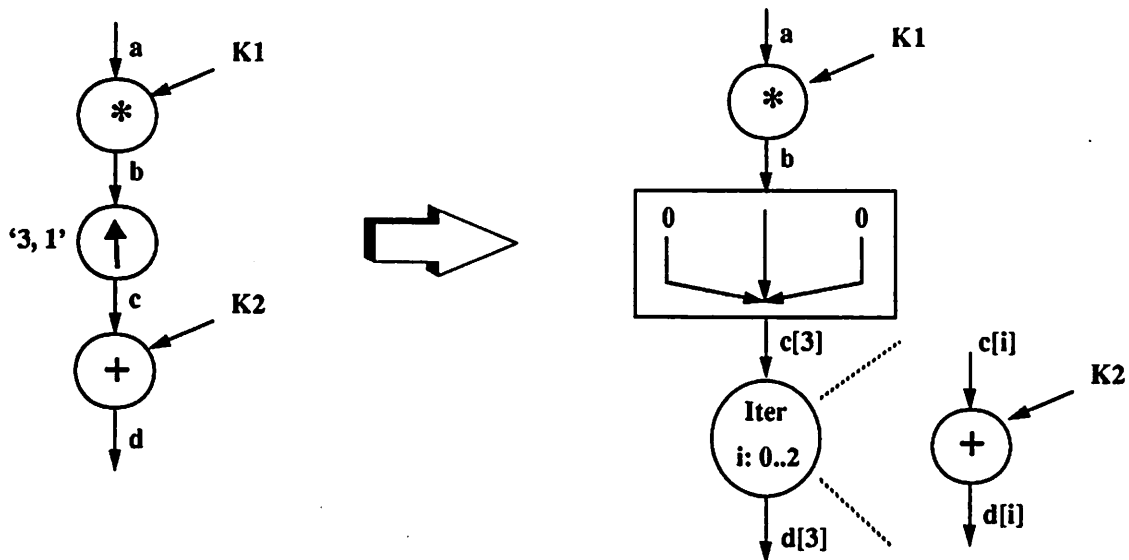
The time demultiplexing function takes an input signal  $x$ , sampled at a rate  $kF_s$ , and splits it into a set of output signals  $y_1, y_2, \dots, y_k$ , each with sample rate  $F_s$ . The samples of the output signals are defined as  $y_1[0] = \{x[0], x[k], x[2k], \dots\}$ ,  $y_2[0] = \{x[1], x[k+1], x[2k+1], \dots\}$ ,  $y_n[0] = \{x[k-1], x[2k-1], x[3k-1], \dots\}$ . The demultiplexing can be done with *nophase*, so that the resultant outputs are in phase with one another, or without, in which case the outputs are offset from one another by 1. Figure 4.6 shows a time demultiplexing operation.

### 4.3.2 Multirate Transformation

Multirate operations are high level DSP concepts which must be translated into more primitive computations for realization. Consider a simple application involving a rate change as shown in Figure 4.7. The upsampling operation produces three samples of signal  $c$  for each sample of signal  $b$ . As a result, the computation after the rate change is repeated 3 times, once for each sample. For each firing of the multiply node, the add node is fired 3 times. A CDFG representation of the required computation of the application is shown on the right. An iteration node is used to replicate the computation after the upsampling. The three samples of signal  $c$  are derived from the one sample of



**FIGURE 4.6 : Time Demultiplexing**



**FIGURE 4.7 : Multirate Transformation Example**

signal  $b$  by padding zeroes according to the phase. The samples are grouped into an array which feeds the iteration. The final output is an array  $d$  of three samples. The resultant description can now be considered as a single rate program.

We now introduce a flowgraph transformation which performs the mapping as outlined above automatically. The main idea is to cluster tasks into processes of

different rates and using iterations, invoke the processes the required number of times to yield an equivalent computation. The transformation involves several steps:

### 1. Reveal Process

This module locates the boundary of processes by examining multirate nodes. Since these multirate nodes may be hidden deep within the hierarchy of the CDFG, the graph is expanded until all multirate nodes are visible at the top level. This can be done with a simple top-down scan.

### 2. Determine Process

This module traverses the graph from input to output and assigns each node a process number. At the same time, the relative rate of a process with respect to the input rate is determined. The input rate is given a normalized value of 1. This routine is essentially a leveling procedure, where a node is assigned a new process number and a new rate if any of its inputs is a multirate node. The new rate is calculated as the predecessor rate \* scale, where scale is a parameter of the multirate node. This rate gives the relative rate of this new process with respect to the input rate. If no input nodes are multirate, they should all have the same sample rate, and the node is assigned the same process and rate. If they do not all have the same rate, we have a *multirate inconsistency* which may lead to deadlock or unbounded memory usage. This phenomenon was studied extensively by Lee [Lee89b]. At the end of the procedure, nodes which have the same rate are given the same process number, and all are assigned a relative rate with respect to the input. An example is shown in Figure 4.8, where the output of process 1 is upsampled by 2, the three signals of process 2 are time-multiplex into one signal, and the result of process 3 is downsampled by 10. The relative rates are shown in the first row of Table 4.1.

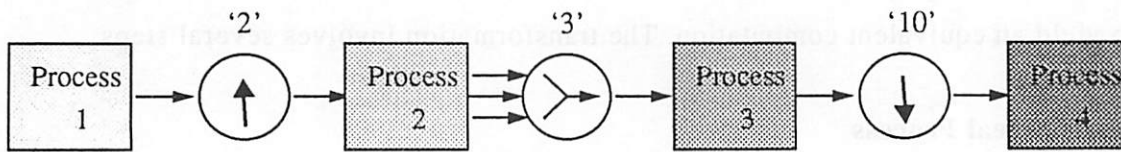


FIGURE 4.8 : A Multirate Application example

The corresponding relative rate is shown in Table 4.1.

	Process 1	Process 2	Process 3	Process 4
Relative Rate	1	2	6	0.6
Integral Rate	10	20	60	6
Absolute Rate	5	10	30	3

TABLE 4.1: Process Rate

### 3. Calculate Process Rate

This phase determines the absolute rate of the processes. The resultant rate dictates how many invocation of each process must occur for correct execution. In the input to output traversal of the previous phase, we keep track of all downsampling operations and calculate the *total downsampling factor*, defined as the product of all downsampling factors. For example, the total downsampling factor of Figure 4.8 is 10. If all relative rates are multiplied by this value, we obtain the *integral rates*, shown in the second row of Table 4.1. These rates are guaranteed to be integral since they are multiples of the maximum divided number. Although these values can be used as the final process rates, they should be reduced to their lowest integral values to save buffer memory. This is because each process must run to completion before the next process is started, forcing all outputs to be accumulated. Minimizing the number of invocations of a process reduces the amount of buffer memory which must be allocated. To do this, the greatest common divisor of all integral rates is determined, and the absolute rate is calculated as the integral rate / gcd (row 3).



#### 4. Create Process Clusters

This phase actually restructures the CDFG by clustering each process into a subgraph, and creating an iteration node to replicate the computation. The bounds on the iteration is given by the absolute rate of the process. Next, the scalar input and output signals to the processes are changed to array signals to store all results. Again, the size of the array is determined by the rate of the process. Finally, the multirate nodes themselves are replaced by operations to perform the correct data routing. For upsampling, this involves routing the input signal to the correct output signal depending on the phase, and padding all other output signals to 0. For downsampling, it involves choosing the correct input signal (according to the phase) to route to the output. For time-multiplexing and time-demultiplexing, both input and output signals have the same dimension, and there is a one-to-one assignment of input to output. The phase in this case determines the offset in the assignments.

#### 5. Flowgraph Optimization

This restructuring phase above basically completes the transformation of multirate CDFG's to single-rate CDFG's. However, it is often possible to perform more optimization on the resultant structure to improve memory usage. For instance, looking at Figure 4.7, we see that  $c[2] = b$ . We can remove this 'equate' operation, and have the output of the first process writes to  $c[2]$  directly. This *equal node removal* is even more effective in time-multiplexing and time-demultiplexing operations, where the result of the restructuring phase yields a set of equate operations between two arrays. Thus, applying this transformation removes any unnecessary copying of data.

In addition, with the current clustering scheme, the flowgraph performs the computation of an entire process before proceeding to the next process. This tends to build up much more input data than is necessary. For example, from Table 4.1, we see that the first loop will generate 10 samples before the second loop starts. However,

computations in the second loop can start as soon as 2 samples are available. By merging the two loops together, we obtain a nested loop structure where the outer loop has 5 iterations, and the inside computation involves two iterations of loop 1, followed by one iteration of loop 2. This reduces the buffer memory size from 10 down to 2. This is discussed in more detail at the end of the thesis as part of the future research.

## 4.4 SUMMARY

The Silage to CDFG translation process has been presented. This entails the parsing of the Silage code, the generation of the CDFG, and the optimization of the CDFG. The key tasks involve translating the Silage delayed signals to the feedback loops and delay lines in the flowgraph, and establishing dependencies between array accesses. This may have to be done across the hierarchical flowgraph structure.

To handle multirate DSP applications, the module uses a node clustering transformation to cluster operations with the same sample rate into processes. The processes are then invoked the required number of times to yield an equivalent computation. The number of invocations are minimized to reduce the size of buffers. Finally, optimizations are applied to remove any unnecessary copying of data.

# 5

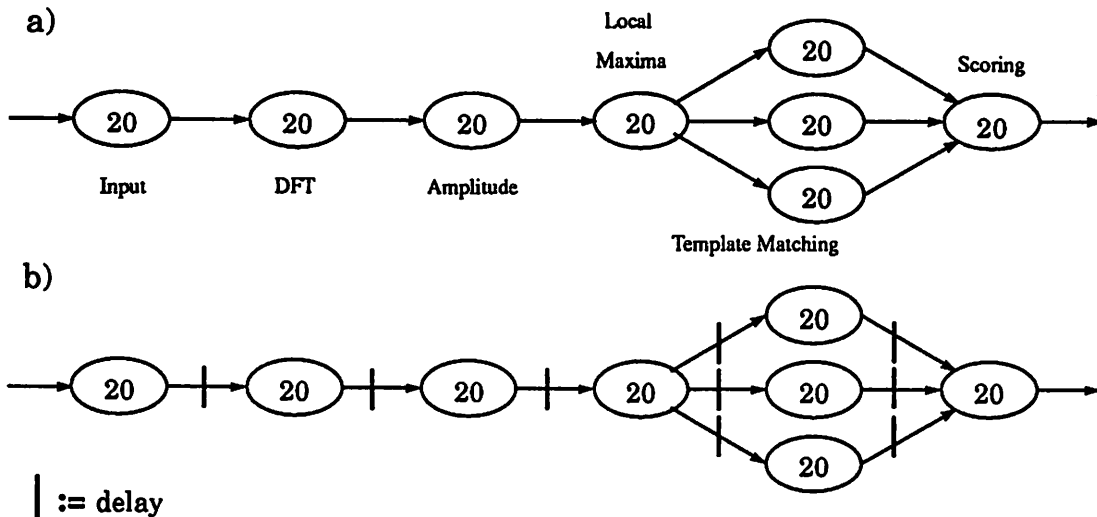
## MODEL OF COMPUTATION

Real-time DSP implementations on multiprocessors require static scheduling to reduce run-time overhead. Every static scheduling strategy is based on a computation model, which describes how tasks are partitioned and executed, and how communications are conducted. The computation model in turns make a number of assumptions on the target architecture. In this chapter, the computation model of the McDAS system is presented. Section 5.1 provides an example to demonstrate the motivations behind the model. Section 5.2 describes the partitioning and execution of tasks. Specifically, it defines the components of a multiprocessor schedule, and analyzes the throughput, speedup and communication overhead of a resulting implementation. Section 5.3 presents the architecture support required by the model.

A static schedule is only good if accurate estimations of the computation, memory, and communication costs are available. Section 5.4 discusses the estimation of computation times and memory requirements of tasks. To demonstrate its effectiveness, the estimated values are compared with actual measurements. The section concludes with a discussion on the limitations of this estimation strategy. Finally, Section 5.5 discusses the estimation of interprocessor communication delays, and shows how it enables the scheduler to take into account the underlying architecture topology.

## 5.1 A MOTIVATING EXAMPLE

A key contribution of this thesis is the scheduling algorithm, which can simultaneously exploit spatial and temporal concurrency in a DSP application. This is illustrated with the example of a Pitch Extractor algorithm for speech, whose flowgraph is shown in Figure 5.1a [Slu80]. It involves calculating the Discrete Fourier Transform of a speech signal, searching for local maxima in the spectrum to find a harmonic pattern, and comparing this against a set of template harmonic sets. In the flowgraph, nodes are labelled with their (fictional) computation costs. For the sake of simplicity, all nodes are given the same cost, and communication costs are not considered. On a single processor, the time elapsed between sample iterations, or the *iteration period*, is 160 time units.

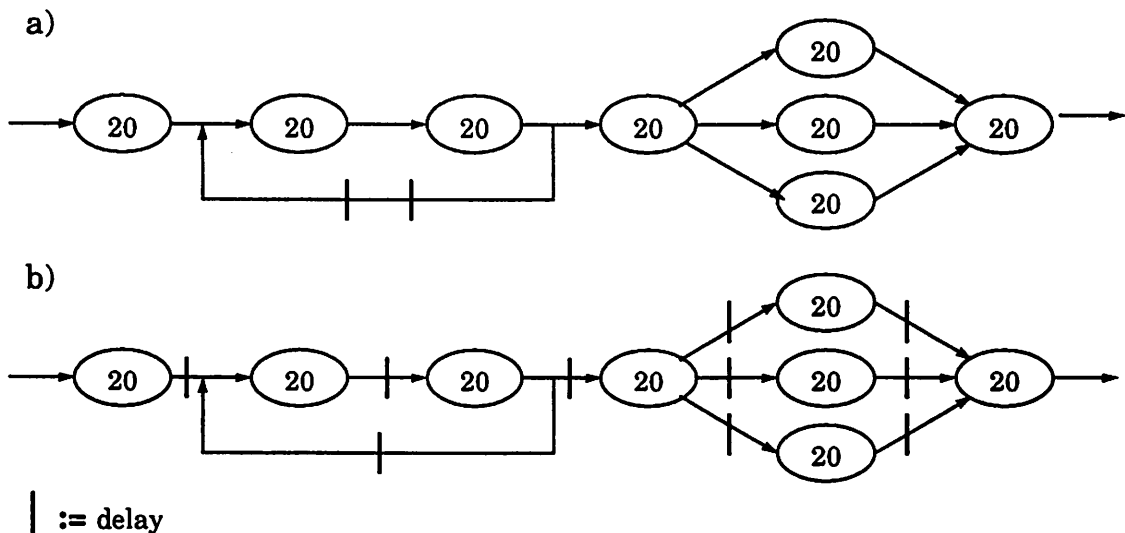


**FIGURE 5.1 : Simultaneous pipelining and parallelism.**

The goal is to exploit all available concurrency to maximize the throughput. Figure 5.1b shows the flowgraph pipelined into 6 stages, with the 5th stage having 3 tasks executed in parallel. If 8 processors are available, this partitioning is optimal, yielding an iteration period of 20 (1 output per 20 time units), a perfect speedup as compared to a uniprocessor implementation. For scheduling techniques that only

consider pipelining[Bok81], the parallel template matching task becomes the bottleneck, limiting the iteration period to be 60 time units, a speedup of only 2.7 out of 8. When considering only parallelism [Pri91][Sih89], only the template matching task can be sped up. The critical path limits the iteration period to 120 time units, a speedup of 1.3 out of 8.

Cycles in the flowgraph hinder the exploitation of pipeline concurrency. However, retiming can be used. Retiming [Lei83] involves the rearranging of delays within cycles to achieve a better performance (see Section 2.1). Consider the flowgraph in Figure 5.2a, the feedback prohibits the addition of pipeline stages on the feed-forward path, making the cycle the bottleneck computation. In order to schedule this computation and at the same time preserve its functionality, a delay can be moved from the feedback path to the forward path as shown in Figure 5.2b. This configuration utilizes all processors and yields the maximum throughput. The combination of retiming, pipelining, and parallelism fully exploits the available concurrency in the graph.

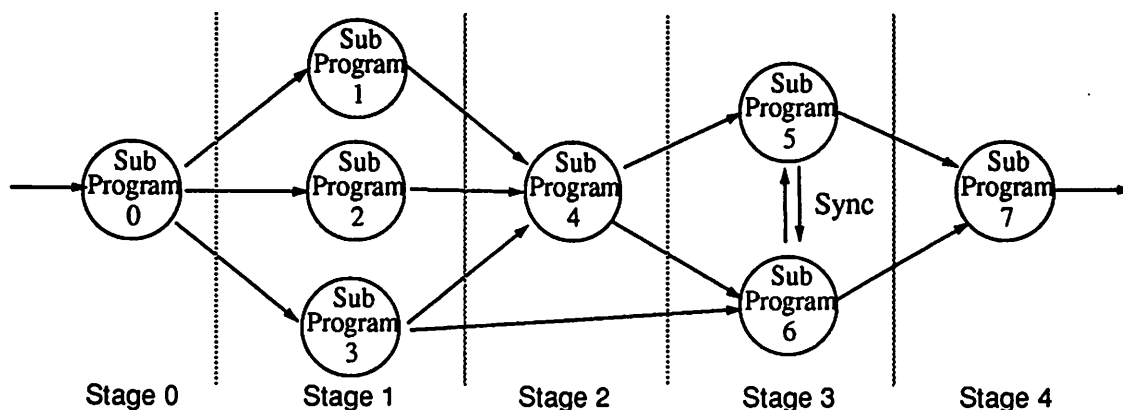


**FIGURE 5.2 : Simultaneous pipelining, retiming, and parallelism.**

## 5.2 COMPUTATION MODEL

This section describes the computation model of the McDAS environment. It explains the execution of a parallel program starting from a scheduled CDFG. As stated in Section 1.2, our target application domain is restricted to synchronous DSP systems. In addition, all multirate operations are assumed to have been transformed to single-sample rate via the transformation detailed in Section 4.4.2. The single-rate CDFG then, represents all the necessary and sufficient operations to be implemented.

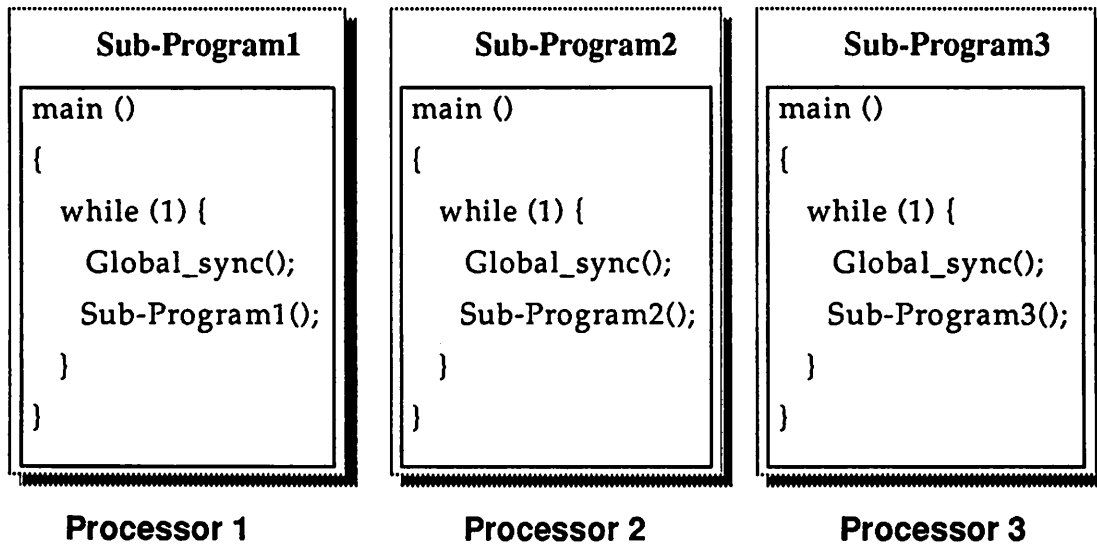
From the discussion in the last section, we see that the execution of a scheduled CDFG involves a number of autonomous sub-programs running simultaneously, with some executing in parallel, and some in a pipeline fashion. An example program structure is shown in Figure 5.3



**FIGURE 5.3: An Example Program Structure**

Each *sub-program* is a set of operations to be executed serially in some predefined order on one processor. Thus there is a one-to-one mapping of sub-programs to processors. The organization of the parallel computation, i.e., the number of sub-programs at any pipeline stage as well as the number of pipeline stages, depends on the types of concurrency available in the application. Each processor executes its assigned code once each sample period, consuming one frame of data from each of its inputs and

producing one frame of data for each of its outputs. Since data comes in an infinite stream, each processor executes repetitively in an infinite loop, synchronizing at the beginning of each sample. The pseudo-code for each processor is shown in Figure 5.4. The global synchronization serves to ensure correct data transfer from one pipeline stage to the next.



**FIGURE 5.4 : Pseudo-code for Sub-Programs**

Edges between the sub-programs are called *buffer edges*, and represent interprocessor communications. These communications are not restricted to be between adjacent pipeline stages, but may occur between sub-programs within the same pipeline stage, or from non-adjacent stages. In addition, the communication is not confined to occur only at the beginning and the end of a sub-program, but may occur at any point within a sub-program. In the example in Figure 5.3, there are eight sub-programs executing in 5 pipeline stages. Stage 1 and 3 have multiple sub-programs executing in parallel. While most data transfers in this example are between adjacent pipeline stages, there is a 2-pipeline transfer between sub-programs 3 and 6, and some interchange of data in the middle of a pipeline between sub-programs 5 and 6. As we will later see,

communications between processors in the same pipeline stage require additional synchronizations.

### 5.2.1 A Multiprocessor Schedule

The partitioning of the application into sub-programs is given by a multiprocessor schedule. A multiprocessor schedule defines the processor assignment and starting time of each node in the CDFG, and the pipeline stage of each processor.

**Definition 5.1:** A multiprocessor schedule  $\Sigma(G) = (N, PA, ST, PS)$  of a CDFG  $G$  on  $P$  processors consists of:

1. A set of nodes  $N$  of the CDFG  $G$ . The nodes can be primitive nodes or hierarchical nodes.
2. A processor assignment  $PA$  of a node.  $PA(n)$  is the processor on which node  $n$  is executed. If  $n$  is a hierarchical node, all descendant nodes of  $n$  are assigned to  $PA(n)$ .
3. A starting time  $ST$  of a node.  $ST(n)$  is the starting time of node  $n$  on processor  $PA(n)$ . It is illegal for two nodes on the same processor to have overlapping execution times.
4. A pipeline stage  $PS$  of a processor.  $PS(p)$  is the pipeline stage of processor  $p$ .

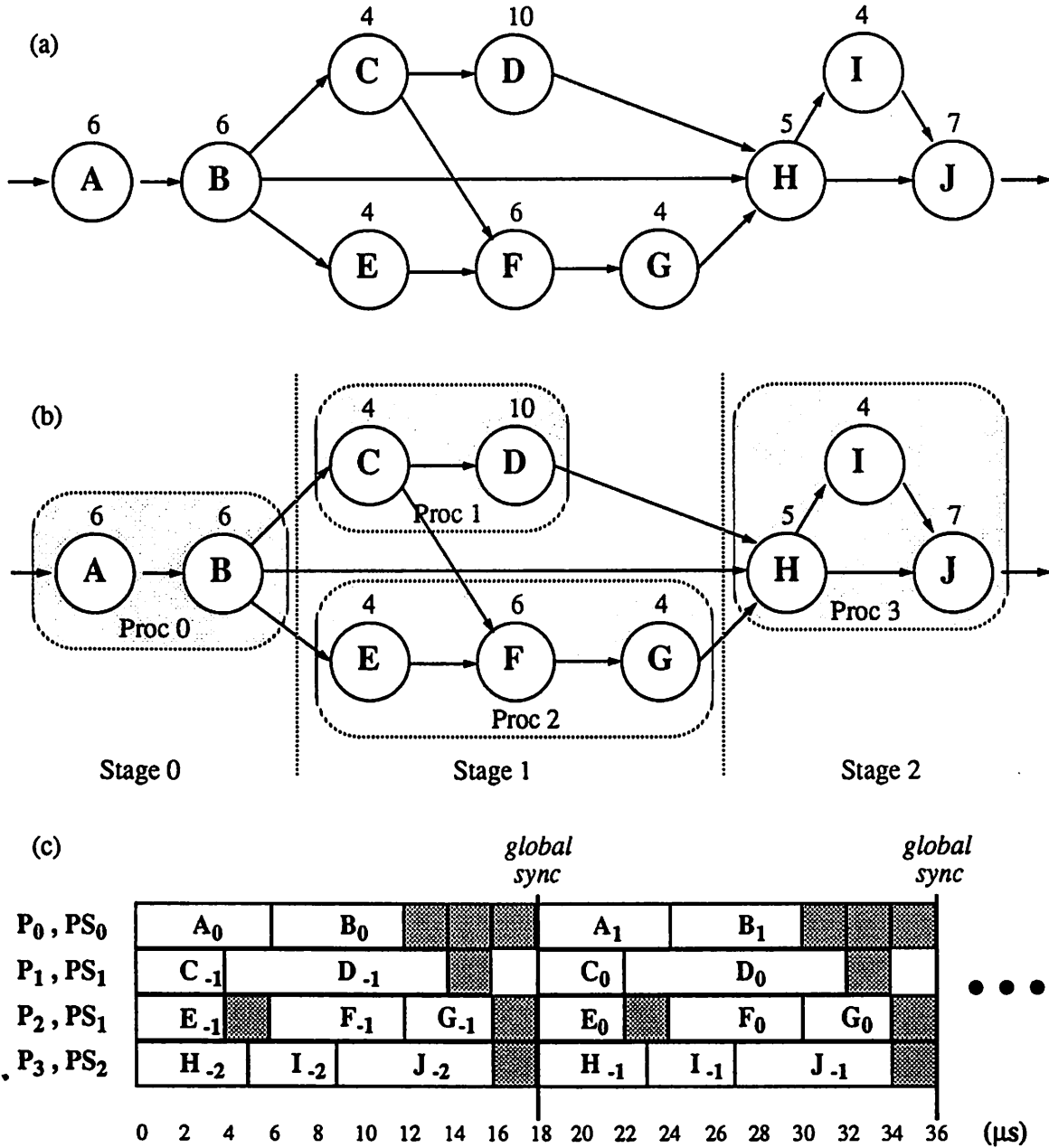
Figure 5.5a and b shows a CDFG and a possible execution schedule on 4 processors. The value above a node in the CDFG represents its computation time in  $\mu\text{s}$  (microseconds). Interprocessor communication is assumed to take  $2 \mu\text{s}$  for any transaction for simplicity. The nodes which are assigned to the same processor form the sub-program for the processor. Each processor executes its set of nodes for each input sample. The order of execution is based on the  $ST$  values of the nodes: The node with the earliest  $ST$  value is executed first, then the node with the second earliest  $ST$  value, and so on. Figure 5.5c shows 2 repetitions of the schedule in the form of a Gantt chart, showing the processor assignment and starting time of each node in the graph. The



processor number and pipeline stage of each processor are given in order (Processor, Pipeline Stage). The subscript on the node name indicates the sample being processed.

The time allocated for interprocessor communication is shown as a shaded box. This cost only occurs when the source and destination nodes are assigned to different processors. For the sake of simplicity, we assume communications only occur at the end of a computation. Node F can only start  $2 \mu\text{s}$  after node C finishes execution, as it must wait for the data from node C to arrive. On the other hand, node D can start immediately after node C finishes as its input data is already available on processor 1. Two communications are necessary at the end of processor 0 as node B needs to send its result to two different processors. Note that for architectures which support broadcasting to more than one processor, this is reduced to one communication. The global synchronization cost is assumed to be negligible in this example.

In multiprocessor scheduling, it is often not possible to exactly compute at compile time the computation and communication times of operations. This can be due to insufficient knowledge of the detailed operation of the architecture, operating system, compiler, or from an abstraction of the computation model necessary to reduce the scheduling time. Thus, although the scheduler may estimate a node to execute in a certain time  $t$ , the actual time may be  $t \pm \epsilon$ . Similarly, the communication delay can be different from estimated. As a result, the  $ST()$  values of a schedule are only regarded as estimates of the starting time, not the actual. This variation can cause incorrect execution if precautions are not taken. Consider sub-program 2 in the previous example. It is assumed that at node F's starting point, node C has finished its computation and has forwarded its data. If this is not the case, node F will execute using incorrect data. To ensure correct execution, a local synchronization is necessary at this point to delay node F until correct data is available. The local synchronization is necessary for any interprocessor communication which occur between sub-programs on the same pipeline stage. If the multiprocessor system does not have built-in synchronization instructions,



**FIGURE 5.5: An Example schedule**

semaphores can be used. In the case of a message passing computer, the send and receive primitives may implicitly perform the necessary synchronizations.

## 5.2.2 Evaluation of a Schedule

Given a schedule, a number of performance parameters to characterize the quality of the schedule can be calculated. The first parameter is speedup. Two types of speedup can be computed: Throughput speedup and latency speedup. Their definitions were given in Section 2.1.

**Definition 5.2:** Let  $\tau(1)$  be the throughput of an application on 1 processor, and let  $\tau(n)$  be the throughput of the same application on  $n$  processors, the *throughput speedup*  $S_T(n)$  obtained by executing on  $n$  processors is given by  $S_T(n) = \tau(n) / \tau(1)$ . In case of a pipelined implementation, the throughput on  $n$  processors  $\tau_{\text{pipe}}(n)$  is determined as 1 over the computation time of the slowest pipeline stage (called  $\sigma_{\text{pipe}}(n)$ ). In this case,  $S_T(n)$  is given as  $\tau_{\text{pipe}}(n) / \tau(1)$ . We refer to  $\sigma_{\text{pipe}}(n)$  as the *stagetime*.

**Definition 5.3:** Let  $\sigma(1)$  be the execution time of an application on 1 processor, and let  $\sigma_{\text{latency}}(n)$  be the latency of the same application on  $n$  processors, then the *latency speedup*  $S_L(n)$  obtained by executing on  $n$  processors is given by  $S_L(n) = \sigma(1) / \sigma_{\text{latency}}(n)$ .

The period of a schedule determines how fast the system can process incoming data, and is given by the stagetime. The stagetime is in turn determined by the execution time of the busiest processor. Assuming a cycle takes 1  $\mu\text{s}$ , the sample period of the example schedule of Figure 5.5 is 18  $\mu\text{s}$ , yielding a throughput of  $1/18 \mu\text{s} = 55.56 \times 10^3$  samples per second. Since three pipeline stages are present, the latency is  $3 \times 18 = 54 \mu\text{s}$ , reflecting the time elapsed between the arrival of an input sample and the availability of the corresponding output.

If only 1 processor were available, the execution time of the CDFG would be the total execution time of the nodes, plus 2 usescs for outputting, for a total of 58 usescs. The throughput speedup of the 4-processor schedule is  $58/18 = 3.22$ . On the other hand, the latency speedup is only  $58/54 = 1.074$ .

**Definition 5.4:** The *percent communication overhead* of a processor gives the percentage of cycles devoted to performing communication over the total number of cycles used. The *average percent communication overhead* is the percent of the total cycles used for communication by all processors over the total cycles used by all processors.

The total number of cycles includes the cycles devoted to computation, communication, as well as to idling. In the example, the communication cost of processor 1 is 4 cycles, whereas the total cycles is 18, yielding a 22.2% communication. The overheads for processors 2, 3 and 4 are 11.1%, 22.2% and 22.2%, respectively. The average percent communication equals  $12/72 = 16.7\%$ .

**Definition 5.5:** The *percent idling-time* of a processor equals the percentage of idle cycles over the total number of cycles used by the processor. The *average percent idling-time* equals the percentage of the total number of idle cycles over the total cycles used by all processors.

In the example, the idling-time of processor 1 is 2 cycles, whereas the total cycles is 18, yielding a 11.1% idling-time. Processors 2, 3 and 4 have 11.1%, 0% and 0% idling-time, respectively. The average percent idling-time is  $4/72 = 5.56\%$ .

## 5.3 ARCHITECTURAL SUPPORT

In this section, the minimal architectural requirements to support the computation model are presented. The characteristics of an individual processor are first described, followed by discussions on the interprocessor connection and communication.

### **5.3.1 Processor Characteristics**

McDAS targets MIMD parallel machines. The assumption of MIMD is essential, since the pipelining of an application usually results in different computations for different processors. Thus each processor must have its own program memory and sequencer. The processors are assumed to be homogeneous and programmable, with an instruction set that includes all standard arithmetic and logic operations. Local data memory or register files are highly desirable for efficient execution. Each processor must also have a mechanism for communicating to one or more neighboring processors. This can be through I/O channels or shared memory. Adequate support for processor synchronization is also required. The processors must be able to synchronize globally, and local synchronization between any two processors must be possible. Hardware support for these operations, while not mandatory, can greatly improve the execution times.

### **5.3.2 Multiprocessor Topology**

The only requirement on the inter-connection structure of the processors is that a processor must be able to communicate with any other processor in the system, although data may go through one or more intermediate processors. The real-time throughput constraints restrict our processors to be tightly coupled rather than distributed over wide networks. Some possible topologies are point-to-point, shared bus, ring, or star. The point-to-point interconnection is the most flexible, but is also the most expensive one to implement. The shared-bus is the cheapest to implement but is limited to a single data transfer at a time. The ring and star architectures represent compromises as they have a set of buses for communication, though no processor is directly connected to all processors.

While there are many choices, it is often possible to match communication patterns to topologies that can support them efficiently. For example, a ring architecture

is ideal for programs consisting of pipelined tasks. Processors can send data to their right neighbors without bus contention. For parallel program execution, however, the communication patterns tend to consist of either broadcasts to a group of processors, or sporadic data transfers between processors. For these, a shared bus architecture is the most efficient since a single transmission can reach any or all processors. Since our execution model involves pipelining as well as parallel execution, both types of communications may exist. Architectures which can configure their interconnection to have both shared and disjoint buses are especially attractive [Wok89]. For fixed-topology architectures, the performance of the system will depend on the matching between the architecture and the algorithm as well as the amount of communication traffic. A poor match can result in large communication overhead and lower speedup.

### 5.3.3 Interprocessor Communication

There are two common methods for interprocessor communication: Message passing between private memories, and communication through shared memory. In a message passing approach, data transfers are implemented using *send* and *receive* primitives. During its processing of a sample, a source processor may issue a *send* instruction, which packages the data into a message and transmits it over the I/O channel. On the other side, the destination processor issues a *receive* instructions to accept and store the message in its local memory.

There are in general two types of send constructs: send with blocking, and send without blocking. When a processor sends a packet with blocking, it does not resume execution until it receives an acknowledgment from the destination processor. With a send without blocking, the source processor can resume execution immediately without waiting for an acknowledge. The blocking is used to guarantee that the source processor does not overwrite a message (with another send) before the destination processor has a chance to read it. This communication block often results in large idle times of

processors, which is not desirable for real-time signal processing. The interprocessor communication mechanism to be presented in Chapter 7 guarantees that no data corruption can occur, making it unnecessary to wait for an acknowledgment after a send operation. Hence, an architecture which supports transmission without blocking is highly desirable.

When a message must be routed through a number of intermediate processors before reaching the destination processor, the path is assumed to be known at compile time (static routing) so that bus activities can be carefully modelled and estimated. The routing is also assumed to be handled automatically by the network, so that the intermediate processors do not have to expend any time or resources to the routing. Finally, since message passing can involve large data transfers with much overhead, the interprocessor communication bandwidth is assumed to be high enough to support real-time processing.

Communication can also be accomplished through shared memory. The memory can be centralized or distributed. In a centralized system, all communicated data reside in the centralized memory, accessible to all processors. The main limitation of the centralized memory is the memory access bottlenecks. In a distributed shared memory system, the memory is partitioned into sections, with each section located physically close to its associated processor. Communication is done via the global-write-local-read scheme where the source processor sends data through the network and writes it directly into the destination processor's section of shared memory. Routing of data across intermediate processors may be necessary and is assumed to be handled by hardware. The destination processor can then read the data locally without accessing the network (see Section 2.2.2.2), greatly reducing bus accesses. However, for broadcast data, all destination processors must have its own copy, as oppose to a single copy in a centralized memory system. Note that for a shared memory system, each processor may still possess its own local private memory for computation.

Both memory organizations can support the proposed computation model. The only desirable feature about a memory structure is that, in the absence of memory access conflicts, the performance is deterministic. This allows computation times of operations to be accurately estimated.

## 5.4 ESTIMATING COMPUTATION TIMES & MEMORY REQUIREMENTS

This section discusses our strategy for estimating the computation times and memory requirements for an application given a CDFG description. The goal is to obtain estimations which are as close to the actual execution values as possible. The more accurate the estimation, the more accurate the static schedule. The estimation process is divided into two sections. The first part, called *operator benchmarking*, is architecture-dependent and involves the profiling of benchmark programs on a processor of the given architecture. From these benchmark programs, a cost, or weight  $w$  is assigned to each primitive operation such as addition, shift, and multiplication. Cost estimates are also derived for the overhead of performing loop increments, loop tests, and function calls. These values would become part of the architectural description of the target multiprocessor. This task is performed manually by the user. The second phase, called *model construction*, is a collection of generic routines to calculate the computation times of hierarchical nodes in the CDFG based on the values obtained above. For a different target architecture, or for a different code generator on the same target architecture, only the benchmarking has to be redone.

### 5.4.1 Operator Benchmarking

The operator benchmarking strategy is based on examining assembly instructions derived from compiling benchmark CDFGs. The computation time of each primitive node can be estimated using a number of different parameters: The number of



assembly instructions used, the number of processor cycles required, or the actual time required to execute the assembly instructions. The instruction count method is discussed first, as it is the basis for the two latter methods. These two methods will be discussed using two different core processors: The Intel 386 processor in the Sequent multiprocessor system, and the AT&T DSP32C digital signal processor in the SMART multiprocessor array.

### Assembly Instruction Count

By examining the C code which was generated by the CDFG code generator, and then the corresponding assembly code generated by the vendor C compiler, it is possible to establish a relationship between a CDFG primitive operation and its actual assembly code implementation. Figure 5.6 shows some mappings from CDFG nodes to assembly instructions for the Sequent's Intel 386 processor. An "add" operation requires 4 instructions: 2 to load the inputs, 1 to perform the add, and 1 to store the output. Other arithmetic and logic operations such as "multiply" and "and" yield the same instruction count. Compiler optimizations which may reduce the number of data transfers between successive operations are not considered here. Other operations which have to be analyzed include shift operations, pointer and array address calculations, function calls, and loop increment and test overhead. A few constructs are discussed below.

The number of assembly instructions in an array access depends on the number of indices in the array, as these indices are used to calculate the actual memory location to be accessed. Array reads and writes are generated from *read* and *write* CDFG nodes. The first assembly instruction loads in the base address of the array. Each array index takes 2 additional assembly instructions to derive its offset from the base address, and to add it to the base address. The final instruction does the actual load or store. The total number of instructions is given by the formula  $2+2*n$ , where  $n$  is the number of indices of the array.

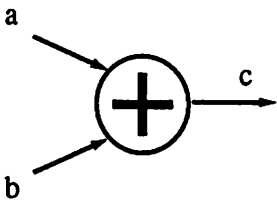
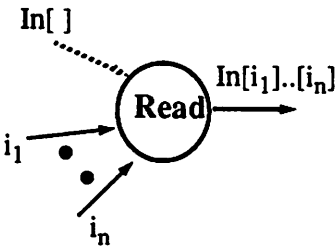
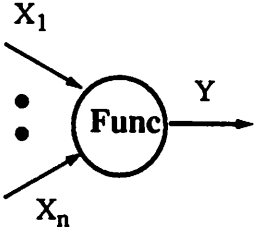
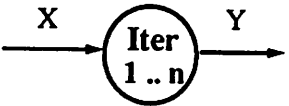
CDFG	C code	Assembly Code	Count
	$c = a + b;$	<pre>load a, r1 load b, r2 add r1, r2, r3 store r3, c</pre>	4
	$In[i_1]..[i_n]$	<pre>move &amp;In[], r1 mult i1, offset, r2 add r1, r2, r1 • • mult in, offset, r2 add r1, r2, r1 load *r1, r1</pre>	$2+2*n$
	$Y =$ $Func(X_1, \dots, X_n);$	<pre>push Xn, stack • • push X1, stack call Func pop stack, r1 store r1, Y</pre>	$3+n+$ $Func$
	<pre>for(i=1;i≤n;i++) {   Loop Body; }</pre>	<pre>initialize i (2) test i (3) call Loop Body increment i (1) test i (3)</pre>	$5 +$ $n * Body$ $+ 4*(n-1)$

FIGURE 5.6: CDFG to Assembly Code for the Intel 386

Function calls are generated from CDFG *func* nodes. Aside from the cost of the function body, there are overheads involve with the passing of the function parameters, the subroutine jump, and the restoration of the stack after the call is completed. Pushing function parameters onto the stack takes one assembly instruction each. The function call takes one instruction, and another instruction is required to restore the stack. If the result is to be stored in memory, an additional instruction is needed. The total number of overhead instructions is given by the formula  $3+n$ , where  $n$  is the number of function parameters.

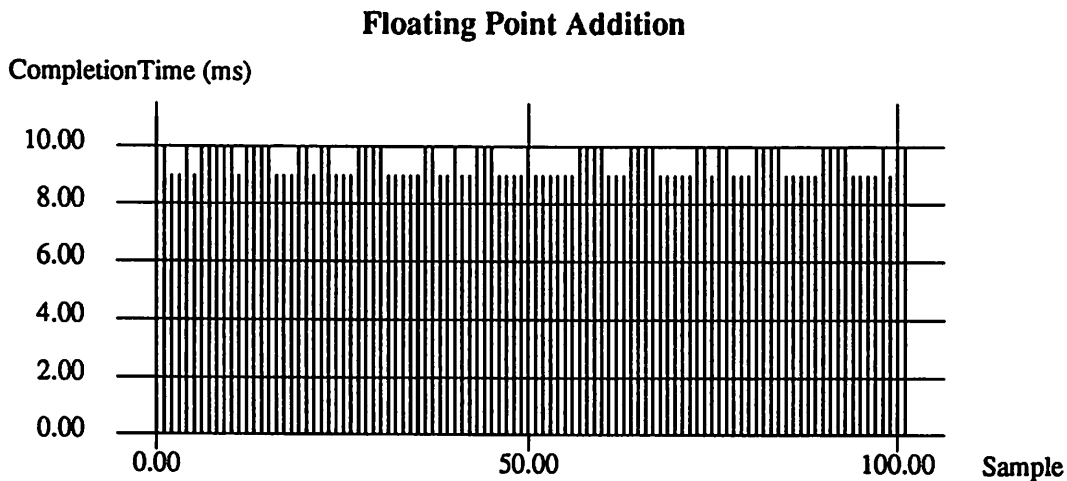
Loop structures are generated from CDFG *iter* nodes. There is a significant overhead in executing loops stemming from the bookkeeping of the loop count. For a loop with  $n$  iterations, the number of assembly instructions used for the actual computation is  $n \times \text{Loop Body}$ . However, at the end of each iteration, the loop index must be incremented and checked against its final value to determine whether or not the loop should terminate. There is a fixed overhead of 2 assembly instructions to initialize the loop index. The termination test, which takes 3 instructions, is done once for each iteration. The loop increment takes 1 instruction, and occurs  $n-1$  times. The total number of overhead instructions then is given by  $5 + 4 \times (n-1)$ , where  $n$  is the number of iterations.

Instruction counting can provide a rough estimate of the computation time of each CDFG primitive operation. However, it is often not sufficient because there is often a wide disparity in the execution times of different assembly instructions. For machines without a floating point unit for instance, a floating point operation can take much longer to complete than an integer operation. Certain manufacturers publish the execution time of each assembly instruction in processor cycles. Using this to convert an instruction count to a cycle count can give a better estimate of the execution time. Other manufacturers offer built-in timing mechanisms which help the user to measure exact execution times. One such vendor is Sequent Computer Systems.

### Time Measurements for the Intel 386 on the Sequent

The Sequent Symmetry multiprocessor system has a microsecond clock which allows parallel programmers to do fine-grain timing studies of program execution. The clock is used to measure the computation time of the Intel 386 assembly code for each CDFG operation. Two calls to the system clock at the beginning and the end of the instruction(s) to be measured yield the number of elapsed microseconds. The overhead of the call is specified by Sequent Computer Systems to be 2  $\mu$ s.

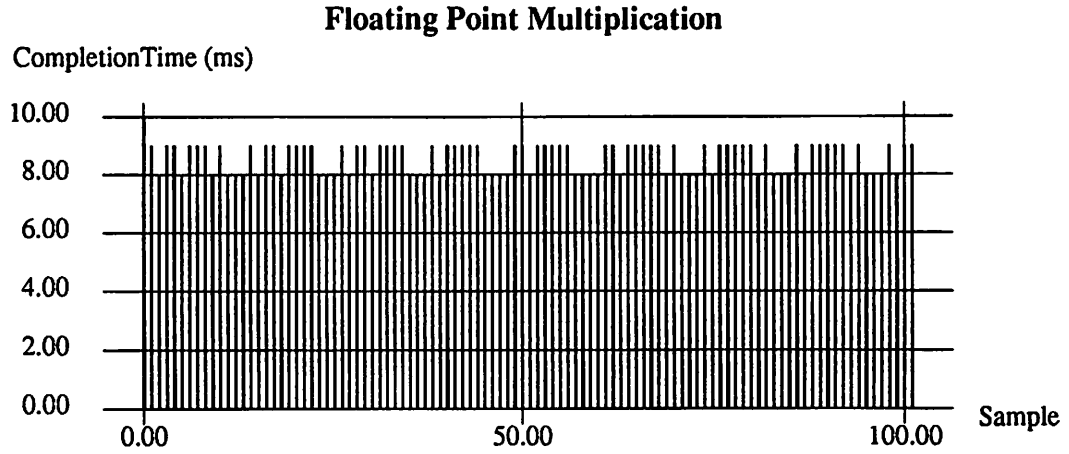
To obtain an accurate reading, many measurements are taken and averaged. Figure 5.7 through Figure 5.9 each show 100 measurements of the computation time of a C statement of the form  $c = a$  'op'  $b$ , where 'op' is a floating point add, multiply, and comparison, respectively. The measurements include the fetching of the two operands from memory and the storing of the resultant operand to memory.



**FIGURE 5.7: Measurements of Floating Point Additions**

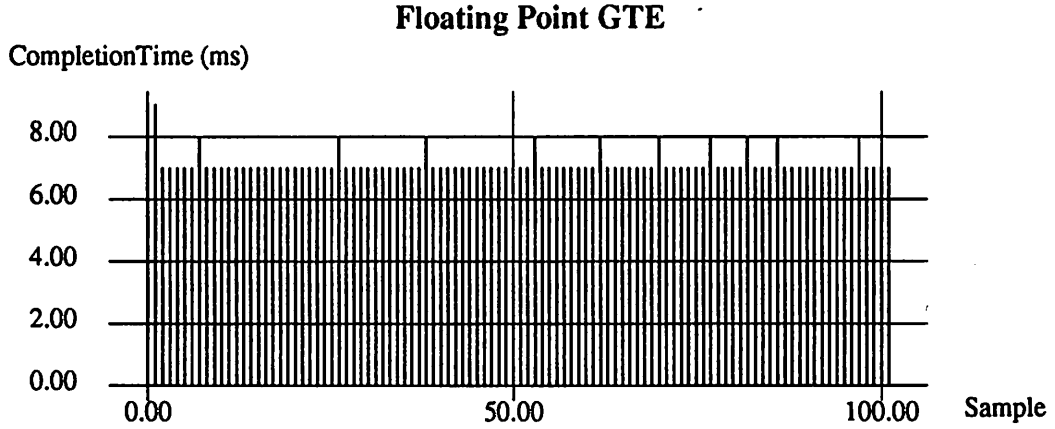
Integer operations are in general cheaper than floating point operations, as illustrated in Figure 5.10 and Figure 5.11.

Table 5.1 and Table 5.2 shows the measured computation times of a number of common operations on the Intel 386 processor. Some operations, such as a function call



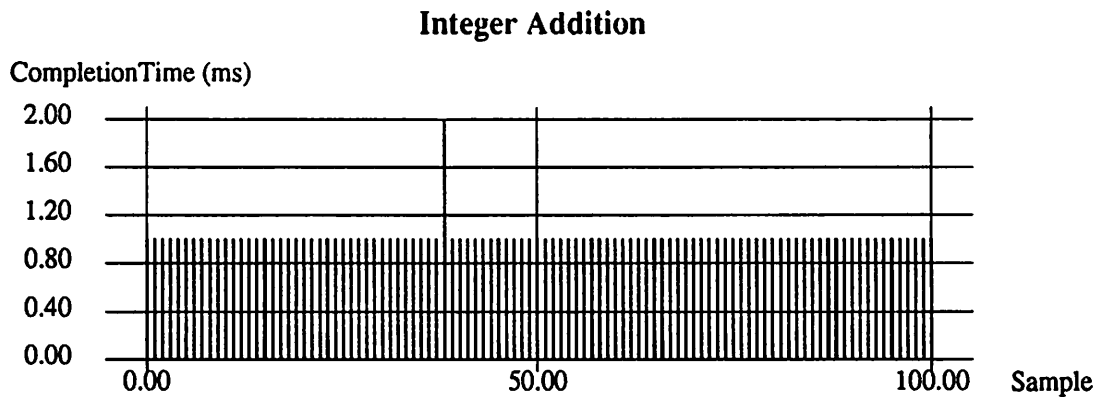
**FIGURE 5.8: Measurements of Floating Point Multiplications**

---



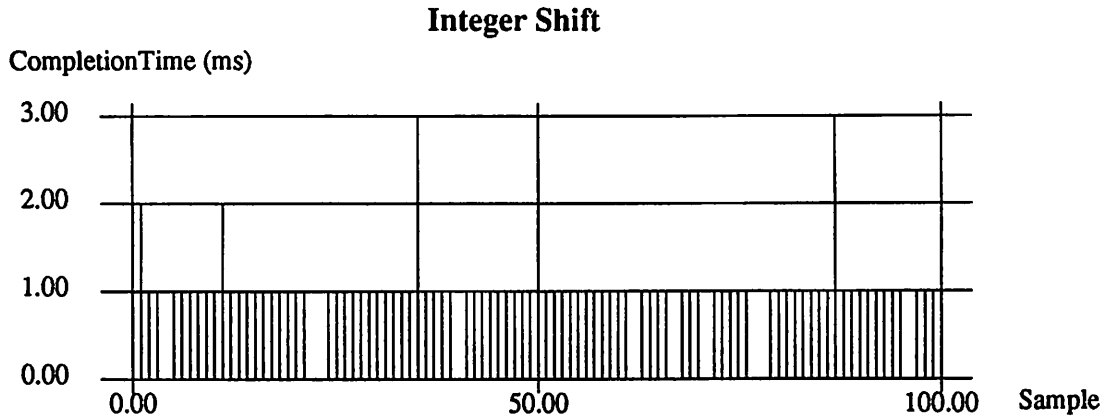
**FIGURE 5.9: Measurements of Floating Point "≥"**

---



**FIGURE 5.10: Measurements of Integer Additions**

---



**FIGURE 5.11: Measurements of Integer Shifts**

and a loop increment, are characterized by more than 1 parameters. In the array access,

Float Add	Float Sub	Float Mult	Float Div	Float GTE	Float Assgn
9.25 ms	9.25 ms	8.5 ms	13.5 ms	7 ms	0.5 ms

**TABLE 5.1**

Int Shift	Int Add	Int Assgn	Array Read	Func Call	Loop Test
1 ms	1 ms	0.5 ms	(1+n) ms	(4 + 6.5n) ms	(1 + 2n) ms

**TABLE 5.2**

n represents the number of indices, while in the function call, it represents the number of function parameters. In the loop test, it represents the number of iterations.

**Time Measurement for the AT&T DSP32C on SMART**

The DSP32C digital signal processor has available an emulator which allows users to execute programs on a virtual DSP32C. It can show the memory reads and writes, and the execution of each instruction down to the processor cycle. The emulator is used to estimate the execution time of primitive CDFG operations by simulating their corresponding assembly code. The computation times of the above operations on the

DSP32C is given in Table 5.3 and Table 5.4. The unit of measurement is in processor cycles. For a 50Mhz DSP32C, each cycle takes 20ns.

Float Add	Float Sub	Float Mult	Float Div	Float GTE	Float Assgn
22	22	22	210	53	16

TABLE 5.3

Int Shift	Int Add	Int Assgn	Array Read	Func Call	Loop Test
$40+8n$	32	16	$24n$	$24+8n$	$40+49n$

TABLE 5.4

In the DSP32C, a floating-point division is implemented by software, and hence requires a significant amount of time. In the integer shift operation,  $n$  represents the number of bits to be shifted.

## 5.4.2 Model Construction

Given the cost of all the primitive nodes and all the overhead operations, the model construction routines calculate the cost of every node in the hierarchy of a CDFG by traversing the flowgraph bottom up, accumulating computation times of primitive nodes into subgraphs, and so on up to the root graph.

Let us represent a hierarchical CDFG as  $G = (N, E)$ , where  $N$  is the set of nodes and  $E$  is the set of edges in the top level hierarchy of the graph. The set  $N$  can be divided into three sets: The set of *primitive* nodes  $N_p$ , the set of function call nodes  $N_f$ , and the set of iteration nodes  $N_i$ . Function call nodes and iteration nodes are hierarchical nodes. Let  $SG(\cdot): N_f \cup N_i \rightarrow G$  denote the function which returns the underlying subgraph of a hierarchical node, and let  $N(\cdot): G \rightarrow N$  be a function which returns the set of nodes of a graph.

The model construction algorithm defines the computation time  $w(\cdot)$  of each node in a hierarchical graph  $G$  using the following three rules:  $\forall n \in N$ ,

1. If  $n \in N_P$ ,  $w(n) =$  predefined cost, based on benchmark results.
2. If  $n \in N_F$ ,  $w(n) = w_o(n) + \sum_{v \in N(SG(n))} w(v)$ , where  $w_o(n)$  represents the overhead of the function call, as analyzed in the benchmarks.
3. For  $n \in N_I$ ,  $w(n) = w_o(n, L) + L \cdot \sum_{v \in N(SG(n))} w(v)$ , where  $w_o(n)$  represents the overhead of the loop, as analyzed in the benchmarks, and  $L$  is the iteration count.

Table 5.5 shows the estimated and measured computation times of a number of standard DSP and numerical applications on the Sequent machine. The programs were generated automatically by the CDFG code generator and Sequent C compiler.

Example	# Operations	Estimated (ms)	Measured (ms)	% Error
8pt-DCT	87	2882	2790	+3.20%
Cordic	494	11,551	11,668	-1.00%
2 Norm	1926	59,259	60,997	-2.85%
Histogram	30,687	248,291	237,667	+4.27%
256pt-DFT	760,330	7,958,590	7,944,570	+0.17%

TABLE 5.5

To quantify the complexity of the examples, the number of primitive operations involved in each is given. The error shows that the estimation routine is able to achieve an approximation to within 5% of the actual execution time. Table 5.6 shows the estimated and measured computation times of several examples on the AT&T DSP32C processor

The measured computation times are obtained using the DSP32C emulator.

Example	# Operations	Estimated (cyc)	Measured (cyc)	% Error
7th-IIR	55	6497	6414	+1.2
8pt-DCT	87	2388	2511	-4.89%

TABLE 5.6



Example	# Operations	Estimated (cyc)	Measured (cyc)	% Error
Cordic	494	135,098	137,924	-2.04%
2 Norm	1926	109,812	114,099	-3.75%
Histogram	30,687	4,186,747	4,227,738	-0.97%

TABLE 5.6

### 5.4.3 Limitation of the Technique

Although the operator benchmarking discussed in this section is accurate, it is based on a simplified code generation procedure. Specifically, it does not take into account architecture-dependent optimization and code generation techniques that may improve execution time. For instance, the estimation of computation time and memory of a CDFG operation include the loading of both input operands into registers, and the storing of the output operand into memory. No attempt was made to take into account register allocation optimizations, which may remove redundant memory accesses. In addition, the processor itself may be heavily-pipelined, or has the capability of execute a number of instructions simultaneously, both of which can significantly affect the code being generated.

The main problem lies in the fact that the CDFG only describes the computations which must take place, not how it will be done. Although not implemented, we see three approaches to solve this problem: Firstly, the operator benchmarking approach can be applied to benchmarks which are optimized. If a large number of real life programs are evaluated, an average value can be obtained for each operator. This is adequate if absolute accuracy is not necessary. The second approach is to build into the estimation routines the models for register allocation, and instruction scheduling that is specific to the processor. The main drawback of this approach is that the model construction routines will become architecture or processor dependent. Another drawback is that detailed low-level interactions must be modelled, yielding an estimation procedure which is complex and slow. The third approach involves the use of

large, library-based computational blocks. These blocks would have optimized assembly code hard-wired into them, making it possible to determine their computation time and memory requirements exactly. The only ambiguity in estimation would come at the interfaces between the library blocks, which for most cases is negligible.

#### 5.4.4 Memory Estimation

A communication between two processors incur memory storage at the destination processor to buffer the data. There is often a limit on the size of this *buffer* memory. It is possible to keep track of the buffer memory usage during scheduling so that solutions which violate the memory limit can be discarded. From a scheduled CDFG, an edge between two nodes assigned to different processors represents an interprocessor communication. The buffer memory requirement of a node  $n$  on a processor  $p$ , denoted as  $bm(n,p)$ , is given as the sum of the buffer memory requirements of all of its input edges. The memory requirement of each input edge depends on the size of the data on the edge and the difference in the source and destination pipeline stages. An edge connecting two nodes assigned to the same processor does not require any buffer memory. The parameter  $bm(n,p)$  is used by the scheduler to prohibit a node from being assigned to a processor if executing this node would overflow the processor's buffer memory. To do this, each processor  $p$  has to also keep track of its remaining buffer memory size during scheduling. This parameter is denoted  $bm_{avail}(p)$ . Chapter 6 will describe how  $bm(n,p)$  and  $bm_{avail}(p)$  are used by the scheduling algorithm.

There are a number of other memory parameters which may be important. One parameter is the program memory required to store the code. This information is available from the computation time estimation. Since each primitive CDFG operator is compiled to assembly code in the operator benchmarking procedure, the code size of a CDFG can be determined by accumulating the code sizes of its nodes. Another

parameter is the memory requirement for static and global variables. Currently, constant edges in a CDFG are compiled to global variables. The estimation of static memory requirements is thus trivial. Finally, it is possible to estimate the maximum stack memory requirements. A stack is used to store formal parameters and local variables in a function. The number of variables can be estimated by examining and counting edges in a CDFG. The addition of these memory estimation tools to the McDAS environment can enhance its generality, and is a good topic for future research.

## **5.5 ESTIMATING COMMUNICATION DELAYS**

The communication delay depends on the amount of data being sent and the distance between the source and destination processors. To calculate the delay, the size of the data, the delay in transmitting a piece of data, and the path it travels must be known. In addition, the delay in routing a message through an intermediate processor and in arbitrating bus accesses (if applicable) must be known. In this section, a time-slot model is presented to estimate communication delays. In this model, when a node is scheduled on a processor, the data transfers that are needed to bring the input variables from their source processors to the current destination processor (if data is non-local) are also scheduled on the appropriate bus or busses. This gives a scheduler a very clear picture of the bus usage, and allows it to include bus congestion effects.

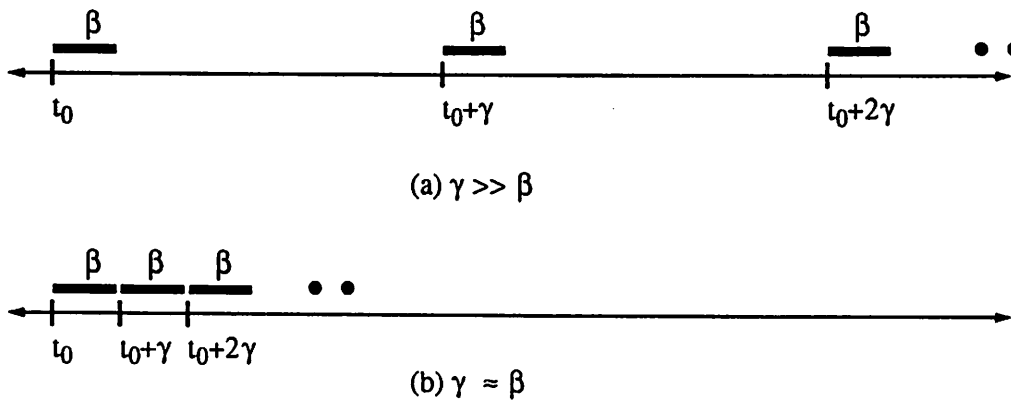
### **5.5.1 Time Slot Model**

For a given architecture, let  $\beta$  be the time needed to transmit a piece of data, and let  $\gamma$  be the minimum delay time between successive transmissions from a processor.  $\gamma$  is applicable when the multiprocessor system does not support an uninterrupted transmission of block data. This task is then realized in software as individual transmission enclosed in a loop. In this case,  $\gamma$  gives the minimum setup time

between each transmission. For a given data transfer on the architecture, let  $t_0$  be the starting time of the transfer, and let  $D$  be the amount of data to be transmitted. Assuming each transfer can send one piece of data, the data transfer occupies the following “time slot” intervals on the bus:

$$[t_0, t_0 + \beta] [t_0 + \gamma, t_0 + \gamma + \beta] \dots [t_0 + (D-1)\gamma, t_0 + (D-1)\gamma + \beta] \quad (\text{EQ 5.1})$$

$t_0$  is assumed to include any constant setup time. Note that  $\gamma$  is required to be larger or equal to  $\beta$  since a transmission cannot occur until the bus is free. In practice, this is often the case as time is required to set up the next data to be transmitted. When  $\gamma = \beta$ , the disjoint time slots merges into a single contiguous time slot  $[t_0, t_0 + D \cdot \beta]$ . In certain cases however, the value of  $\gamma$  can be quite large, especially when each transmission occurs at the end of each long iteration. In this case,  $\gamma$  represents the computation of an entire iteration of a loop. The two cases for  $\gamma \gg \beta$  and  $\gamma = \beta$  are shown in Figure 5.12.



**FIGURE 5.12: Communication Time Slots**

When the number of data transmissions is large, it can be quite time consuming to keep track and process each individual time slot, and it may be necessary to group the time slots into larger slots, or even into one contiguous time slot. For  $\gamma \gg \beta$ , merging the time slots into one time slot as  $[t_0, t_0 + D \cdot \beta]$  or  $[t_0, t_0 + D \cdot \gamma]$  would grossly misrepresent the activity on the bus. As a result, the McDAS system allows the user to

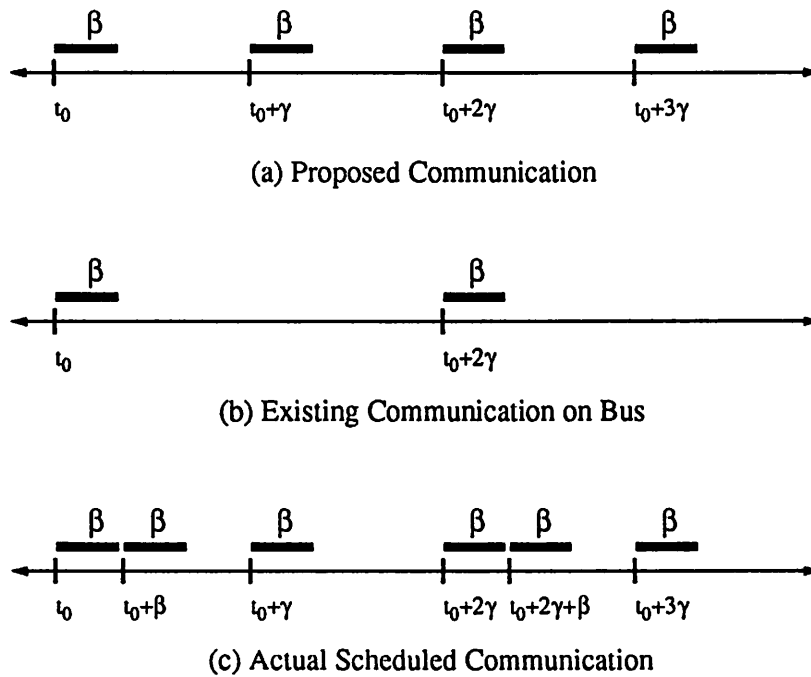
manipulate an *effort* parameter from the command window interface which will vary the abstraction of the time slot model. For a strong effort, each time slot is modelled, and for a weak effort, a number of time slots are merged together, trading-off accuracy for processing time.

In the scheduling algorithm to be discussed in the next chapter, interprocessor communications will be scheduled using the time slot model presented here. When scheduling a communication, the proposed time slots are constructed and merged with the time slots already allocated on the bus (due to previous communications), one at a time. At any point where a request conflicts with an already scheduled request (their time slots intersect), *bus congestion* occurs, and the bus arbitration mechanism of the target architecture must be used to resolve the conflict. The approach taken by the time slot model is to delay the transmission until the next available time slot on the bus, resulting in a delay in arrival time of the data.

An example of a time slot delay due to bus congestion is shown in Figure 5.13. The communication of Figure 5.13a is to be scheduled onto a bus with an existing allocation as shown in Figure 5.13b. The resultant communication schedule is shown in Figure 5.13c. The arrival time of the entire communication to the destination processor is  $t_0 + 3\gamma + \beta$

When the communication must make several bus hops to reach its destination, the time slot scheduling is performed one bus at a time. In other words, the time slots are scheduled on the first bus, and the results after scheduling are used as the starting time of the time slots on the second bus. The scheduling is complete once the time slots are scheduled on the last bus of the path.

We now define a number of parameters, ending with the *Earliest Starting Time* parameter, which is used by the scheduling algorithm. A key feature of the scheduling algorithm is that a node is only scheduled once all of its predecessor nodes have been



**FIGURE 5.13 : Bus Congestion Modeling**

scheduled. Hence, in deciding whether a node  $n_i$  should be scheduled on a processor  $p_k$ , it is necessary to examine the communications that must occur to bring all the input data needed by node  $n$  to processor  $p$ . These communication delays can be estimated using the time slot model above. All information needed is available at this point. The completion time of an input node (hence the starting time of the communication) and the source processor are known since the input node is already scheduled. The size of the data package is obtained from the edge linking the input node to node  $n$ . The destination processor is  $p$ , and the current state of the bus is obtained by careful bookkeeping of all previously scheduled communications.

**Definition 5.6:** The *Arrival time*  $t_{\text{arv}}(n_s, p)$  denotes the time at which data computed in a source node  $n_s$  is available at processor  $p$ . We assume the node  $n_s$  is already scheduled on a processor and the time slot model is used to schedule the data transfers on the appropriate bus or buses.  $t_{\text{arv}}(n_s, p)$  gives the time the last data package arrives at  $p$ .

**Definition 5.7:** The *Available time*  $t_{\text{avail}}(n, p)$  is the time at which all input data to node  $n$  is available at processor  $p$ . It is calculated over the set  $I(n)$  of all input nodes of  $n$  as:

$$t_{\text{avail}}(n, p) = \max \{ t_{\text{arv}}(n_i, p) \mid i \in I(n) \} \quad (\text{EQ 5.2})$$

**Definition 5.8:** The *Ready time*  $t_{\text{ready}}(p)$  is the time processor  $p$  has finished executing its last assigned node.

For a node  $n$  to start on a processor  $p$ , all of its input data must be available at processor  $p$ , and the processor must have completed any previously assigned computation.

**Definition 5.9:** The *Earliest Starting time*  $\xi(n, p)$  of node  $n$  on processor  $p$  is defined as:

$$\xi(n, p) = \max \{ t_{\text{avail}}(n, p), t_{\text{ready}}(p) \} \quad (\text{EQ 5.3})$$

$\xi(n, p)$  effectively abstracts the underlying architecture to the level of the starting times of nodes on processors. The scheduling algorithm is only concerned with this information to make its decisions, irrespective of the architecture. As a result,  $\xi(n, p)$  serves as the interface between the architecture-dependent estimations and the scheduling algorithm. This modularity allows the scheduler to deal with any architecture in a unified manner. Each architecture would only have to provide its own calculation of  $\xi(n, p)$ .

## 5.6 SUMMARY

In order to perform static multiprocessor scheduling, a clear understanding of the run-time behavior of the parallel program is required. This involves understanding the computation and interprocessor communication model of the program, as well as having accurate estimations of the computation and communication costs.

In this chapter, we presented a computation model composed of a number of subprograms executing together in a pipeline and parallel fashion. The architectural requirements to support such a model, in terms of interconnectivity, memory structure, and synchronizations, are discussed. To derive computation and memory costs of tasks, an estimation technique based on operator benchmarking and model construction is presented. The method is shown to be accurate to within 5% of the actual costs for the benchmarked architectures. Finally, a time-slot communication model is introduced to estimate interprocessor communication delays. The model allows the scheduler to explicitly schedule data transfers along with computations to accurately estimate their completion times. As a side effect, bus congestion is automatically taken into account.



# 6

## SCHEDULING

In this chapter, the scheduling algorithm is presented under two performance objectives: *Scheduling for Fixed Throughput*, and *Scheduling for Maximum Throughput*. Scheduling for fixed throughput performance is applicable in a real-time environment, where the sampling rate is dictated by the application. Scheduling for maximum throughput, on the other hand, is more appropriate for speeding up simulation. In Section 6.1, the scheduling problem under both objectives is formulated. Section 6.2 to 6.4 describes the fixed throughput scheduling algorithm. In Section 6.2, a preliminary algorithm is described, showing how both pipelining and parallel execution can be simultaneously considered. In Section 6.3, the algorithm is augmented with a path merging procedure to improve processor utilization. Section 6.4 extends the algorithm further to perform retiming when flowgraph cycles are present. A node decomposition technique to traverse granularity is discussed in Section 6.5. The advantage of combining node decomposition with pipelining and parallelism is also discussed. Finally, Section 6.6 presents the modifications necessary to adapt the algorithm to schedule for maximum throughput.

## 6.1 PROBLEM DEFINITION

In this section, an explicit formulation of the scheduling problem is given. To show how it differs from previous formulations, the deficiencies of previous approaches are summarized. A scheduling strategy to address these problems is then derived.

### 6.1.1 Problem Formulation

#### Scheduling for Fixed Throughput

The input to the scheduler consists of three components -- a control/data flowgraph  $G$ , a real-time throughput constraint, and an architecture description containing a processor count  $P$ , a buffer memory size  $BM$ , a computation time and memory cost model, and a communication cost model. The throughput constraint is expressed in terms of an available sample period. The goal is to obtain a non-preemptive compile-time schedule for a  $P$ -processor (or less) machine which meets the throughput constraint.

#### Scheduling for Maximum Throughput

The input to the scheduler consists of two parts -- a control/data flowgraph  $G$ , and an architecture description containing a processor count  $P$ , a buffer memory size  $BM$ , a computation time and memory cost model, and a communication cost model. The goal is to obtain a non-preemptive compile-time schedule for a  $P$ -processor machine which results in the highest throughput performance.

The CDFG represents a signal processing computation which must be repeated once for each input frame. The CDFG can be hierarchical, with primitive and hierarchical nodes. The iteration nodes are labelled as either "parallel" or "serial", depicting the data dependency between the iterations. This dependency was derived during the Silage to CDFG compilation. The nodes are assigned computation times and

memory requirements following the benchmarking procedures outlined in Chapter 5. The interprocessor communication model is based on the time slot bus reservation model also outlined in the previous chapter. The routines for calculating the Earliest Starting Times of nodes are assumed available for the target architecture. A processor count  $P$  completes the architecture description. The scheduler is restricted to be non-preemptive and static, a necessity for high throughput real-time implementations.

Like most problems in multiprocessor scheduling, the formulations above fall in the class of NP-complete problems. For this reason, algorithms which obtain optimal solutions are discarded in favor of heuristics which obtain a fairly good suboptimal solution in a reasonable time. We review a number of previous heuristics in the next subsection.

## 6.1.2 Previous Approaches

Previous DSP multiprocessor scheduling techniques suffer from a number of deficiencies which reduce their effectiveness or applicability to a wide range of applications. First of all, implicit to the CDFG description is the assumption is that the application belongs to the DSP domain. Hence, there is an infinite time loop surrounding the computation which allows the exploitation of temporal concurrency, i.e. the use of pipelining and retiming. This is not assumed in the classical multiprocessor scheduling formulation [Hu61], which is not DSP-based. The only form of concurrency that can be exploited in these problems is the parallelism concurrency. However, a number of techniques for DSP multiprocessor scheduling [Pri91][Sih90] still do not exploit temporal concurrency in any form. Instead, they minimize the completion time of a single execution of the application, and replicate the schedule to handle stream data. Secondly, most approaches [Yu84][Hwa89][Sih90][Sar89][Sch85] do not attempt to exploit the granularity levels of the flowgraph. As a result, the input flowgraph is usually flat instead of hierarchical. While these techniques can support

flowgraph nodes of different sizes, there is no attempt to decompose large nodes into smaller nodes to improve the quality of the resultant schedules. Thirdly, some techniques are only of theoretical value in that they don't allow for a resource availability constraints [Sch85][Lee85][Par89]. Finally, most approaches have a very simplified model of interprocessor communication [Bok88][Sar89], and some don't consider it at all [Sch85][Hu61]. Exceptions are found in [Yu84][Sih90][Hwa89] which schedules data transfers on buses to model delays due to bus congestion. However, all communications initiated by a node are modeled as occurring at the end of the node's computation. This is often not the way the actual communication will occur. Consider the case of a node containing a loop which is producing an array of data, one element in each iteration. It is better to send each piece of data as soon as it is available, instead of waiting till the end of the loop to send the entire array. This not only has the advantage of spreading the communication out to avoid bus congestion, but can also allow nodes which only need parts of the array to begin execution as soon as their segments are available. Grouping these separate communications into one slot can grossly misrepresent the activities on the bus, leading to inaccurate estimations of communication costs.

### **6. 1. 3 Our Scheduling Strategy**

The strategy is to attack these deficiencies on two fronts. On one front, it is crucial to provide the scheduler with as much information about the input application and the target architecture as possible. The result of this work is the hierarchical CDFG representation and the computation and communication models, which were discussed in the previous three chapters. The CDFG format stores all levels of hierarchy in the application in a efficient manner, allowing the scheduler to quickly traverse the different granularity levels available in the application. The computation model gives the scheduler precise computation times and memory requirements of nodes at any level of hierarchy, and lastly, the time slot communication model allows the scheduler to

accurately calculate the transfer time of data between processors, even for data which are sent while the source node is still executing. On the other front, a scheduling technique which can use all of the information provided to achieve a high quality solution is developed. The performance criteria is chosen to be the system throughput, which is deemed more important than other criteria in real-time DSP processing. The scheduling algorithm is discussed in greater detail in the remainder of the chapter.

## 6.2 SCHEDULING FOR FIXED THROUGHPUT

The algorithm considers temporal and spatial concurrency to obtain a schedule which simultaneously satisfies a system throughput requirement as well as processor and memory bounds constraints. Initially, a number of parameters, conditions, and bounds are introduced. Several examples are then analyzed to illustrate the key points of the algorithm. Finally, the algorithm is described in a formal way, and its complexity analyzed.

### 6.2.1 Definitions

**Definition 6.1:**  $W_{total}$  is defined as the computation time of the entire graph  $G$ .

**Definition 6.2:** The *stagetime*  $T$  is defined as the reciprocal of the throughput of the system. In a real-time formulation, it is also the sample period.

The stagetime equals the time allocated to each pipeline stage in the system, and thus to each processor in that stage. It is possible to derive some bounds on the stagetime given the total computation  $W_{total}$  and the number of processors  $P$ .

**Theorem 6.1:** An upper bound  $T_{ub}$  on the stagetime is  $W_{total}$ . A lower bound  $T_{lb}$  on the stagetime is  $W_{total} / P$ .

**Proof:** Since  $W_{\text{total}}$  represent the total amount of computation in the application, there is no need to allocate more time. Hence  $T_{\text{ub}} = W_{\text{total}}$ . With  $P$  processors, a perfect load balancing will give each processor  $W_{\text{total}} / P$  amount of work. Since scheduling may yield an imperfect load balancing, and extra time for interprocessor communication may be required, the load on the bottleneck processor is  $\geq W_{\text{total}} / P$ . Hence  $T_{\text{lb}} = W_{\text{total}} / P$ .

**Remark:** Since  $T_{\text{lb}}$  represents the lowest sample period achievable with  $P$  processors. The user-specify sample period can be checked against  $T_{\text{lb}}$  at the start of the scheduling to see if it is achievable.

**Theorem 6.2:** The computation cost  $W_{\text{max}}$  of the largest node in the CDFG (at a given level of granularity) is  $\leq T$ . This is termed the *Maximum Granularity Condition*.

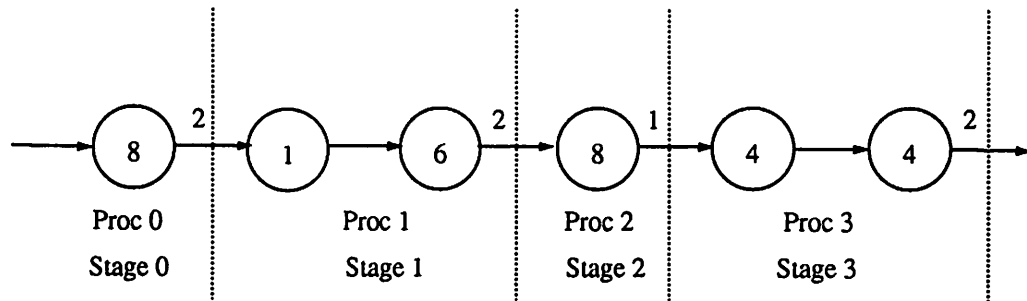
**Proof:** A node of computation time  $> T$  cannot be scheduled on a processor with only time  $T$  to execute.

Hence, after checking that  $T$  is feasible, we must go through a graph expansion phase where all hierarchical nodes whose computation times violate the maximum granularity condition are decomposed into smaller nodes. The node decomposition procedure is described in detail in section 6.4. For the remainder of the section, we will assume that we have a CDFG which satisfies the maximum granularity condition.

## 6. 2. 2 The Scheduling Appeal: Intuitive Description

Given the stagetime  $T$ , the scheduler traverses the CDFG from input to output, partitioning the graph into stages of pipelines. Nodes are scheduled onto a processor until the total computation costs of the nodes plus the communication cost of the output edges exceeds the stagetime  $T$ . Once a pipeline is filled, the scheduler proceeds to schedule the remaining nodes on the next pipeline stage. At the end, the graph is partitioned into a number of pipeline stages, and the number of processors needed is

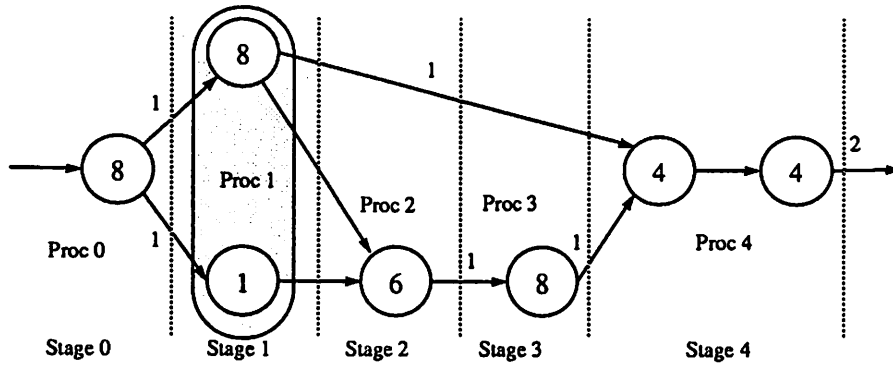
returned. An example of how the scheduler work on a simple serial CDFG is shown in Figure 6.1. Values inside the nodes represent estimates of their computation costs, and



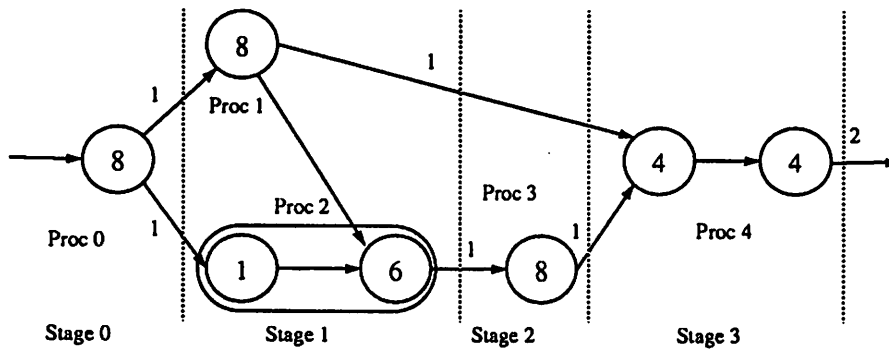
**FIGURE 6.1: Scheduling a serial graph with  $T = 10$**

values on the edges represent the additional delays for communication (for the sake of simplicity, the time-slot model is not used to calculate communication delays). When two nodes are assigned to the same processor, communications between them incur no cost.

For a serial graph like Figure 6.1, the node accumulation is straightforward. To schedule general graphs, it is necessary to resolve how to handle the parallelism available in branching paths. One algorithm would be to continue accumulating nodes to fill the stagetime, whether they are parallel or not. This would result in a schedule shown in Figure 6.2a. A more sophisticated algorithm would exploit the parallelism in the graph to yield a schedule shown in Figure 6.2b. This schedule uses the same number of processors but has a smaller latency and communication cost, and possibly even a better throughput. Exploiting parallelism while pipelining makes the scheduling task much more difficult. Since the number of processors is fixed, not all parallelism can be exploited, and the algorithm must decide which operations deserve extra processors and which do not. The naive approach, on the other hand, does not have to do this as it always puts one processor per pipeline stage. The exact criteria used for node scheduling is discussed in the next subsection.



(a) A naive approach



(b) A better approach

**FIGURE 6.2 : Scheduling a general graph with  $T = 10$** 

### 6.2.3 Node Scheduling

As illustrated in the above examples, the key goal in the scheduling for pipeline computation is to pack as many nodes into a pipeline stage as possible. This suggests a scheduling strategy which assigns a node to processors where it can have the earliest start time, taking into account communication delays, memory capacity, and processor availability. This requires exactly the information given by  $\xi(n, p)$ , the Earliest Starting Time of node  $n$  on processor  $p$ , as defined in chapter 5.

We propose a “list scheduling” algorithm which traverses the graph from input to output, assigning to every node a processor and a starting time, and to every processor a pipeline stage. At any point in the scheduling process, the scheduler keeps a list of ready nodes  $\mathcal{R}$ , and a list of available processors  $\mathcal{P}$ .  $\mathcal{R}$  contains all nodes

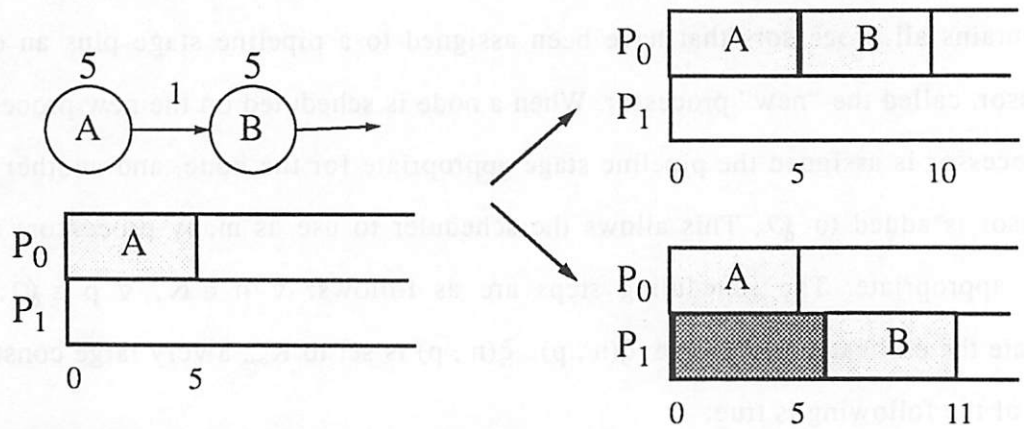


whose input nodes have been already scheduled. Initially, it contains all input nodes.  $\wp$  contains all processors that have been assigned to a pipeline stage plus an extra processor, called the “new” processor. When a node is scheduled on the new processor, the processor is assigned the pipeline stage appropriate for the node, and another new processor is added to  $\wp$ . This allows the scheduler to use as many processors as it deems appropriate. The scheduling steps are as follows:  $\forall n \in \mathcal{N}, \forall p \in \wp$ , we calculate the earliest starting time  $\xi(n, p)$ .  $\xi(n, p)$  is set to  $K_\infty$ , a very large constant, if any of the following is true:

1. There is insufficient buffer memory in processor  $p$  to execute node  $n$ , i.e.  $bm_{avail}(p) < bm(n, p)$ .
2.  $p$  was already assigned a pipeline stage which is different from the stage needed to execute  $n$ .
3. There is insufficient available time left in  $p$  to execute  $n$  within the stagetime limit.

A processor  $p$  for which  $\xi(n, p) < K_\infty$  is called a *feasible* processor for  $n$ . Condition 1 assumes the buffer memory is local to the processor. If it is in a centralized memory, the condition would be if  $bm_{avail}(P) < bm(n, p)$ , where  $bm_{avail}(P)$  is the remaining buffer memory of the entire system.

Consider the heuristic where at each scheduling step, the node  $n^*$  and processor  $p^*$  which minimizes  $\xi(n, p)$  is chosen for scheduling. By using the earliest starting time as a measure for scheduling, the communication cost is explicitly considered in the processor assignment. Take the simple CDFG as shown in Figure 6.3. Node A is already assigned to processor  $P_0$ , and it is now necessary to schedule node B. If B is assigned to  $P_0$ , it can start immediately after A terminates since there is no interprocessor communication. On the other hand, if B is assigned to  $P_1$ , there will be a communication cost, and the earliest starting time on  $P_1$  is 6 (again, the additive communication delay model is used for simplicity). In this way, nodes which



**FIGURE 6.3 : Earliest Starting Time Scheduling**

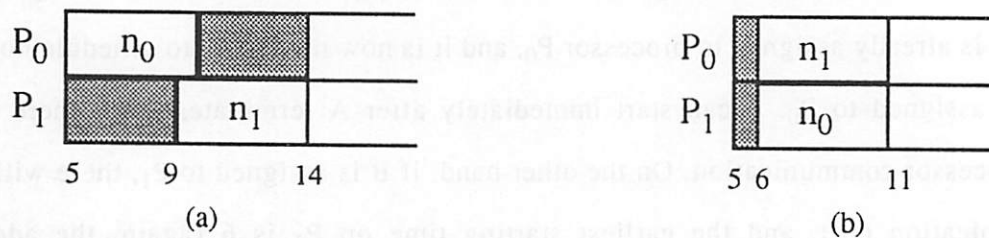
communicate heavily with each other are grouped in the same processor as much as possible to eliminate interprocessor communications.

This heuristic alone is not sufficient however, as it neglects other ready nodes which may vie for the same processors. Consider the case shown in Table 6.1. Both

	$w(n_i)$	$\xi(n_i, p_0)$	$\xi(n_i, p_1)$
$n_0$	5	5	6
$n_1$	5	6	9

**TABLE 6.1**

nodes  $n_0$  and  $n_1$  can execute on processors  $p_0$  and  $p_1$ , although they start at different times due to communication delays. If the minimum  $\xi(n, p)$  criteria is used, the resultant schedule is shown in Figure 6.4a. If the processor assignment is reversed, we obtain the schedule as shown in Figure 6.4b. As we can see, the second processor



**FIGURE 6.4 : Greedy scheduling**

assignment packs nodes tighter. The reason is that, although the earliest start time for node  $n_0$  is on processor  $p_0$ , scheduling it there takes that option away from node  $n_1$ . This forces  $n_1$  to be scheduled on  $p_1$ , where it starts much later. On the other hand, the penalty for scheduling  $n_0$  on  $p_1$  is far less. To solve this resource contention problem, the *Difference Measure*  $\delta(n)$  is introduced.

**Definition 6.3:** The *Difference Measure*  $\delta(n)$  is defined as:

$$\hat{\xi}(n) = \text{Min} \{ \xi(n, p) \mid p \in \mathcal{P} \} \quad (\text{EQ 6.1})$$

$$\hat{p}(n) = \{ p \in \mathcal{P} \mid \xi(n, p) = \hat{\xi}(n) \} \quad (\text{EQ 6.2})$$

$$\tilde{\xi}(n) = \text{Min} \{ \xi(n, p) \mid p \in \mathcal{P} - \hat{p}(n) \} \quad (\text{EQ 6.3})$$

$$\delta(n) = \tilde{\xi}(n) - \hat{\xi}(n) \quad (\text{EQ 6.4})$$

For a node  $n \in \mathcal{N}$ , Eq. 6.1 defines  $\hat{\xi}(n)$  to be the earliest starting time of node  $n$  among all available processors, and Eq. 6.2 defines  $\hat{p}(n)$  to be the processor which achieves  $\hat{\xi}(n)$ . Eq. 6.3 defines  $\tilde{\xi}(n)$  to be the earliest starting time of node  $n$  in the remaining processors.  $\hat{\xi}(n)$  is guaranteed to be  $< K_\infty$  since  $n$  can always execute on the new processor in  $\mathcal{P}$ .  $\delta(n)$  gives a measure of how good the best assignment  $\hat{\xi}$  is, compared to the second best  $\tilde{\xi}$ . A node  $n$  with a large  $\delta(n)$  says that the best assignment is much better than the second best, whereas a small  $\delta(n)$  says there exist comparable choices. Thus, it is more urgent to assign nodes with a large  $\delta(n)$ .

A candidate node and candidate processor are then chosen as follows: The node  $n$ ,  $n \in \mathcal{N}$ , corresponding to the largest  $\delta(n)$  is chosen as the candidate node, and the processor where it achieves its earliest starting time is chosen as the candidate processor. Formally, the candidate node  $n^*$  and processor  $p^*$  pair is given as:

$$n^* = \{ n \in \mathcal{N} \mid \delta(n) \text{ is maximum} \} \quad (\text{EQ 6.5})$$

$$p^* = \hat{p}(n^*) \quad (\text{EQ 6.6})$$

Nodes which have only one feasible processor assignment have their  $\tilde{\xi}(n) = K_{\infty}$ , so the largest  $\delta(n)$  yields the node that can start earliest among these. Thus this scheduling heuristic resorts to the previously mentioned earliest starting time heuristic when there exist ready nodes with only one possible processor assignment. Only when all ready nodes have at least two feasible processors does  $\delta(n)$  enter to choose among processors. If the earliest start time of a node is the same on both the new processor and another already used processor, the used processor is chosen. Thus, a new processor is only chosen in a scheduling step for a node  $n_i$  when this node can start earlier on the new processor than in any existing feasible processor.

Table 6.2 gives the earliest starting time  $\xi(n_i, p_k)$  of three nodes  $n_0, n_1, n_2$ , on three available processors  $p_0, p_1, p_2$ . The  $\hat{\xi}(n)$ ,  $\tilde{\xi}(n)$ , and  $\delta(n)$  values are also shown. The earliest starting node is  $n_0$  on processor  $p_0$ . However, the candidate node chosen is  $n_2$  on  $p_1$  since its alternative choice,  $p_2$ , is a lot worse.  $\xi(n_2, p_0) = K_{\infty}$ , signifying that  $p_0$  is not a feasible processor for  $n_2$ .

	$w(n_i)$	$\xi(n_i, p_0)$	$\xi(n_i, p_1)$	$\xi(n_i, p_2)$	$\hat{\xi}(n_i)$	$\tilde{\xi}(n_i)$	$\delta(n_i)$
$n_0$	4	4	8	12	4	8	4
$n_1$	5	7	16	10	7	10	3
$n_2$	3	$K_{\infty}$	6	12	6	12	6

TABLE 6.2

Once a candidate pair is chosen, the scheduled node is removed from  $\mathcal{N}$ , and new ready nodes are added. If the new processor in  $\mathcal{P}$  was used, another new processor is added to  $\mathcal{P}$ . Processors assigned to pipeline stages which are no longer considered are removed from  $\mathcal{P}$  to avoid unnecessary computations.  $\xi(n, p)$  and  $\delta(n)$  values that are affected by the assignment are updated, and the next node-processor pair is chosen. The scheduling algorithm ends when all nodes are scheduled. In the example above, scheduling  $n_2$  on  $p_1$  would push back the ready time of  $p_1$  to 9, affecting all  $\xi(n_i, p_1)$  values  $< 9$ , and hence those  $\delta(n_i)$ .

The pseudo-code for the scheduling algorithm is described below:

```

Main (Graph, T) {
    Assign Computation times and Memory usage to all nodes in Graph;
    Calculate  $W_{total}$ ,  $W_{max}$ ,  $T_{ub}$ ,  $T_{lb}$ ;
    Check feasibility of stagetime T;
    ExpandGraph(Graph,  $W_{max}$ );
    Schedule (Graph, T);
}

Schedule (Graph, T) {
    Input nodes  $\Rightarrow \mathfrak{N}$ ,  $p_0 \Rightarrow \emptyset$ ;
    Repeat while  $\mathfrak{N} \neq \emptyset$ 
        For each  $n \in \mathfrak{N}$ ,  $p \in \emptyset$  do
            Calculate  $\xi(n, p)$ ;
            Calculate  $\delta(n)$ ;
        Schedule candidate  $n^*$  and  $p^*$ ;
        Update  $\mathfrak{N}$ ,  $\emptyset$ ;
}

```

## 6.2.4 Complexity Analysis

The complexity of the algorithm is  $O(N(N+E))$ , and is derived as follows: Assume that the total number of nodes is  $N$ , the total number of edges is  $E$ , and the maximum number of processors is  $P$ . Assume that the calculation of  $t_{arv}(n, p)$  takes  $O(1)$  time (which is true if a single contiguous time slot is used to estimate each communication in the bus scheduling model). Given a fixed processor  $p_k$ , for each new node  $n$  put in  $\mathfrak{N}$ , the calculation of  $\xi(n, p_k)$  requires calculating  $t_{arv}(n, p_k)$  for each input edge. For the whole graph, all edges are visited, for  $O(E)$  calculations. Adding and removing nodes to and from  $\mathfrak{N}$  takes  $O(N)$  time. Hence the total computation runs in  $O(N+E)$  time. Since  $P$  processors are considered for each node, the complexity is  $O(P(N+E))$ . Finally, for nodes in  $\mathfrak{N}$  which were not chosen in a scheduling step,  $\xi(n_i, p^*)$  is updated in constant time to reflect the new  $t_{ready}(p^*)$ . Since there are at most  $N$  such nodes in  $\mathfrak{N}$ , at most  $N$   $\xi(n, p)$  updates are made each scheduling step. For  $N$  scheduling steps,  $N^2$  calculations are required. The total complexity is  $O(P(N+E)+N^2)$ .

Since the algorithm can continue to add processors beyond  $P$ , the potential number of processors considered can be  $N$ , the number of nodes. Hence the final complexity is  $O(N(N+E))$ .

## 6.3 PATH MERGING

To exploit using parallel execution along with pipelining, the scheduling algorithm tries to assign nodes to different processors whenever parallelism is available. In its greedy attempt to maximize parallelism, the algorithm may under-utilize a processor. In this section, this problem is investigated in detail, and an approach to improve the processor utilization is presented.

### 6.3.1 Problem Definition

Parallelism emerges when a path splits into two or more parallel paths, and ends when the two paths join. Parallel paths are characterized by branch and join nodes, defined as follows:

**Definition 6.4:** A *Branch* node is a node which has two or more successor nodes. A *Join* node is a node which has two or more predecessor nodes.

It is common for the scheduling algorithm to assign different processors to the successor nodes of a branch node. Consider the branching path shown in Figure 6.5.

Assume  $T = 20$ . The schedule after processing nodes  $A$  and  $B$  is shown in Figure 6.5a. Node  $C$  can start at time 10 on processor  $P_0$  and time 6 on a new processor  $P_1$ . Hence, it is scheduled on  $P_1$ . This decision is made regardless of the structure of the flowgraph afterwards. This greedy heuristic may lead the algorithm to assign very little computation to a processor. This would be the case if, for example, the flowgraph in Figure 6.5 continues as shown in Figure 6.6. Nodes  $D$  and  $E$  are assigned to processor

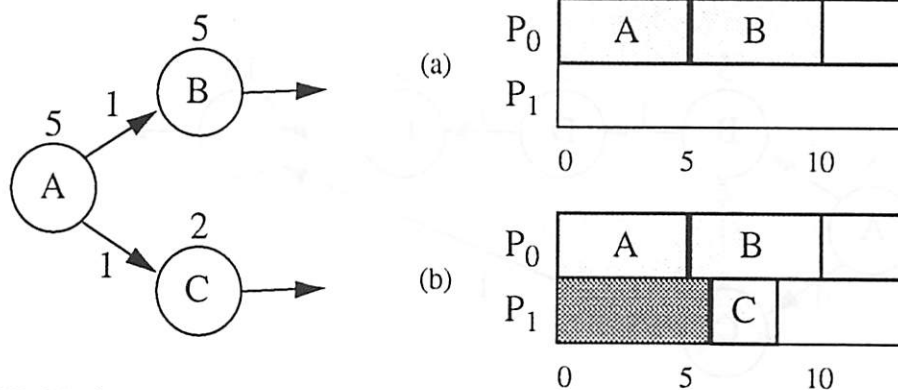


FIGURE 6.5: Scheduling of Branching Path

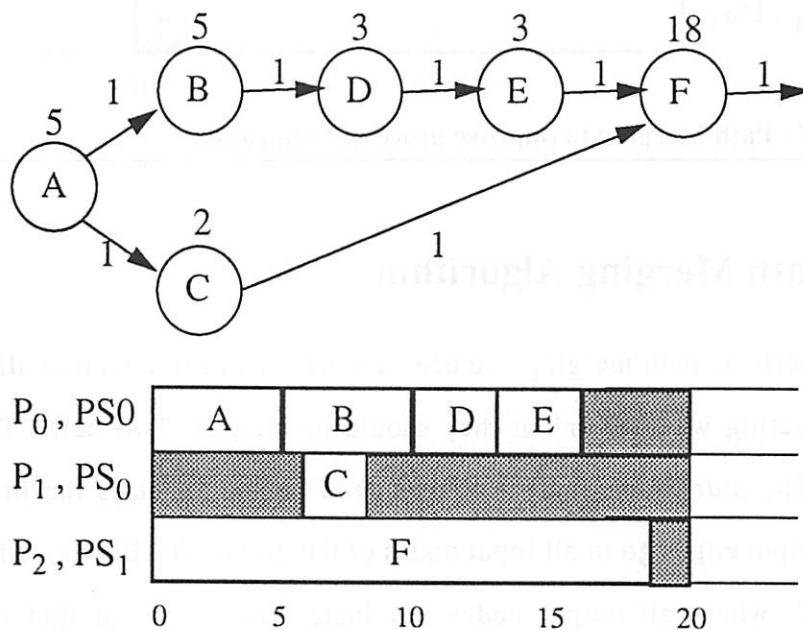
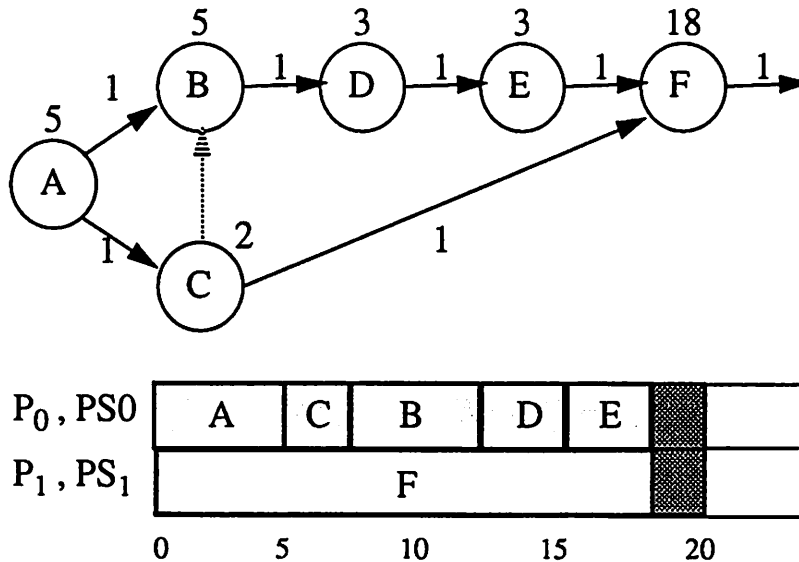


FIGURE 6.6 : Under-utilized Processor Assignment

P<sub>1</sub>, minimizing interprocessor communications. Node F is assigned to processor P<sub>2</sub> on the next pipeline stage, leaving processor P<sub>1</sub> under-utilized.

To solve this problem, parallel paths can be *merged* together so they are forced to execute on one processor. This is achieved by adding a dependency edge from one end of one path to the beginning of the other, serializing the communication. In the above example, adding a dependency edge from node C to node B yields a schedule which

only uses two instead of three processors. The resultant CDFG and schedule is shown in Figure 6.7.



**FIGURE 6.7 : Path Merging to improve processor utilization**

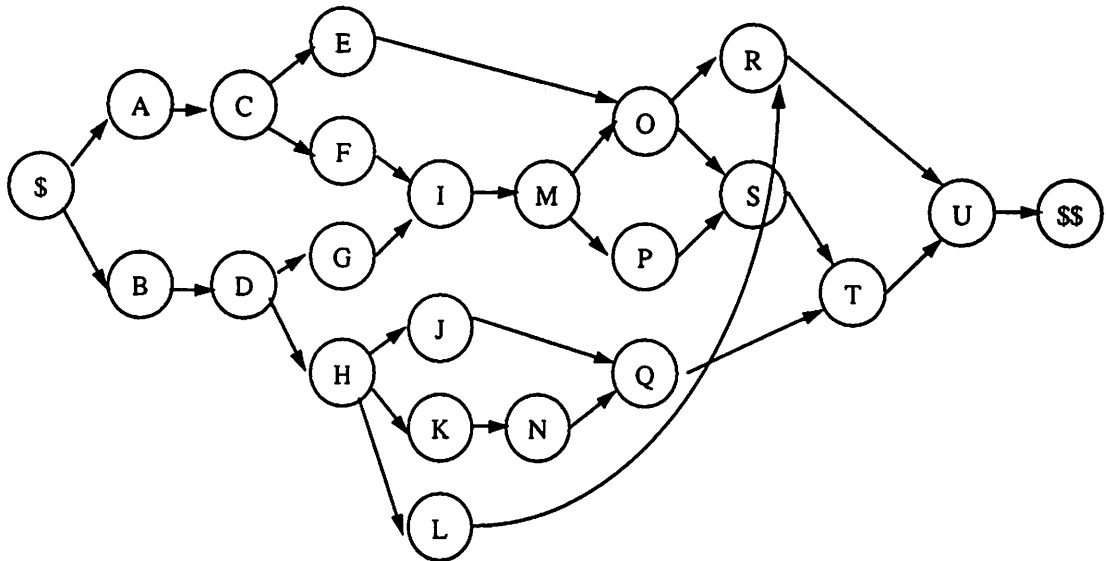
### 6.3.2 Path Merging Algorithm

To perform path merging, we need a mechanism for detecting all parallel paths and for evaluating whether or not they should be merged. Two new CDFG nodes are introduced: The *Start* node and the *End* node. The Start node is the first node in the CDFG. Its output edges go to all input nodes of the graph. The End node is the last node in the CDFG, where all output nodes terminate. The Start and End nodes have no behavior associated with them, and are only introduced to simplify the detection of parallel paths. With these nodes, all parallel paths can be found by examining only branch nodes. The Start node serves as a branch node for all input nodes, while the End node serves as a join node for all output nodes.

**Definition 6.5:** The *Overlap Path Time*  $t_{op}$  between two parallel paths measures the time both paths are simultaneously active. This is equivalent to the elapsed time between the branch node of the path and the point where the two paths join again. The introduction of the End node assures that all paths eventually join.



$t_{op}$  is the minimum of the computation costs of the two paths, and it measures the amount of parallelism that exists between the paths. The smaller the  $t_{op}$ , the less the parallelism that can be exploited. Figure 6.8 shows a CDFG and the overlap path time of some of its paths. All nodes have costs 1. The Start and End nodes are labelled as \$ and \$\$ respectively.



SAMPLE PATHS	$t_{op}$
ORU-OSTU, MOS-MPS, CEO-CFIMO, HJQ-HKNQ	1
HJQTU-HLRU, HKNQTU-HLRU, DGIMOR-DHLR	2
\$ACFI-SBDGI, ACEO-BDGIMO, DGIMPST-DHJQT	3
DGIMPST-DHKNQT	4

**FIGURE 6.8 : Overlap Path Time**

and \$\$ respectively.

The successor nodes to a branch node are called *head* nodes, for the obvious reason that they are the starting nodes of the parallel paths. In the above graph, nodes J, K, and L are head nodes of branch node H. The join node of two parallel paths is chosen to be the first common descendant node of the two corresponding head nodes. To facilitate the calculation of  $t_{op}$ , the *transitive closure* [Sed88] information of the CDFG is used to quickly locate the join node. The transitive closure of a graph G is a graph  $G^*$

where there is an edge  $e_{ij}$  connecting node  $n_i$  and  $n_j$ , if there is a path from node  $n_i$  to node  $n_j$  in the original graph  $G$ . This is an  $O(n^3)$  algorithm, where  $n$  is the number of nodes. For quick access to the data dependency information, an *adjacency matrix* representation of the flow graph is used. The calculation of  $t_{op}$  is as follows: The nodes in the CDFG are leveled from output using their computation costs as weights. Given two head nodes  $n_j$  and  $n_k$ , the join node  $n_{jk}$  is given as the node with the highest output level which is still a descendant of both  $n_j$  and  $n_k$ . The computation on the path from a head node to the join node is given by the difference in their output levels. The  $t_{op}$  of two paths is then given as the minimum of the path computations. Given the transitive closure matrix, and the output levels  $ol(n_i)$ , an  $O(N^2)$  algorithm for calculating  $t_{op}$  of parallel paths is as follows:

```

OverlapPathTime (Graph) {
  For each branch node  $n_i \in \text{Graph}$  do
    For each pair of nodes  $(n_j, n_k)$  which are successor nodes to node  $n_i$  do
      Locate join node  $n_{jk}$  of  $(n_j, n_k)$  as the node with the highest output level
        which is a descendant node to both  $n_j$  and  $n_k$  ;
       $t_{op}(n_j, n_k) = \text{MIN}[ol(n_j), ol(n_k)] - ol(n_{jk})$ ;
  return minimum  $t_{op}(n_j, n_k)$ ;
}

```

The heuristic for path merging is as follows: The pair of parallel paths with the smallest  $t_{op}$  is the best candidate for merging as it contains the least amount of parallelism. If these two paths were assigned to different processors, one of the processors is probably greatly under-utilized. The smallest  $t_{op}$  is returned in the overlap path time calculation above, and a routine  $\text{MergePath}(\text{Graph})$  adds a dependency edge from one end of the path to the beginning of the other.  $t_{op}$  is then updated taking the new dependency into account.  $\text{MergePath}()$  returns FALSE when there is no more parallel paths to merge. The path merging heuristic is used to continually modify and reschedule the CDFG. Each time  $\text{MergePath}()$  is called, the current path with the smallest  $t_{op}$  is merged, eliminating another possible point for the under-utilization of processors. The  $\text{Schedule}()$  routine is repeated along with  $\text{MergePath}()$  as long as more

processors are needed then are available and as long as there are still parallel paths to merge. The overall algorithm, called Partition(), is as follows:

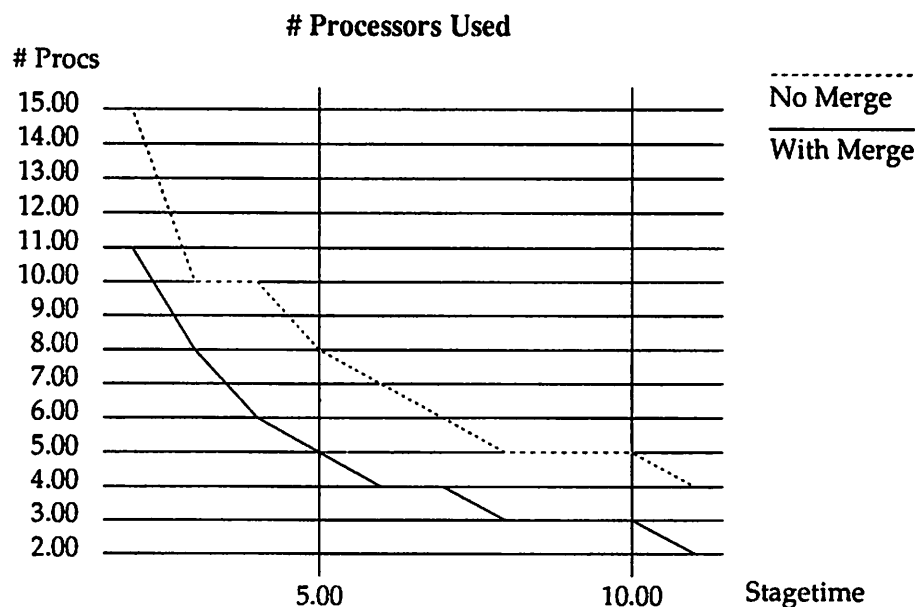
```

Partition (Graph, T) {
  proc = Schedule(Graph, T);
  Repeat while (proc > P and MergePath(Graph) == TRUE)
    Update Transitive Closure Graph, OutputLevel;
    proc = Schedule(Graph, T);
}

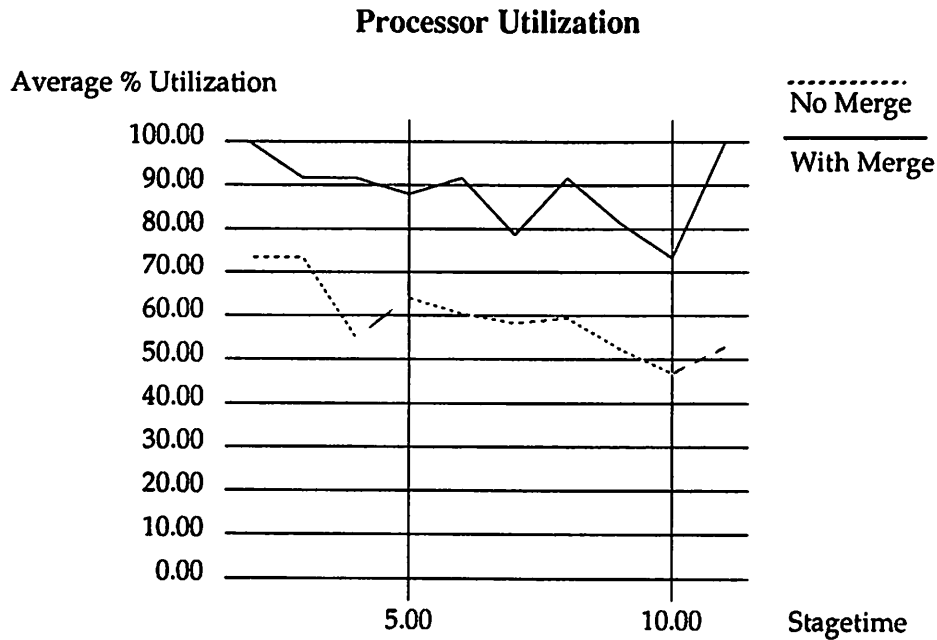
```

Upon exit of the loop, we either have a feasible schedule using  $P$  or less processors, or an infeasible schedule ( $\text{proc} > P$ ) which uses the minimum number of processors for the given stagetime  $T$ . For this latter case, a longer stagetime is needed or more processors have to be allocated.

Figure 6.9 shows the reduction in the number of processors used when path merging is applied to the example of Figure 6.8. Figure 6.10 shows the corresponding improvement in average percent processor utilization.



**FIGURE 6.9 : Reduction in Processors due to Path Merging**



**FIGURE 6.10 : Improvement in Processor Utilization due to Path Merging**

### 6.3.3 Complexity Analysis

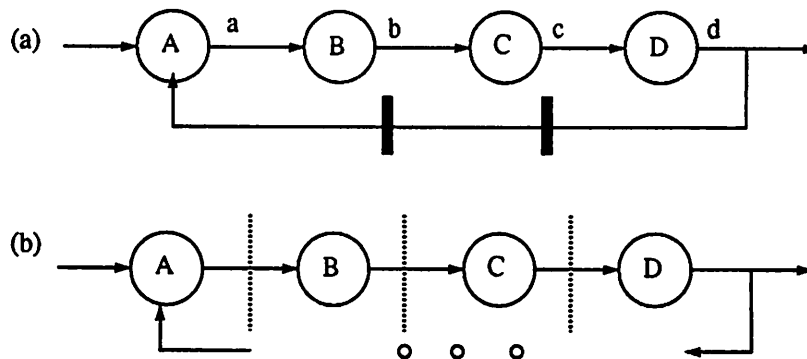
The calculation in the inner loop involves updating  $t_{op}$ , finding the minimum  $t_{op}$ , adding a dependency edge, and scheduling the graph. Updating  $t_{op}$  involves updating the output levels and the transitive closure graph. To update the output levels, only the source node of the newly added dependency edge and its ancestors need to have their output levels recalculated. This takes  $O(N)$  time. In the transitive closure graph, an edge has to be added between the source node and its ancestors to each descendant of the destination node. This takes  $O(N^2)$  time. Locating the minimum  $t_{op}$  takes  $O(N^2)$ , and scheduling takes  $O(N(N+E))$  time. This inner loop repeats at most  $N$  times since there are at most  $N-1$  merges possible. Hence, the algorithm runs in  $O(N^2(N+E))$  time. For most examples that we have encountered,  $E \approx N$ , giving an  $O(N^3)$  approximation.

## 6.4 RETIMING

The scheduling algorithm presented so far works well for flow graphs which do not contain cycles. For those that do, some enhancements to the algorithm are needed. In this section, a modification to the algorithm is presented to allow the retiming of flow graph cycles to improve scheduling.

### 6.4.1 Problem Definition

Consider a simple flow graph with feedback as shown in Figure 6.11. There are two sample delay operators on the feedback path. In order to execute node A, it is necessary to have the data on edge d from two samples back in time, which we will denote as  $d@2$ , following the Silage convention. If the scheduling algorithm is applied



**FIGURE 6.11 : Retiming Flow graphs**

to the flow graph without considering the feedback path, it may pipeline the computation as shown in Figure 6.11b. Since node D is now 3 samples behind node A,  $d@2$  will not be available when node A needs it. Thus it is necessary to limit the number of pipeline stages on the forward path to ensure that proper data is available when it is needed. Referring to Figure 6.11a, assume node A is applied to sample 0 at time 0. Since the sample period is the stagemtime  $T$ , node A will be applied to sample 2 at time  $2T$ . To execute this node, the output data of node D on sample 0 must be available.

This implies that the execution of the nodes A, B, C, D on sample 0 (indeed, on all samples) must complete within time  $2T$ . In general, if  $\Delta_c$  is the number of delays in the cycle, then the cycle must execute within time  $\Delta_c T$ . The following definitions formalize this bound on the execution of cycles.

**Definition 6.6:** Let  $G$  be the flowgraph at some granularity level, and let  $C$  be any cycle in  $G$ . Let:

$\Delta_c \equiv$  the number of delays in cycle  $C$

$W_c \equiv$  the total computation time of cycle  $C$ .

$E_c \equiv$  the total time a valid schedule would need to execute cycle  $C$ .

**Theorem 6.3:** For any cycle  $C$ ,

$$W_c \leq E_c \leq \Delta_c T \quad (\text{EQ 6.7})$$

**Proof:** The proof for  $E_c \leq \Delta_c T$  is an obvious generalization of the argument above. Without loss of generality, we can assume that all  $\Delta_c$  delays are grouped together. Let  $D$  be the last node before the delay, and let  $A$  be the first node after the delay. Let  $d$  be the output of node  $D$ . At time 0, node  $A$  processes sample 0. At time  $\Delta_c T$ , node  $A$  needs data  $d$  from node  $D$  for sample 0. Thus, the execution time of the cycle,  $E_c$ , must be done by this time.  $W_c \leq E_c$  because a schedule may not necessarily execute cycle  $C$  contiguously.

With this result, we get the following condition, called the *Cycle Scheduling Bound* condition.

**Theorem 6.4:** Let  $n_0, n_1, \dots, n_N$  be the set of nodes in cycle  $C$ , and let  $n_0$  be the first scheduled node among them. Let  $t_{\text{start}}(n_0)$  be the starting time of node  $n_0$ , and  $t_{\text{done}}(n_k)$  be the completion time of any node  $n_k$  in the cycle. A valid schedule must satisfy:

$$t_{\text{done}}(n_k) - t_{\text{start}}(n_0) \leq \Delta_c T \quad (\text{EQ 6.8})$$

**Proof:** This follows directly from the fact that  $t_{\text{done}}(n_k) - t_{\text{start}}(n_0) \leq E_c$ , for every node  $n_k$  in the cycle. Theorem 6.4 will be used by the scheduling algorithm to ensure the Cycle Scheduling Bound is satisfied when nodes of a cycle are scheduled.

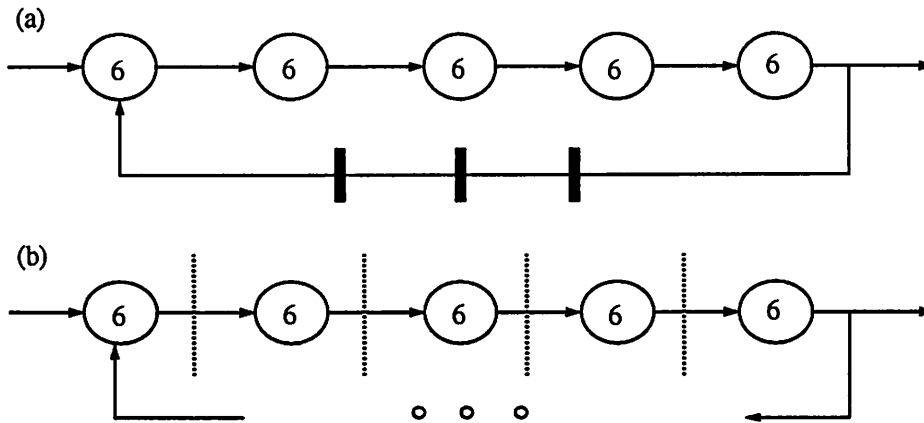
**Theorem 6.5:** From Theorem 6.3, for every cycle  $c$ ,  $T \geq W_c / \Delta_c$ . In particular,  $T \geq T_{\text{CB}}$ , where  $T_{\text{CB}}$  is the *Stagetime Cycle Bound* given by:

$$T_{\text{CB}} = \text{Max}_c W_c / \Delta_c \quad (\text{EQ 6.9})$$

$W_c$  is a function of the granularity level of the graph. As the granularity gets finer, the computation times of cycles decrease as hierarchical nodes in the cycles are replaced by only those nodes of the subgraph which belong to the cycle. In this case,  $T_{\text{CB}}$  also decreases. Some cycles are entirely imbedded in hierarchical nodes, and are only uncovered when these nodes are expanded. For that case,  $T_{\text{CB}}$  may actually increase. Once all cycles are uncovered, however,  $T_{\text{CB}}$  decreases monotonically to a bound known as the *Iteration Period Bound*  $T_{\text{IPB}}$  [Sch85], the stagetime cycle bound for the finest granularity graph. This bound is the minimum achievable latency between sample iterations, and is a theoretical lower bound for our solution.

When the scheduler makes a partition on the forward path, it is in effect putting a logical delay there. To maintain correct functionality of the flowgraph, it is necessary to remove a delay in the feedback path. This can be interpreted as moving delays in the cycle around to maximize our throughput, a concept known as retiming [Lei83]. By choosing  $T \geq T_{\text{CB}}$ , one might conjecture that sufficient time is allocated to each pipeline stage to guarantee that at most  $\Delta_c$  partitions are made in scheduling cycle  $c$ , thereby automatically adhering to the *Cycle Scheduling Bound* on the nodes of the cycle. This, unfortunately, is only guaranteed if the nodes in the cycle are scheduled contiguously, the stagetime is exploited completely, and communication delays are not considered. In practice however, data transfers are often present, and using the stagetime completely is difficult due to the granularity of the nodes. An

example of a cycle scheduling violation due to large granularity of the nodes is shown in Figure 6.12.



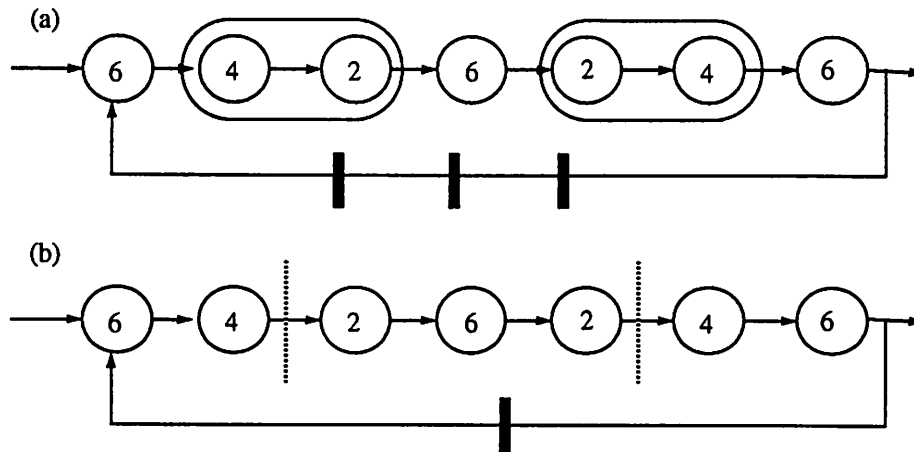
**FIGURE 6.12: Cycle bound violation.**

From Figure 6.12a, the stagetime cycle bound  $T_{CB}$  is found to be  $30/3 = 10$ . Using a stagetime  $T = T_{CB} = 10$ , we find that we need 4 partitions, which violates the Cycle Scheduling Bound condition (Figure 6.12b). The weight of the nodes are too big to fit in the remaining available time, leaving holes or *slacks* in the stagetime. If the nodes are broken down into smaller nodes, as in Figure 6.13a, a feasible partition with no slacks can be obtained (Figure 6.13b). Communication costs are not considered in this example.

## 6.4.2 Retiming Algorithm

Using the analysis above, the scheduling algorithm presented so far can easily be modified to handle cycles. From Theorem 6.5, the stagetime lower bound  $T_{lb}$  is modified to  $\text{Max}(W_{\text{total}} / P, T_{IPB})$  to include the theoretical lower bound due to cycles. Each time a node  $n_i$  in a cycle is scheduled, Theorem 6.4 is used to ensure that the *Cycle Scheduling Bound* for the node is satisfied. If it is violated, the large nodes in the cycle are decomposed in an attempt to minimize slacks in the stagetime. This is repeated until





**FIGURE 6.13: Node Decomposition to satisfy cycle bound condition.**

there is no more violation or no more nodes in the cycle can be decomposed. This processing step is called `CyclePartition()`, and is given as:

```

CyclePartition (Graph, T) {
  (proc, ViolateFlag) = Partition (Graph, T);
  Repeat while (ViolateFlag == TRUE and IsGraphFlat(Graph) == FALSE)
    Graph = Expand nodes in critical cycles;
    Find all cycles in Graph;
    Update  $W_{max}$  and  $T_{CB}$ ;
    (proc, ViolateFlag) = Partition (Graph, T);
}

```

Note there that  $T_{CB}$  is updated each time a node is decomposed. The overall algorithm is modified to be:

```

Main (Graph, T)
  Calculate  $W_{total}$ ,  $W_{max}$ ,  $T_{CB}$ ,  $T_{ub}$ ,  $T_{lb}$ ;
  Check feasibility of stagetime T;
  ExpandGraph(Graph,  $W_{max}$ );
  CyclePartition (Graph, T);
}

```

The inner-most `Schedule()` algorithm is modified to perform the Cycle Scheduling Bound check as follows:

```

Schedule (Graph, T) {
  Input nodes  $\Rightarrow \mathcal{N}$ ,  $p_0 \Rightarrow \emptyset$ ;
  Repeat while  $\mathcal{N} \neq \emptyset$ 
    For each  $n \in \mathcal{N}$ ,  $p \in \emptyset$  do
      Calculate  $\xi(n, p)$ ;
      Calculate  $\delta(n)$ ;
    Schedule candidate  $n^*$  and  $p^*$ 
    if (Cycle Scheduling Bound(Graph) == Violated) return(FALSE);
    Update  $\mathcal{N}$ ,  $\emptyset$ ;
}

```

### 6.4.3 Complexity Analysis

In CyclePartition(), Partition() is called repetitively on finer and finer granularity flow graphs to satisfy the cycle scheduling bound. When the node decomposed in the cycle is an iteration, each decomposition increases the number of processors allocated to the iteration by 1, and terminates when the number of processors reaches  $P$ . Thus, Partition() repeats at most  $P$  steps when decomposing iterations. When decomposing functions, it may repeat more, and there is no maximum bound. Benchmarks show that the typical number of function node decompositions is less than five. The main computations in the CyclePartition() routine is the search for all cycles in the CDFG, and the Partition() routine. For cycle detection, we use Johnson's algorithm for finding all the elementary cycles of a directed graph [Joh75], which is the fastest algorithm known. It has a time of complexity of  $O((N+E)(C+1))$ , where  $C$  is the number of cycles in the graph, and a space complexity of  $O(N+E)$ . Note that the cycles can overlapped with each other, and hence when a node is scheduled, the cycle scheduling bound condition must be checked for all cycles for which the node is a member of. From the last section, Partition() was analyzed to run in  $O(N^2(N+E))$  time. Hence, the running time of CyclePartition() is  $O(P(N+E)[N^2 + (C+1)])$ .

## 6.5 NODE DECOMPOSITION

Node decomposition is the means by which the scheduling algorithm traverse from a coarse grain flowgraph to a finer grain flowgraph. In previous subsections, we saw that nodes are broken up on two occasions: When they are larger than the available stagetime  $T$ , and when they are part of a cycle that violates the cycle scheduling bound. The first case is denoted as *bottleneck node decomposition* and the second case as *critical cycle decomposition*. So far in the section, we have discussed **when** to decompose nodes, now we concentrate on **how** the decomposition is done.

Presently, there are two types of hierarchical nodes that are candidates for decomposition: *Function nodes* and *Iteration nodes*. Function nodes are decomposed by replacing the nodes with their subgraphs. Iteration nodes are replaced by parallel or serial sub-nodes (depending on the data dependencies between iterations), each computing a subset of the iteration range. The method of breaking up the iterations differs between the decomposition of bottleneck nodes and the decomposition of critical cycles. In both cases, the size of the sub-nodes is determined *dynamically* during the scheduling procedure.

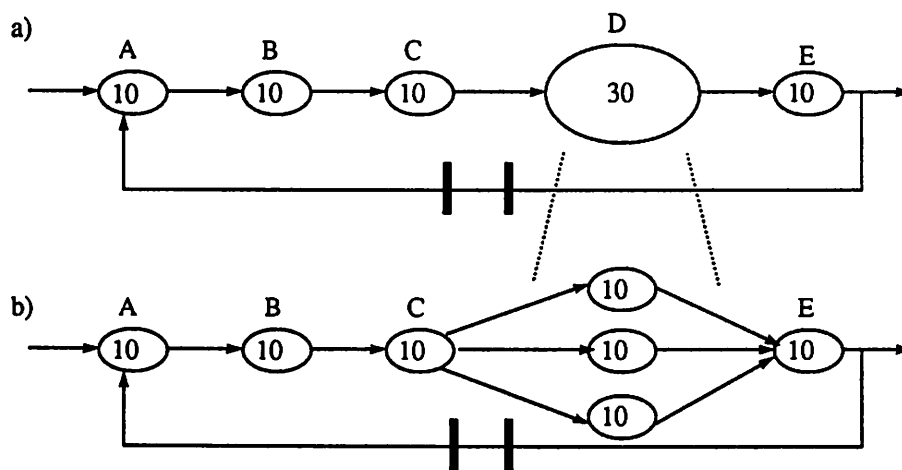
### 6.5.1 Bottleneck Node Decomposition

When decomposing bottleneck iteration nodes, the sub-nodes adapt to the available time remaining in the stage. Thus, the number of iterations assigned to the sub-nodes are not fixed until the first sub-node is scheduled. At that time, the first sub-node is assigned as many iterations as can fit in the remaining stage, and subsequent sub-nodes are assigned as many iterations as can fit in a new, empty stage with stagetime  $T$ . This partitioning strategy allows the iterations to float across processors from one pass of the scheduler to the next to fit the changing stagetime  $T$ .

## 6.5.2 Critical Cycle Decomposition

When decomposing iteration nodes in critical cycles, the goal is to obtain a partition which satisfies the *cycle scheduling bound* condition. For nodes with serial dependency, decomposition does not decrease the cycle computation time, but may improve the stagetime slacks. Therefore, the same technique as discussed above is used. For nodes with parallel dependency, decomposition actually decreases the computation time of the cycle, making it easier to meet the *cycle scheduling bound*.

Consider a cycle containing a parallel iteration node as shown in Figure 6.14.



**FIGURE 6.14** Decomposing parallel iterations in cycles.

Assuming a tight stagetime  $T = T_{CB} = 35$  and no communication delays, scheduling Figure 6.14a with  $t_{start}(n_A) = 0$  would yield  $t_{done}(n_E) = 2*35+10 = 80$ . Since  $t_{start}(n_A) + 2T = 70$ , the cycle scheduling bound is violated. In Figure 6.14b, after decomposition, the stagetime cycle bound is reduced from 35 to 25.  $T$  is no longer tight, and scheduling yields  $t_{done}(n_E) = 1*35+20 = 55$ ,  $t_{start}(n_A) + 2T = 70$ , meeting the cycle scheduling bound condition. Of course, the price we pay is the 2 additional processors needed to execute the iterations in parallel. Note that the naive scheduling algorithm described in

Figure 6.2a would not be able to improve the stagetime at all, as the parallelism of the iteration is not exploited.

The goal in the parallel iteration node case is to parallelize the node just enough to satisfy the cycle scheduling bound condition, as excessive parallelizing would only use more processors than necessary. To accomplish this, the parallel iteration node is incrementally divided into equal weight sub-nodes until a partition with no violation or the maximum decomposition is reached. Since the cycle scheduling bound is tightest on the sub-node with the maximum computation, a division into equal weight sub-nodes maximizes the chance of meeting the bound.

Currently, loops can only be divided at the boundary of each iteration. No attempt will be made, for instance, to partition 3.5 iterations of a loop on one processor and the remainder on another.

## **6.6 SCHEDULING FOR MAXIMUM THROUGHPUT**

The formulation of the scheduling algorithm from sections 6.2 to 6.5 completes the formulation of a scheduling strategy for fixed throughput implementations. In this section, the scheduling for maximum throughput problem is addressed. Its main application is the minimization of simulation time on a target architecture. Since there is no throughput constraint to meet, only those constraints imposed by the target architecture (number of processors, amount of memory, communication bandwidth) are still present.

### **6.6.1 Bounded Search Heuristic**

To maximize the throughput of the schedule, we have to minimize the stagetime  $T$  and hence, the time allotted to each processor. This is done by performing a

bounded search using the CyclePartition() routine introduced earlier as a probing function. First, the upper and lower bounds on the stagetime are determined. A candidate stagetime  $T$  is chosen between the two bounds, and checked for feasibility using CyclePartition(). This routine, as a side-effect to scheduling the flowgraph, returns the number of processors that are needed to execute the flowgraph given a stagetime  $T$ . By comparing the number of processors required to the actual number of processors that are available, we can adjust the stagetime  $T$  appropriately and re-partition. This iterative refinement process terminates with the minimum feasible stagetime. This search strategy will be discussed in greater detail.

Just as in the scheduling algorithm for real-time implementation, the upper and lower bounds on the stagetime are calculated. However, there is no throughput feasibility check to perform as there are no performance constraint to satisfy. Similarly, the granularity expansion of the CDFG so that the bottleneck node  $W_{\max} \leq T_{\text{proposed}}$  is no longer applicable as there is no fixed  $T_{\text{proposed}}$ . To ensure that nodes are not expanded more than they have to be, the proposed algorithm will start at the top level of hierarchy, and will systematically decompose nodes only when necessary. The *Maximum Granularity Condition* still applies, and limits how low  $T_{\text{proposed}}$  can go in any iteration. But more importantly, it plays a crucial role in determining when a node should be decomposed. The pseudo-code for the algorithm is shown below:

```

Main (Graph) {
    Assign Computation times and Memory usage to all nodes in Graph;
    Calculate  $W_{\text{total}}, W_{\text{max}}, T_{\text{ub}}, T_{\text{lb}}, T_{\text{CB}}$ ;
     $W_{\text{max}} = \text{MaxWeight}(\text{Graph});$ 
     $T_{\text{CB}} = \text{CycleBound}(\text{Graph});$ 
     $T = \text{Max}(T_{\text{CB}}, (T_{\text{ub}} + T_{\text{lb}})/2, W_{\text{max}});$ 
    Repeat while (  $T_{\text{lb}} < T_{\text{ub}}$  )
        CyclePartition (Graph, T);
        if (proc == P)
            Graphoptimal = Graph;
        if (proc > P)
             $T_{\text{lb}} = T;$ 

```

```

if (proc ≤ P)
  Tub = T;
  if (T == Wmax)
    Graph = Expand bottleneck nodes with weights Wmax;
    Update Wmax and TCB;
  if (T == TCB)
    Graph = Expand nodes in Critical cycles;
    Update Wmax and TCB;
  T = Max(TCB, (Tub + Tlb)/2, Wmax);
}

```

The major modification to the previous algorithm is the outer loop to determine the minimum stagetime and the systematic node decomposition strategy based on available processors. The CyclePartition() routine schedules the CDFG with stagetime  $T$  and returns the number of processors required. If this number is greater than the number of processors available, we increase  $T$  to give each processor more time. If it is less or equal, we decrease  $T$  so that more processors are used. This is achieved by updating  $T_{ub}$  or  $T_{lb}$  with the current  $T$ . By always picking the next  $T$  between these two values, we guarantee convergence to the minimum feasible stagetime.

When decreasing  $T$ , care must be taken to ensure that the Maximum Granularity Condition is not violated. When there are nodes in the graph with weights as large as  $T$ , and the search decides that  $T$  can be decreased further, it will break up these nodes to find a better solution. These nodes are called *bottleneck* nodes. In this way, large granularity nodes are only decomposed if they block the stagetime minimization process. This keeps the number of nodes in the graph at a minimum.

The proposed stagetime is bounded below at all times by the bottleneck nodes and by the critical cycle bound, forcing the search to follow a pattern as shown in Figure 6.15. The stagetime  $T_i$  represents the proposed stagetime at each iteration, and  $G_i$  represents the corresponding CDFG. At first, the proposed stagetime decreases monotonically as it continues to be feasible. As it decreases, the flowgraph is continually transformed into finer and finer granularity. At a sufficiently fine



**FIGURE 6.15: Bounded Search Pattern**

granularity level, there are no longer any bottleneck nodes or cycles, and the stagetime alternates in a binary search pattern. From this phase to the end of the search, the flowgraph remains at the same granularity level.

## 6.6.2 Complexity Analysis

The outer loop of the algorithm is essentially a binary search which takes  $O(\log_2 (W_{total}))$  time. Each proposed stagetime involves calling `CyclePartition()`, which takes  $O(P(N+E)[N^2 + (C+1)])$ . Hence, the complexity of the total algorithm is  $O(P(N+E)\log_2 (W_{total})[N^2 + (C+1)])$ . Although the bound is high, actual running times on examples have been quite fast. This is due to the fact that the scheduler only decomposes hierarchical nodes when it has to, restricting the number of nodes actually processed to a minimum.

## 6.7 SUMMARY

A multiprocessor scheduling algorithm, which simultaneously exploits pipelining and parallelism has been presented. To be able to exploit parallelism in addition to pipelining means that at any given pipeline stage, the algorithm must decide whether to use one or more processors working in parallel. Since the number of processors is fixed, not all parallelism can be exploited, and the scheduler must decide which node deserves extra processors and which does not. The approach taken is to exploit parallelism in a greedy fashion and iteratively improve the solution via a path merging step. Results show that path merging is able to significantly improve processor



utilization. To handle applications with feedback cycles, the scheduling algorithm is enhanced to allow the retiming of flowgraph cycles. The conditions for a retiming violation is derived and reduced to a single test statement during scheduling.

The algorithm can also exploit the hierarchical representation of the CDFG to arrive at the most suitable granularity for scheduling. The ability to simultaneously consider the many types of concurrency plus the ability to traverse the flowgraph at different granularity levels allows the scheduler to maximally exploit the potential concurrency of an algorithm. Previous approaches tend to concentrate on a specific concurrency type to exploit, restricting their domain to a narrow class of applications possessing this concurrency. Some features addressed by these methods are functional pipelining [Bok88], loop pipelining [Gir87], functional parallelism [Sih89], data parallelism or data partitioning [Pri91]. By combining concurrency with granularity, the scheduler is able to use all of the techniques above, and any combination of them in a unified manner. The number of available processors dictate the granularity of the concurrency to be exploited. At a high level, functional pipelining and parallelism is used. At a lower granularity, loop pipelining and data parallelism can be exploited. The search automatically guides the scheduler to the most appropriate technique for the application and processor count. This feature allows the scheduler to find efficient multiprocessor schedules for a wide range of DSP applications.

# 7

## CODE GENERATION

In this chapter, a code generation strategy for multiprocessor machines is presented. The main difference over the code generation for uniprocessor machines is the additional task of allocating memory for interprocessor communications and generating synchronizations for proper data transfer. Section 7.1 gives an overview of the code generation strategy in McDAS for both real-time implementation and simulation speedup. The two main components in the code generator are outlined: The memory mapper and the code emitter. In Section 7.2, the memory mapper module is discussed. Techniques to allocate memory for interprocessor communications are described for centralized-memory and distributed-memory systems, as well as for message passing architectures. Interprocessor synchronizations are also discussed as part of the overall interprocessor communication strategy. The code emission algorithm is presented in Section 7.3. In particular, it describes how interprocessor communication instructions are generated, and how delayed signals in DSP are implemented. It also presents a mechanism for performing both floating-point and fixed-point simulations from the same emitted C code. Finally, the section concludes with a discussion on the direct generation of DSP code from a flow graph specification.

## 7.1 OVERVIEW

The McDAS code generator takes as input a scheduled CDFG, a target architecture, a command file, and generates executable code for each processor. The CDFG should be decorated with the scheduling information as determined by the scheduler. Specifically, each node at the top level of hierarchy of the CDFG should be assigned a processor, a pipeline stage, and its execution order in the processor. If any such node is hierarchical, all nodes in its subgraph are assigned to the same processor. The execution order of the subgraph nodes will be determined by the code generator, based on data dependency. The command file is used by the code generator in simulation mode. It gives information on the number of samples to simulate, where to find the input signal files, what signals to display and where to store their values.

The organization of the code generator is shown in Figure 7.1. The three main components of the code generator are the *Front End Parser*, the *Memory Mapper* and the *Code Emitter*. The current code generator generates C code for the Sequent multiprocessor system. Other target machines may be supported by modifying a number of routines in the Code Emitter.

The front end parser is responsible for reading in the CDFG and the command file. The schedule on the CDFG is checked to make sure that all nodes assigned to the same processor are assigned to the same pipeline stage and have a unique execution order in the processor. Using the information from the command file, edges which correspond to input or display signals are annotated with their respective files. This information will be used by the code emitter later on to generate the necessary file I/O operations. Next, the memory mapper takes the scheduled CDFG and derives a memory layout for the target multiprocessor. This information is decorated on the CDFG by annotating those edges which represent interprocessor communications and those edges which require interprocessor synchronizations. This phase will be described in detail in

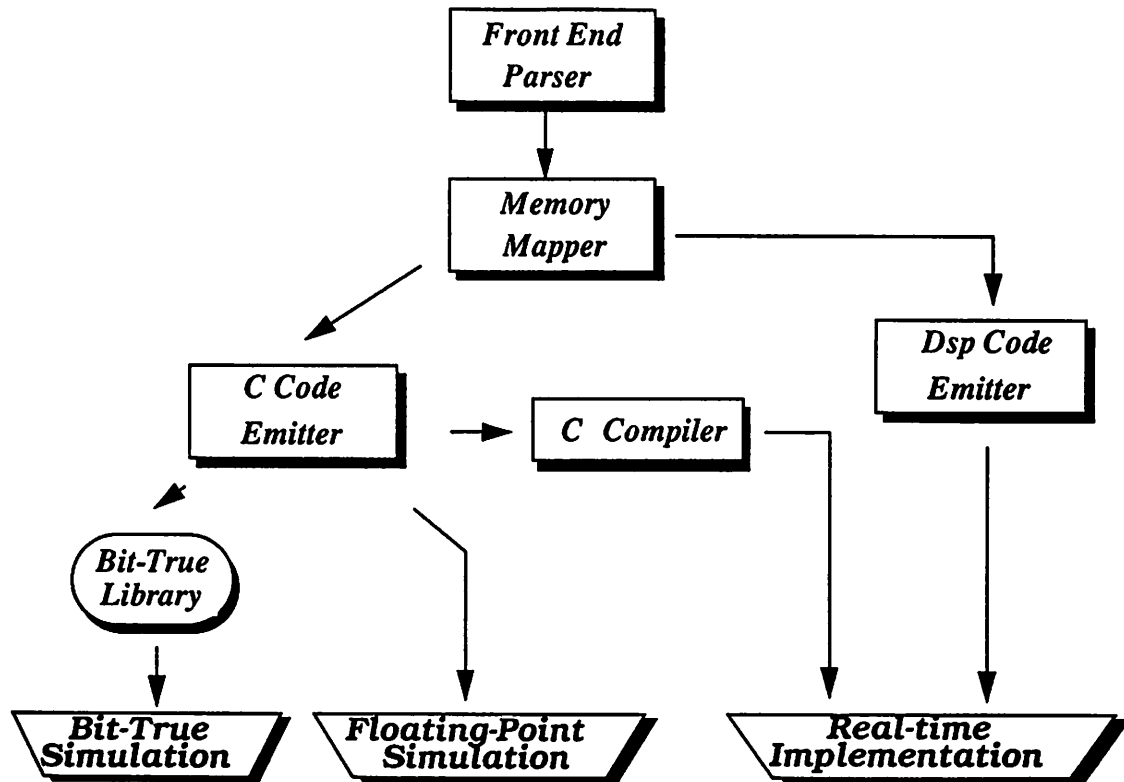


FIGURE 7.1: Code Generator Overview

the next section. After memory mapping, the code emitter module takes the decorated CDFG and generates the required code. For simulation, the C code emitter is used. The same C code generated can implement both bit-true and floating-point simulations. In case of bit-true simulation, a library of bit-level arithmetic and logic routines is available to be linked with the C code. If real-time implementation is desired, DSP assembly code is generated. This is obtained by compiling the C code or by direct generation from the CDFG. Currently, McDAS uses commercial C compilers to generate code for real-time implementations. The direct generation of optimized DSP code from a flowgraph description (Dsp Code Emitter) is another approach. It is crucial for high performance applications as the code generated by the C compilers is not well optimized. Section 7.3 describes several research projects currently underway to tackle this problem.

## 7.2 MEMORY MAPPING

A key element in the compilation of programs for multiprocessors is the layout of the memory to be used for interprocessor communication. For each transaction, memory must be allocated and synchronization steps must be inserted to ensure the proper delivery of data from the source to the destination processor. The pseudo-code for the memory mapper phase is:

```
MemoryMapper(Graph) {
    Cluster all nodes assigned to one processor into subgraphs;
    Derive local synchronizations between processors where necessary;
    Layout and allocate memory for interprocessor communications;
}
```

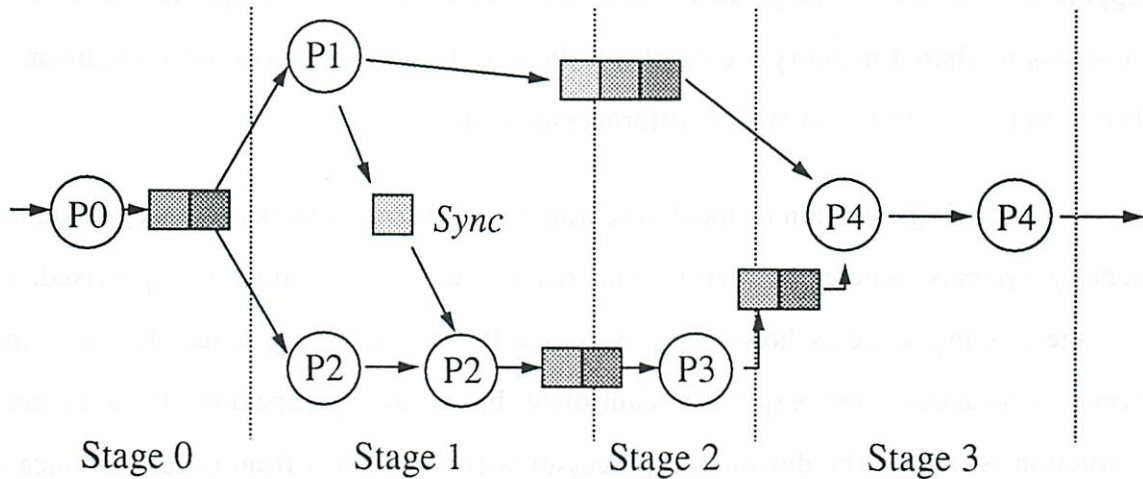
The memory mapper starts out by clustering the nodes in the CDFG into subgraphs according to their processor assignments. Each subgraph then defines the computation to be performed on the assigned processor. The pipeline stage assignment of the processor is derived from the common pipeline assignment of its nodes.

The edges which connect the subgraphs are those edges whose input and output nodes are assigned to different processors. In other words, they are *buffer* edges representing interprocessor communications. The second phase determines if any local synchronization, in addition to the global barrier synchronization done at the beginning of each sample execution (see Section 5.1), is necessary. The last phase of the memory mapper module analyzes these edges in order to allocate the required memory to support the transfers. These last two phases are discussed in detail in the remainder of the section.

### 7.2.1 The FIFO Communication Model

The underlying model for interprocessor communications in McDAS is based on FIFO queues. The use of FIFOs to support the passing of data is discussed in the

BLOSIM system [Mes84]. A queue is assigned to each interprocessor communication. The size of the queue is equal to  $PS(P_{dest}) - PS(P_{source}) + 1$ , where  $PS(P)$  is the Pipeline stage of processor  $P$ . The storage capacity of each data block on the queue corresponds to the amount of data sent, which can be derived from the buffer edge. The FIFO queues for the CDFG on Figure 6.2b are shown in Figure 7.2.



**FIGURE 7.2 : FIFO queues allocation**

Each processor consumes a data block on each of its input queues and produces a data block for each of its output queues. A pointer keeps track of the current free data block. The source processor always writes to the free data block while the destination processor always reads from the oldest block. If all processors globally synchronized at the beginning of each sample period, this scheme allows the source processor to send data to the destination processor sample after sample without ever corrupting any unread data. Hence, no other synchronizations are needed in general.

### 7.2.2 Local Synchronization

When two processors in the same pipeline stage need to communicate, the length of the FIFO is one. Explicit synchronization is therefore necessary to ensure that the destination processor will only read the data block *after* the source processor has

written to it. This is also illustrated in Figure 7.2. This is termed *local synchronization* to differentiate it from the global barrier synchronization. The local synchronization is needed on each buffer edge whose source and destination subgraphs belong to the same pipeline stage. These edges are specially marked so they will be recognized by the code emitter later on. If the synchronizations of arbitrary subgroups of processors are supported by hardware, they can be used. Otherwise, software semaphores which rely on access to shared memory are employed [Pet85]. This synchronization mechanism is slower, but is supported in most multiprocessor systems.

Explicit generation of local synchronization instructions is necessary in shared memory systems, where the writing and reading of memory are not supervised. In message passing systems however, this is usually not necessary since the send and receive instructions are explicitly controlled by hardware. Specifically, a receive instruction issued by the destination processor will only return from execution once it has received the new data from the send instruction.

Note that if it was possible to perfectly predict the computation and communication times exactly and if it was possible to invoke execution of a node on the exact time as calculated by the scheduler, no local synchronizations would be necessary as the nodes are only scheduled after all input data have arrived. Unfortunately, since such accurate estimations are not available, the synchronizations are necessary to guarantee correct execution.

The FIFO model shields the code generator from the underlying communication mechanism of the target multiprocessor. In the next two subsections, we discuss how the FIFO's are implemented on shared memory and message passing systems.

### 7.2.3 Shared Memory Implementation

A shared memory multiprocessor system can have two types of memory organization: Centralized shared memory, and distributed shared memory. Both are capable of supporting the FIFO model, although they differ in memory usage and performance efficiency. A centralized shared memory is the most memory efficient, but its main limitation is the memory access bottleneck. A distributed shared memory can involve replicating data, but does not suffer from memory accesses conflicts as much.

#### 7.2.3.1 Centralized Shared Memory

In a centralized shared memory system, all FIFO's reside in the centralized shared memory, and can be accessed by both source and destination processors. Only one copy of each FIFO is necessary. The algorithm for allocating memory to the FIFO's is straightforward in this case:

```

CentralMemoryLayout(BufferEdgeList) {
  Let Ptr points to the beginning of the centralized shared memory;
  For each edge in BufferEdgeList do
    QueueLength =  $PS(P_{dest}) - PS(P_{source}) + 1$ ;
    QueueSize = size of each data block * QueueLength;
    Allocate memory section [Ptr , Ptr + QueueSize] to edge;
    Update Ptr to point to Ptr + QueueSize;
}

```

Table 7.1 shows the centralized memory allocation for the scheduled CDFG of Figure 7.2. In the table,  $E_{ij}$  represents the buffer edge from processor  $i$  to processor  $j$ , and  $Data(E_{ij})$  gives the amount of data in bytes of each communication. Assume  $E_{01}$ ,  $E_{02}$  represent a broadcast, and  $E_{12}$  and  $E_{14}$  are separate writes. The FIFO length and memory segments are calculated using the algorithm above. Note that only one copy of the broadcast data  $E_{01}$ ,  $E_{02}$  is needed.

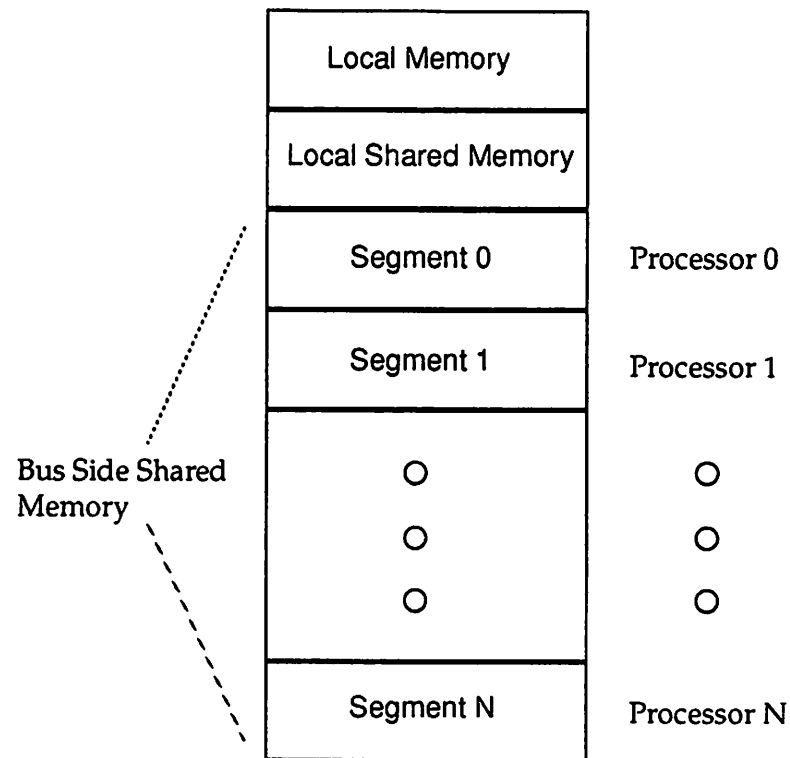


$E_{ij}$	Data( $E_{ij}$ )	QueueLength	Memory Segment
$E_{01}, E_{02}$	8	2	0 - 15
$E_{12}$	12	1	16 - 27
$E_{14}$	16	3	28 - 75
$E_{23}$	8	2	76-91
$E_{34}$	16	2	92-123

TABLE 7.1

### 7.2.3.2 Distributed Shared Memory

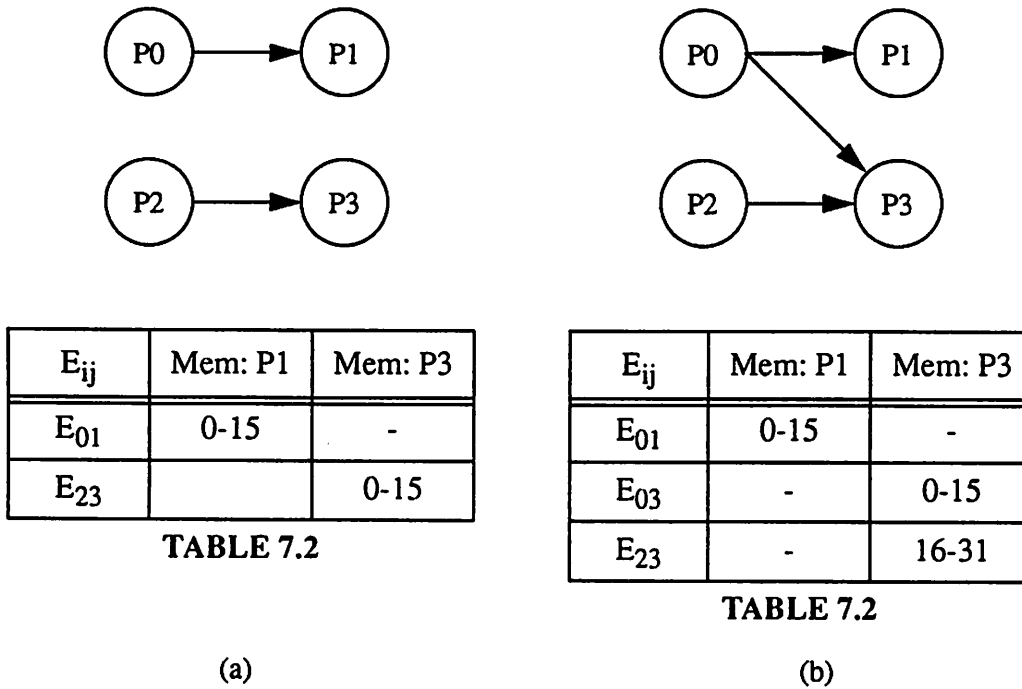
In a distributed shared memory system, the FIFO's are placed in the destination processors's section of shared memory, and communication is done via a global-write-local-read scheme where the source processor sends data through the network and writes it directly into the destination processor's section of shared memory. The destination processor can then read the data locally without accessing the network. Figure 7.3 shows an example layout of a distributed memory address space, taken from the SMART system [Koh89]. This system is described in more detail in the next chapter. Each processor has its own local memory to store private data and program which are not shared among processors. The local shared memory is used for accessing (reading) any shared data. This access is carried out through a local bus independent of the interconnection network. The bus side shared address space, which is assigned for the communication through the network, is partitioned into equal memory segments and distributed among processors. The physical memory of each segment corresponds to the physical memory of the associated processor's local shared memory address space. Data, which is written to segment  $i$ , can be read from the local shared memory of processor  $i$ , thereby achieving the interprocessor communication. When data is broadcasted with a single instruction, the same data is written to the same address in each destination processor's local shared memory. Distributed memory



**FIGURE 7.3: Distributed Memory Address Segmentation**

systems can reduce bus conflicts in general. However, for broadcast data, all destination processors must have their own copy of the FIFO, as opposed to a single copy in a centralized memory system.

The algorithm for allocating memory segments to the FIFO's is more involved as each processor's local shared memory must be optimized individually. Consider the communication patterns illustrated in Figure 7.4. Assume each communication takes 16 bytes, and the local shared memory starts at address 0. In case (a), since the communications are independent, it is possible to allocate the same memory addresses to both destination processors 1 and 3, yielding a mapping shown in Table 7.2a. In case (b) however, the broadcast of processor 0 to both processors 1 and 3 claims segment 0-15 on both processors. The memory allocation of  $E_{23}$  is therefore pushed to 16-31.



**FIGURE 7.4: Memory Allocation for Distributed Memory**

The memory allocation problem is then formulated as follows: Given a number of interprocessor communications, some of which may be broadcasts, derive a memory allocation so that the memory allocated on each processor's local shared memory is  $\leq M$ , a specified constant. The challenge lies in the fact that communications between exclusive groups of processors can use the same address, but broadcast communications place the constraints that the same addresses must be reserved on all destination processors. This problem is an instance of the bin-packing problem with the extra constraints imposed by the broadcasts. Bin-packing belongs to the class of NP-complete problems for which there is no known efficient solution [Gar79]. Since  $M$  is typically large, it makes more sense to use a greedy but fast heuristic to find a feasible mapping as oppose to exerting our efforts to minimize the memory allocated. We proposed the following heuristic:

```
DistributedMemoryLayout(BufferEdgeList) {
    Sort the BufferEdgeList so edges with the highest broadcast have the
    highest priority;
```

```

For each edge in BufferEdgeList with highest priority do
  QueueLength = PS( $P_{dest}$ ) - PS( $P_{source}$ ) + 1;
  QueueSize = size of each data block * QueueLength;
  Let  $\wp$  = set of destination processors of edge;
  Ptr = beginning of local shared memory - 1;
  while (TRUE) do
    Let Ptr = minimum address of unallocated memory > Ptr in  $\wp$ ;
    if (Ptr to Ptr + QueueSize is free on all processors in  $\wp$ )
      break;
    Allocate segments Ptr to Ptr + QueueSize on all processors in  $\wp$ ;
  }

```

The algorithm treats those edges which have the most broadcasts first as they impose the most constraints. For each communication, the first feasible memory segment on all destination processors is chosen. If  $E$  is the number of buffer edges, and  $P$  is the number of processors, the time it takes to find a feasible mapping for each edge is at most  $O(EP)$  since each segment is checked against at most  $P$  processors, and at most  $E$  segments are proposed. The complexity of the entire algorithm is  $O(E^2P)$ .

Table 7.3 shows the distributed memory allocation for the scheduled CDFG of

$E_{ij}$	Size( $E_{ij}$ )	Mem: P1	Mem: P2	Mem: P3	Mem: P4
$E_{01}$	16	0-15	-		
$E_{02}$	16	-	0-15		
$E_{12}$	12	-	16 - 27		
$E_{14}$	48	-	-		0-47
$E_{23}$	16	-	-	0-15	
$E_{34}$	32	-	-		48-79

TABLE 7.3

Figure 7.2. The layout is optimum, with no holes in the allocation.

## 7.2.4 Message-Passing Implementation

In a message-passing system, each processor sees only its own private memory, and communications must occur by data transfers. Since no address space is shared, all shared data must be explicitly copied and sent. This makes message-passing systems identical to the distributed shared bus system as far as memory layout is concerned. As a result, the algorithm described in the previous subsection is used here as well. The FIFO resides in the private memory of the destination processor, and *send* and *receive* constructs are used carry out the data transfer. All that is required at the source processor side is the destination processor identification in the *send* instruction, while all that is required at the destination processor is a FIFO to store the data.

## 7.3 CODE EMISSION

Once the scheduled CDFG are partitioned into subgraphs and the memory layout is analyzed, the code emitter is used to generate code for each respective processor. In this section, we describe the interprocessor communication mechanism, the state variable maintenance mechanism, and the C code emission algorithm. The section concludes with a discussion on the direct generation of DSP code from a flowgraph specification.

### 7.3.1 Circular Buffering

Two issues must be addressed before code generation can occur. They are the bookkeeping of interprocessor communications, and the access and updating of state variables. An addressing scheme based on circular buffers can be used to solve both problems.

### 7.3.1.1 Interprocessor Communication

From the memory layout step, enough memory has been allocated to implement each interprocessor communication. The only problem which remains is an addressing scheme which is systematic enough to be implemented automatically. This is possible if we implement the FIFO as a circular buffer, and use modulo addressing to generate the addresses for both the source and destination processors.

The address calculation is dependent on 3 variables: An address offset  $d$ , a current buffer index  $i$ , and the size of the buffer  $m$ , and is calculated as follows:

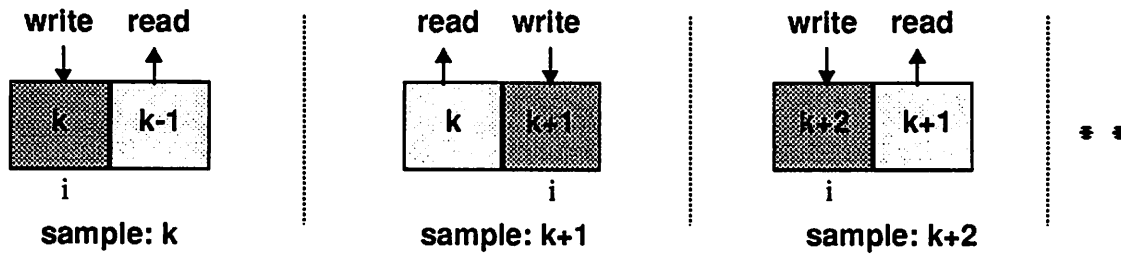
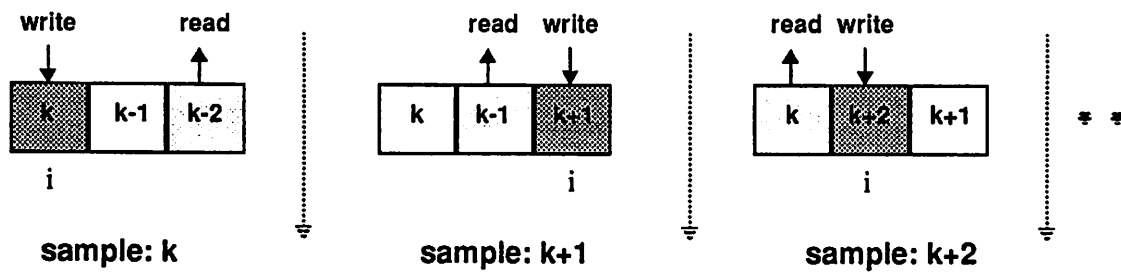
$$\text{addr}(d, i, m) = (i+d)\%m \quad (\text{EQ 7.1})$$

where ‘%’ represents the modulus operator which produces the remainder when  $(i+d)$  is divided by  $m$ . To implement the FIFO, both source and destination processors manage their own copy of the buffer index  $i$ , which they both initialize to the same value, usually 0, and both decrement their respective values by one after each sample iteration. The address offset  $d$  is the pipeline stage of the processor, and  $m$  is the number of data blocks in the buffer. The buffer index decrement is also modular  $m$  according to the equation

$$\text{if } (i-1 < 0) \ i = m-1 \quad (\text{EQ 7.2})$$

How this supports the systematic and efficient addressing of the FIFO is illustrated in Figure 7.5.

In Figure 7.5a, a communication between two processors on adjacent pipeline stages is shown. The address offset  $d$  of the two processors differs by 1, and the reading and writing to the FIFO occurs on different data blocks. After each sample, the buffer index circular shifts to the left so that the destination processor can read the data which was written by the source processor in the last sample. Meanwhile, the source processor is writing its new result to the other data block, overwriting the old data. Thus for

(a) Source Pipeline:  $d$  : Destination Pipeline:  $d+1$ (b) Source Pipeline:  $d$  : Destination Pipeline:  $d+2$ **FIGURE 7.5: Circular Buffer Implementation of FIFO.**

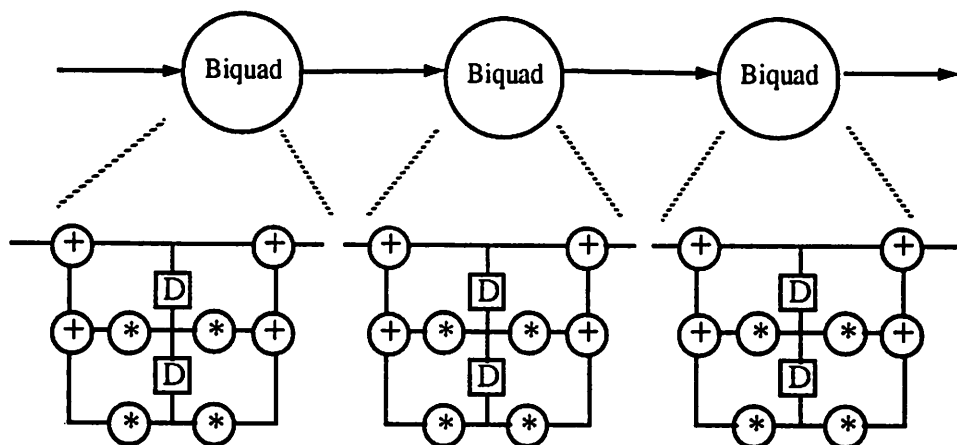
adjacent stage communication, the read and write swap addresses. When separated by more than one pipeline stages, the size of the circular buffer increases proportionally, and the same addressing strategy can apply. Figure 7.5b shows the communication of two processors separated by two pipeline stages. Here, the address offset is 2, and the reading of data is two blocks behind the writing of data. Note that for two processors communicating in the same pipeline stage, the buffer consists of only one data block, and synchronization is necessary, as was discussed in the previous section.

### 7.3.1.2 State Variables

Digital signal processing frequently uses delayed values of signals, that is, the values of signals at previous samples. General purpose computing, however, has no support for this concept. As a result, it is necessary to map it to more primitive computations which are commonly supported. To do this, the CDFG subgraph is traversed, and the maximum delay of all delayed signals are determined. This is easily

derived when analyzing delay nodes and delay lines. An array is needed to store the signal and its delayed values up to the maximum delay. Accessing a delayed signal is now mapped to accessing the array with the index being the delay value. At the end of each simulation cycle, before the next sample is read in, the array is updated to simulate the fact that it is now one sample later. One way to update the array is to shift every element of the array to the right by one. This requires copying every element in the delay line, which can be expensive for large delay examples. A much more efficient way is to use the same circular buffer mechanism as described above to model the delay line. The buffer index once again indicates the current sample. The array update now amounts to decrementing the index according to Equation 7.2. The buffer offset  $d$  is the amount of delay in the signal we want to access. Equation 7.1 now implements a delay signal access. Each access takes longer but the penalty is small compare to the copying of the array. Using circular buffers to implement delay lines is commonly used in DSP processors [Mot90], which provide hardware support for modulo addressing.

The last issue is the maintenance of state variables across function calls. Consider the scenario illustrated in Figure 7.6. The goal is to generate code for a 6th



**FIGURE 7.6: 6th order IIR filter**

order IIR filter, which comprises of three biquads. Each biquad has its own set of coefficients, and its own internal states, resulting from the delayed signals. On each use



of the biquad, the input sample and the coefficients are passed in. Since the code is exactly the same, it is desirable to have it implemented as a subroutine. The problem, however, is that each biquad has its own internal states, which should be updated only once every sample, not every time the biquad code is invoked. In this example, since the biquad is invoked three times per sample, three sets of internal states have to be kept. One approach is to keep three copies of the function, with each copy keeping its own internal states. This increases the code size tremendously. The problem gets worse when these functions are embedded in other functions which are again used multiple times. Clearly, this problem must be solved using a different approach.

The approach adopted is taken from the S2C Compiler [Sch88]. It adheres to the data flow paradigm, which says each node operation depends only on its inputs. In other words, the internal states are passed in along with other inputs to the function. Since the function results are only dependent upon the inputs, only one copy of the code is needed. Careful bookkeeping is required to systematically pass in the correct state.

### 7.3.2 C Code Emission

This section discusses the C code emission process. It assumes that the front-end parsing and the memory layout has already been done. The pseudo-code of the emission algorithm is given below. It assumes that a separate file is generated for each processor.

```

C_CodeEmission(Graph) {
  for each SubGraph i do
    OpenFile("proci.c");
    Allocate C structures to implement circular buffers for buffer edges;
    Allocate C structures to implement circular buffers for delayed signals;
    GenDeclarations();
    GenInitializations();
    GenReadInput();
    GenSimulationLoop();
    GenFunction(SubGraph);

```

```

        CloseFile("proci.c");
    }

```

After the CDFG has been partitioned into subgraphs for each processor, the code emission algorithm is applied to each subgraph. A new file is opened for each to store the code. All buffer edges representing interprocessor communications are processed to generate the necessary C data structures to implement the circular buffers. This includes the arrays for the buffers, and the buffer indices. Next, all delayed signals are processed and the same circular buffer data structures are generated. GenDeclarations() declares all the C structures above, as well as global constants. GenInitializations() generates code to initialize the circular buffers and global constants. GenReadInput() generates code to read input samples from a file, and write output samples to a file. The file names were derived from the command file. The code automatically exits if the end of file is reached. GenSimulationLoop() generates the infinite time loop which calls the subprogram, and updates all delay structures. Finally, GenFunction() generates code for the subprogram itself.

GenFunction() is a recursive procedure to generate the top level program and all of its subroutines, which are present if there exists any *func* node in the subgraph. The pseudo-code for GenFunction() is:

```

GenFunction(Graph) {
    for each function i do
        GenFunc(function);
}

GenFunc(Graph) {
    GenLocalDeclarations();
    if (Graph != Main)
        OrderExecution(Graph);
    GenStatements(Graph);
}

GenStatements(Graph) {

```

```

for each node in Graph do
  if (node is hierarchical)
    GenStatements(node->Subgraph);
  else
    GenCode(Node);
}

```

If the graph is the top level subgraph, the nodes are already ordered according to the schedule. However, for lower level subroutines, the nodes are ordered from input to output to guarantee that when code for a node is emitted, the code to generate the inputs of the node have already been emitted. GenCode() emits code to perform the operation of the node. The input and output variables are derived from the input and output edges, respectively. In addition, a check is performed at the beginning and end of each node generation to see if any local synchronization instructions are needed. If they are, the input and output edges of the node should have been annotated earlier with the appropriate information.

### 7.3.3 Floating-Point & Fixed-Point Simulation

The C code generator is designed to produce code which can be simulated using floating-point data types for quasi-infinite precision simulation, or fixed-point data types for bit-true simulation. Bit-true simulation is needed to assess the rounding and truncation effects of the fixed-point arithmetic on the behavior of the algorithm. By comparing the bit-true simulation to the floating-point simulation, the designer can decide on the best word width and fraction width for the application, and pick the target processor for real-time implementation accordingly. If the hardware is fixed, bit-true simulation will at least let the designer see the resultant outputs.

Both simulation models can run from the same C code. The technique has been implemented in the S2C compiler[Sch88]. All fixed-point primitive operations are implemented as C macros. Two header files are created: *highlevel.h* and *bittrue.h*. Depending on which type of simulation is desired, the corresponding header file is

compiled along with the C code. The `highlevel.h` header file is used for floating point simulation, while the `bittrue.h` file is used for fixed point simulation. Each file defines the same macro differently, depending on whether the computation is floating or fixed-point. As an example, when a fixed-point add node is encountered, the following macro is generated:

```
FixAdd(In1, TypeIn1, In2, TypeIn2, Out, TypeOut);
```

If `highlevel.h` is included, this macro would be defined simply as:

```
Out = In1 + In2;
```

with the type of all the edges declared as floats. If `bittrue.h` is included, this macro would be defined as:

```
AddBits(In1.bits, In2.bits, Out.bits);
if (TypeIn1 != TypeOut) CastBits(Out.bits, TypeIn1, TypeOut);
Coerce(Out.bits, TypeOut);
```

with the type of each edge declared as a C structure of bit streams. The fixed-point add operation entails adding the bits of the inputs and casting the result to the type of the output.

The header files give definitions for all arithmetic, logical, and relational operations. The separation of the behavior of the node and its implementation through the use of macros and header files allows the code emitter to be retargetable to different implementations of the operations. All modifications to the implementation of a fixed-point operation require only the modification of the header files. The code emitter module is not changed. The power of this technique extends beyond the usage as described in this section. It is possible to tailor the header file to issue C code which is known to compile efficiently on a particular machine. Extending this idea one step further, it is possible to emit DSP assembly code directly for real-time implementation.

While this is certainly realizable, it may still not produce code efficient enough for real-time execution. This topic is discussed in more detail in the next section.

### 7.3.4 DSP Code Emission

The development of DSP processors has made a significant impact on the real-time implementation of sophisticated DSP algorithms. DSP processors are specialized programmable microcomputers which often use extensive pipelining, multiple independent memories, parallel functional units, hardware looping, modulo addressing, and other innovative techniques to allow real-time processing. Their impressive processing power along with their low cost make them ideal for a wide range of real-time DSP applications. Recently, their usefulness has increased even further with the introduction of support for multiprocessor implementations. The newly introduced TMS320C40 from Texas Instruments Inc., for instance, has six ports for direct interprocessor communication and a six-channel coprocessor [Wat92].

However, it is generally difficult to generate optimized code for these DSP chips. This is primarily due to the constraints imposed by the pipelined architecture and the parallel data transfers. Other features such as zero-overhead looping, single-cycle multiply-accumulate, and modulo addressing are also difficult to exploit. To obtain a good implementation, DSP designers have traditionally been forced to manually code their applications in assembly.

To ease this task, many techniques have been developed to generate optimized realizations of DSP applications from high level languages or flowgraph descriptions. Almost all DSP processor vendors such as TI, Motorola, AT&T supply C compilers for their processors. Our experience with one of these compilers, the AT&T C compiler for the DSP32, has been disappointing. This usually stems from the fact that some semantic information is lost in converting the DSP description into C, and the C compiler may not be able to perform the desired optimizations[Bau90]. While these compilers are

continually improving, the general consensus is that the best result will come from direct code generation from flowgraph specification.

A simple and effective approach is to provide a library of DSP functions, each hand coded and optimized. The larger the granularity of the function, the more effective the optimization. This technique has been used by Comdisco Systems Inc. in their Code Generation System [Pow92] and in the Gabriel system from UC Berkeley [Lee89c]. The major problem with this approach is the dependency on a library set, a lack of register and memory allocation choices and possible redundant data transfers between these functions. Recently, these two groups have joined to improve their code generation strategy. In order to increase code generator allocation choices, they define their library blocks in a meta-assembly language that uses the syntax of the assembly code of the target processor, but symbolically references registers and memory. The optimizing code generator compiles these segments together, allocates registers and memory, and inserts data movement instructions as needed to produce good code [Pow92]. While this will improve the code, the designer is still restricted to the library of components for optimized code.

A more general approach by Genin [Gen89] uses a ruled-base pattern recognizer to identify groups of low level nodes and merge them into nodes with a higher semantic content. For examples, additions, products, and delays are merged into *product accumulation with delay* nodes. These nodes can again be merged into *filter* nodes. By accumulating such information, general strategies for optimal code generation and memory management can be derived, which make the best possible use of the context. This allows two identical nodes compiled in two different contexts to produce different code. The reported performance of the code generator is about 5 to 50 times faster than the one produced with C compilers and is comparable to the code generated by DSP experts.

Another code generator by Kim [Kim90] relies on the fact that DSP processors are usually pipelined and often rely on pointer based addressing modes. Hence, the compilation concentrates on the allocation of pointer registers rather than on arithmetic operations. The register allocation algorithm keeps track of the addressing span of registers. Each time pointer register addressing is required, the span ranges are referenced to see if there is a register which contains the address in its span range. This register is chosen for loading of data. To improve code even more, the register loads are placed in NOP instructions which are inserted to meet pipeline synchronizations. Finally, special attention is given to register allocation of inner loops to minimize register loads between iterations.

The last two techniques are complementary as they attack important but different parts of the code generation problem. An effective code generator must combine the techniques proposed here to simultaneously address the register allocation problem due to pipelining, parallel data accesses, and pointer addressing, and at the same time recognize common DSP constructs to efficiently use multiply-accumulate instructions, hardware looping, and modular addressing. Furthermore, considerations must also be given to the system constraints imposed by interprocessor communication and synchronization requirements, as well as shared memory access conflicts.

Finally, as new code generation optimizations are developed for these specialized DSP processors, it is not clear whether these effects can be estimated quickly and accurately at the flowgraph level. This feature, as recalled, is of prime importance as estimations of the computation times, memory requirements, and interprocessor communication delays are required for scheduling. An optimized code generator which cannot be characterized well from the flowgraph level will not be as beneficial as one which is. In summary, while multiprocessing scheduling has received much of the attention, a great deal of work is still needed in the code generation phase.

## 7.4 SUMMARY

A multiprocessor code generation strategy has been presented. It is performed in two steps. The first step allocate and layout the buffer memory used for interprocessor communication. Several algorithms were presented to layout the memory for centralized and distributed memory systems. A FIFO data structure is used to store the communications. Under the McDAS pipelined execution model, the FIFO guarantees that no data will be overwritten before it has been read. An addressing scheme is presented to allow each processor to systematically access the correct location of the FIFOs at each sample.

The second step is the code emitter, which generates code for each processor. Details on how the C code can be used for fixed-point and floating-point simulation is given. For real-time execution, the DSP code generated by compiling the C code is found to be too inefficient. Direct DSP code generation from the CDFG is proposed. A discussion of the issues and the current research efforts in this area is presented.



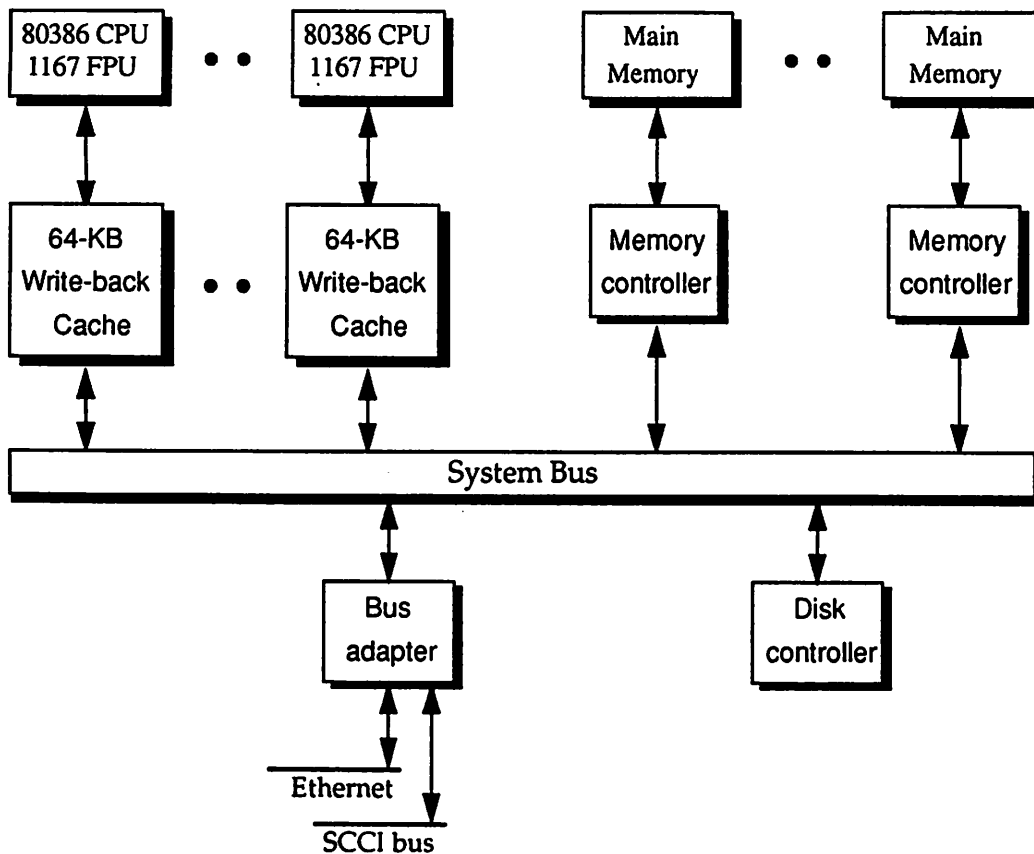
# SCHEDULING RESULTS

In this chapter, the results obtained from the scheduling and execution of programs using McDAS are presented. In Section 8.1, two multiprocessor systems are described. The first is a commercial multiprocessor system from Sequent Computer Systems, while the second is a custom-built DSP multiprocessor called SMART. In Section 8.2, we present and analyze a number of examples which are scheduled and compiled on these systems. Both the scheduling performance as well as the accuracy of the estimation techniques are analyzed.

## 8.1 TARGET ARCHITECTURES

### 8.1.1 The Sequent Symmetry Multiprocessor

The Symmetry multiprocessor [Lov88] by Sequent Computer Systems is a tightly coupled, single shared-bus MIMD multiprocessor composed of 4 to 30 processors. Our particular machine has 14 processors. The bus is a 64-bit bus with a bandwidth of 53 MB/sec. Each processor is an Intel 386, coupled with a Weitek 1167 floating point accelerator. Figure 8.1 shows a block diagram of the Symmetry architecture. The processors communicate through a centralized shared memory, although caching is supported to reduce bus traffic. Each cache has a size of 64KB to provide a high hit rate and therefore a high effective memory bandwidth as seen by the



**FIGURE 8.1: The Sequent Symmetry Multiprocessor**

processor. A write-back cache policy is used to reduce the number of write operations on the shared bus. To support exclusive access to shared data structures, the Symmetry provides semaphores to allow the user to lock any section of physical memory. Hardware support for barrier synchronizations is also provided.

The Symmetry system runs the DYNIX operating system, a version of UNIX, and supports programming in C, Fortran, and Pascal. Shared data can be declared explicitly in the program, and is accessible by all processors working in parallel. The spawning of parallel processes to be executed on different processors is done using a system call called *fork*. In the Symmetry, the code for all processors is placed in one file. Each processor has associated with it a processor number, which is used to assign a sub-program to its corresponding processor. Special calls are available to reserve and

free processors for execution. The code structure for a statically scheduled program on the Sequent computer using N processors is shown below:

```

main () {
    m_set_procs(N);
    m_fork(Program);
    m_kill_procs();
}

Program() {
    proc = m_get_myid();
    switch(proc)
        case 0: execute subgraph0;
        case 1: execute subgraph1;
        case 2: execute subgraph 2;
        .
        .
        case N-1: execute subgraph N-1;
}

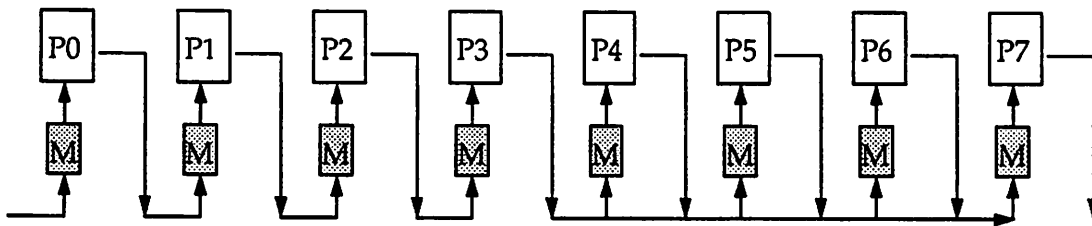
```

Examples of some C programs generated by the code generator for the Sequent Symmetry machine are shown in Appendix C.

### 8. 1. 2 The SMART Multiprocessor

The SMART (Switchable Multiprocessor Architecture supporting Real Time applications) multiprocessor [Koh89] is a dedicated compute-engine developed to allow real-time behavioral simulation of DSP algorithms. The machine attempts to speedup simulation by at least two orders of magnitude as compared to general purpose computer architectures. The speedup is achieved through two means. First, in order to handle the number crunching bottleneck, a high performance DSP processor with both floating-point and fixed-point computations is used as the core processing unit. This results in an order of magnitude in speedup. Secondly, an additional order of magnitude in speedup is obtained by exploiting the high degree of concurrency present in most DSP applications through multiprocessing.

The SMART system consists of an array of 8 AT&T DSP32C digital signal processors connected in a linear fashion by a single shared-bus. The peak performance of the system is 160MFLOPS. A key feature is that the bus is *configurable*, in that there are switches between neighboring processors which can be opened or closed to divide the processors into groups. Processors with local communications are put into one group so they can communicate among themselves independent of other groups, thus boosting the overall communication bandwidth of the bus. Bypass units across the switches allow global communication among all processors. A block diagram of the SMART configurable bus is shown in Figure 8.2.



**FIGURE 8.2: The SMART Multiprocessor**

To reduce the amount of accesses to the shared memory, the large global shared memory is distributed to all processors. The basic scheme of communicating data between two processors is that of a *global-write and local-read* scheme, where the source processor can write directly into the destination processor's section of shared memory. The destination processor can then read the data via its local bus. This roughly reduces the number of accesses to the shared bus by one half.

In case of multiple requests to use the shared bus, some requests will be stalled, resulting in communication overhead. The SMART system provides a write-queue for all interprocessor write operations so that the source processor can immediately resume its computation as soon as data has been written to the queue. By overlapping the computation time with the overhead time due to bus arbitration, the effective communication overhead is reduced.

Finally, SMART provides hardware synchronization primitives for barriers and locks.

Programming of the SMART system is done by writing C code and compiling it with the AT&T C compiler. Unfortunately, while benchmarking was done on a DSP32C emulator to provide computation to characterize SMART, the machine was not operational at the time the code generator was completed. Hence, it is not possible in this thesis to compare the scheduling results with the actual running times on the SMART machine as it was possible for the Sequent.

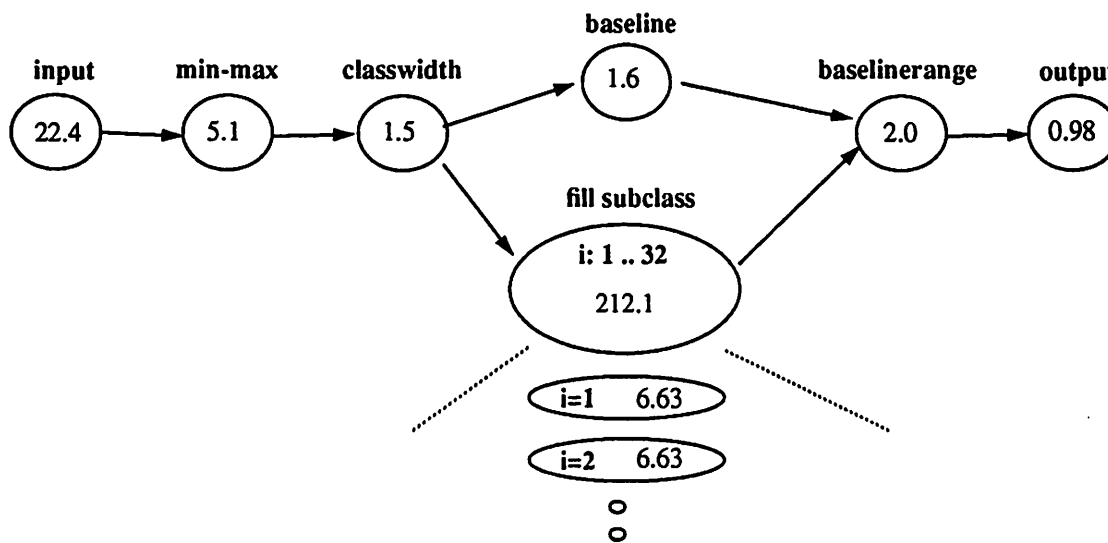
## 8.2 RESULTS

In this section, we analyze a number of examples which were scheduled and compiled by McDAS. The analysis is organized into specific topics, each addressing a different issue. The first two subsections examine the scheduling of a specific example on both the Sequent and SMART architectures, respectively. They give insights into how tasks are partitioned and scheduled, and provide a comparison between the estimated and actual computation times. Implementations on different numbers of processors and topologies are also analyzed to evaluate the effects of these architectural changes on the performance. Finally, the third subsection analyzes the scheduling results of a wide range of applications on a fixed architecture, specifically the Sequent machine. The applications are chosen to have different concurrency types, granularity, and communication patterns.

### 8.2.1 Scheduling a Histogram Computation on the Sequent Multiprocessor

A histogram computation involves grouping a sequence of samples into subclasses based on their values. In this example, the input is an array of 128 samples, to be partitioned into 32 subclasses. The computation proceeds as follows: First, the

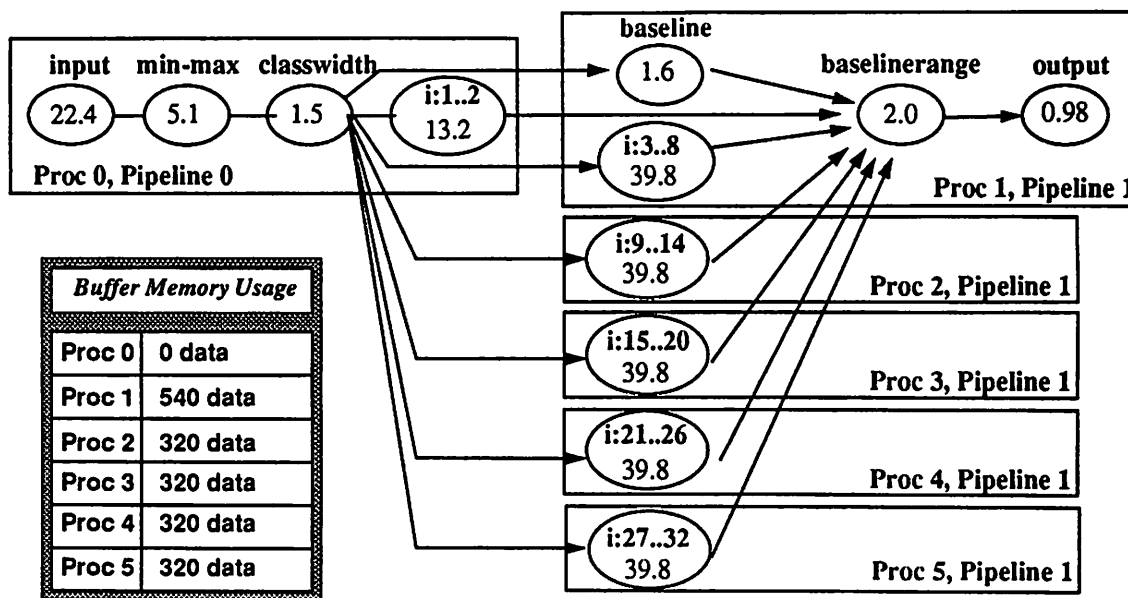
minimum and maximum values of the array are determined. Next, the size of each subclass is calculated from the total range. The third task fills in all the subclasses. This is a parallel task where each subclass scans the array and counts how many samples fall within its scope. The fourth task determines the baseline class, defined as the subclass whose range covers 0. Finally, the last task computes the baseline range, given as the number of samples in the 5 subclasses surrounding the baseline class. Input and Output tasks perform the file reading and writing. The CDFG is shown in Figure 8.3. The value



**FIGURE 8.3 : Histogram Example**

inside each node represents its estimated computation time in milliseconds. As can be seen, the parallel iteration is the most computationally-intensive node. The total computation time equals 245.6 ms, with 68 nodes in the hierarchical graph, and 30,687 nodes in the flattened graph.

In a first analysis, the example is scheduled onto 6 processors of the Sequent. Figure 8.4 shows the resultant CDFG. With 6 processors, the minimum stagetime is  $30,687/6$  or 41ms. The scheduler automatically decomposes the parallel iteration as the stagetime is decreased. The scheduling takes 20 sec of CPU time, and includes the logging of intermediate solutions after each iteration. The resultant schedule has 2 pipeline stages, with processor 0 in stage 0 and processors 1-5 in stage 1. The buffer



**FIGURE 8.4 : Histogram Example Scheduled on 6 Processors.**

memory needed for interprocessor communication is shown in the accompanied table. Processors 2-5 all receive the 128 sample input array as well as an array giving the lower and upper bounds of the 32 subclasses, for a total of 160 data words. Since the pipeline length is 2, the total size of the buffer is 320. Processor 1 receives this and the results of all the iterations for a total of 540 data words. Analyzing the completion times of the processors as estimated by the scheduler and as actually measured on the Sequent (Figure 8.5), we see that the load is evenly distributed across the processors, and the estimated time agrees very well with the actual completion time.

To illustrate the correlation between the estimated and the actual completion time further, Figure 8.6 plots the two completion times as a function of the number of available processors. The quality of the estimation remains quite good across a wide range of load partitions. Figure 8.7 shows the speedup obtained in the Histogram example, as a function of the processors. For this example, we observe that McDAS is able to consistently achieve a faster throughput with each additional processor. For

### Load Balancing

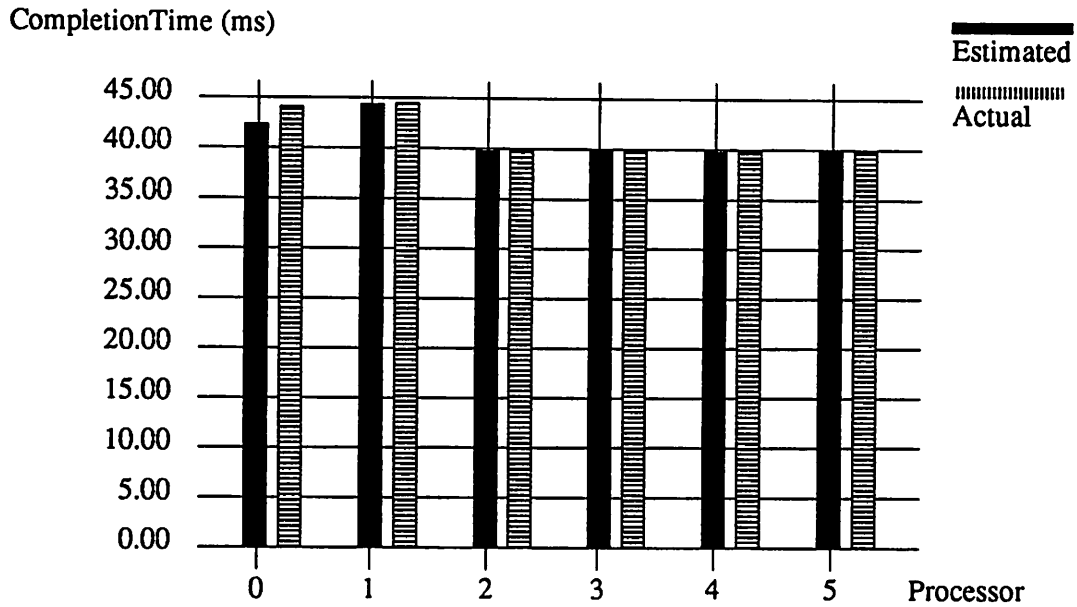


FIGURE 8.5 : Load Balancing on 6 Processors for Histogram Example

### Completion Time

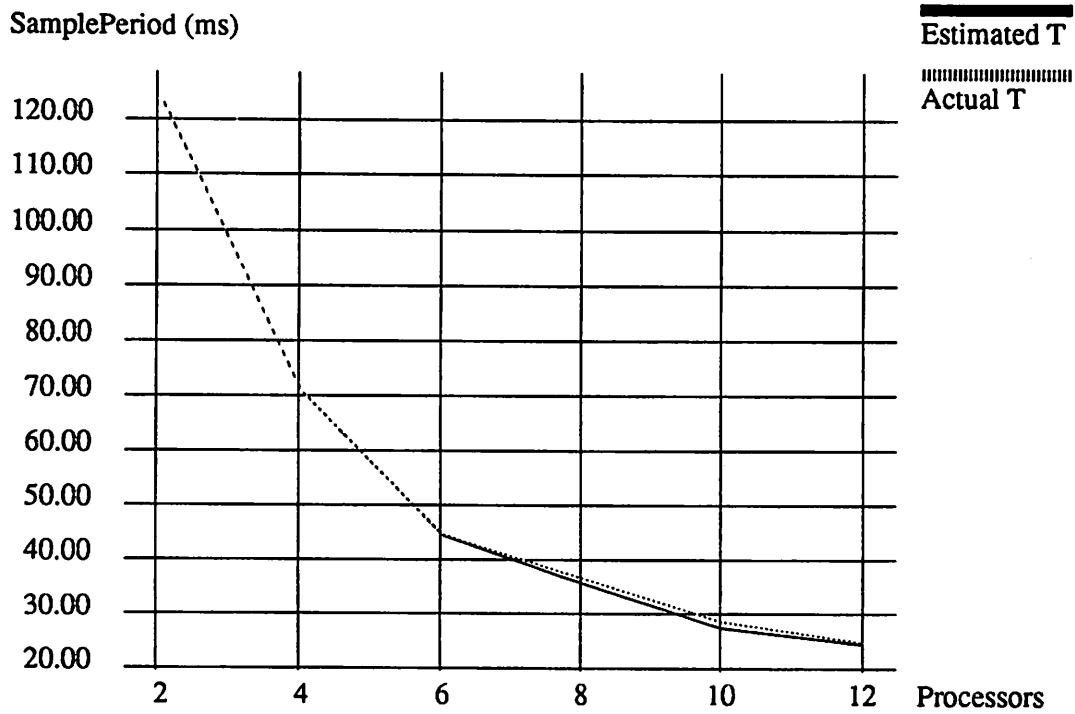
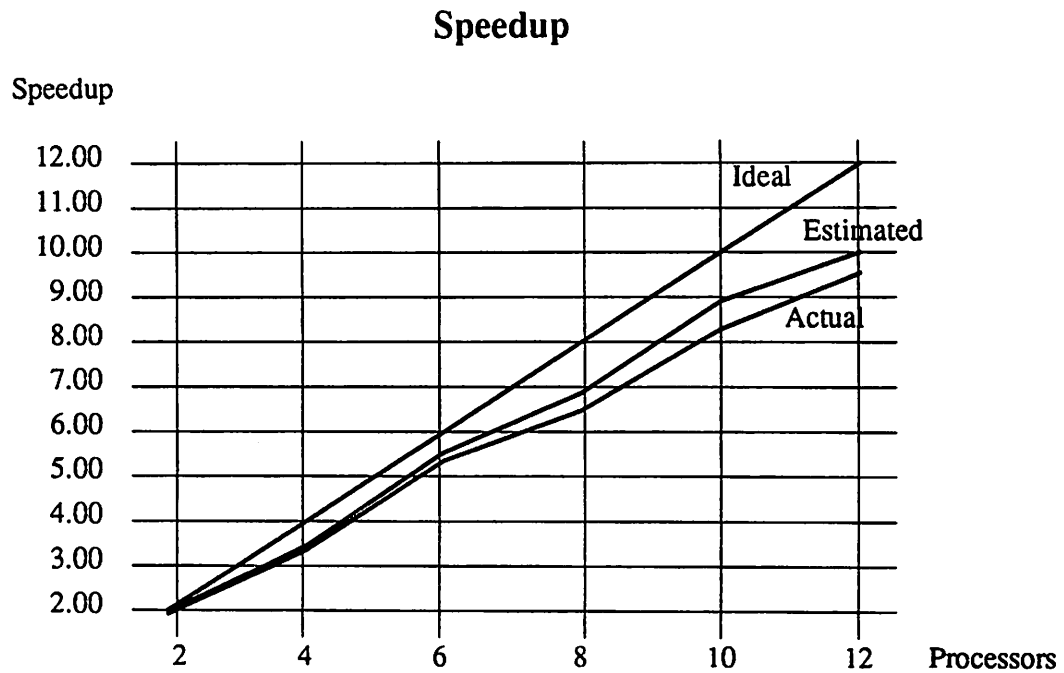


FIGURE 8.6 : Estimated vs. Actual Running Times for Histogram Example

comparison, the ideal speedup is shown. This can only result if there is perfect load



balancing and there is no cost for interprocessor communication.



**FIGURE 8.7 : Speedup Plot for Histogram Example**

Table 8.1 shows the sizes of the maximum buffer memory required of a processor as a function of the available processors. An important point to note here is

# Procs	# Pipelines	Size of Maximum Buffer	Size of Total Buffer
2	2	540	540
4	2	540	1180
6	2	540	1820
8	2	416	2336
10	3	678	3660
12	3	735	5279

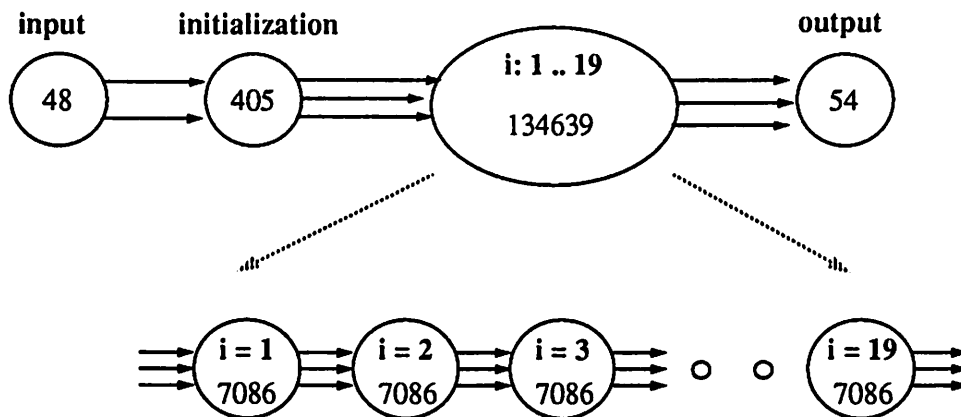
**TABLE 8.1**

that the buffer memory size increases with the number of pipelines in the implementation. This is expected since the length of the FIFO buffers varies directly

with the pipeline length. The difference will be even more pronounced in the next example, where pipelining is extensively used.

## 8.2.2 Scheduling a Cordic Computation on the SMART Multiprocessor

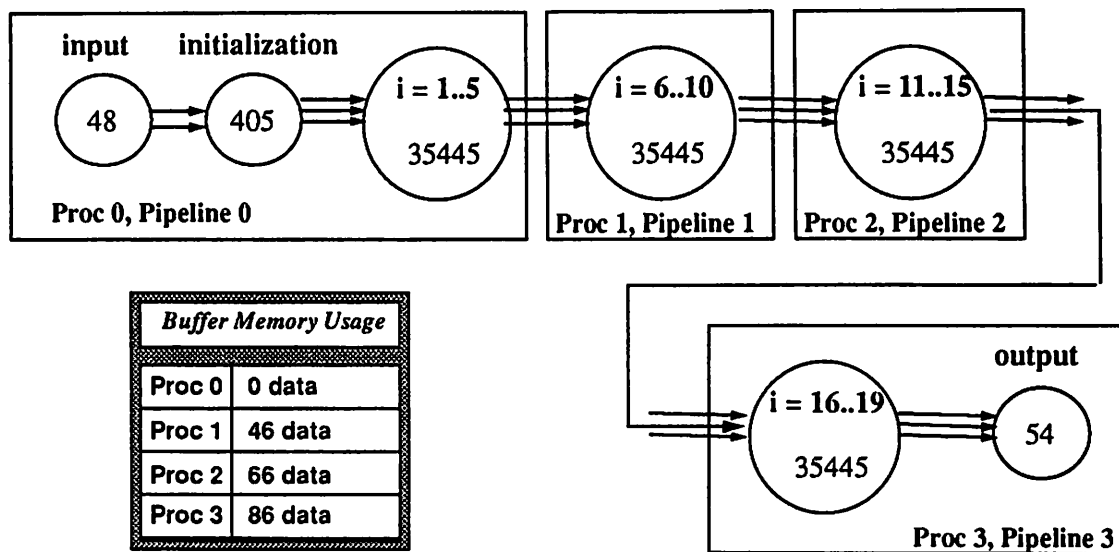
The Cordic algorithm converts cartesian to polar coordinates iteratively in 20 steps. It takes as input an (X,Y) coordinate, as well as an array of correction angles. The loop iteratively calculates the corresponding amplitude and phase. Figure 8.8 shows the algorithm. The nodes are annotated with their computation times, which are measured



**FIGURE 8.8:** Cordic Example

in clock cycles. For a DSP32C processor running at 50Mhz, each clock cycle takes 20ns. Since each iteration is dependent on the results of the previous iteration, the computation is sequential in nature. A scheduling algorithm which only exploits spatial concurrency would perform poorly on this example. McDAS, on the other hand, is able to achieve a good speedup by pipelining the loop and assigning successive loop iterations to successive processors.

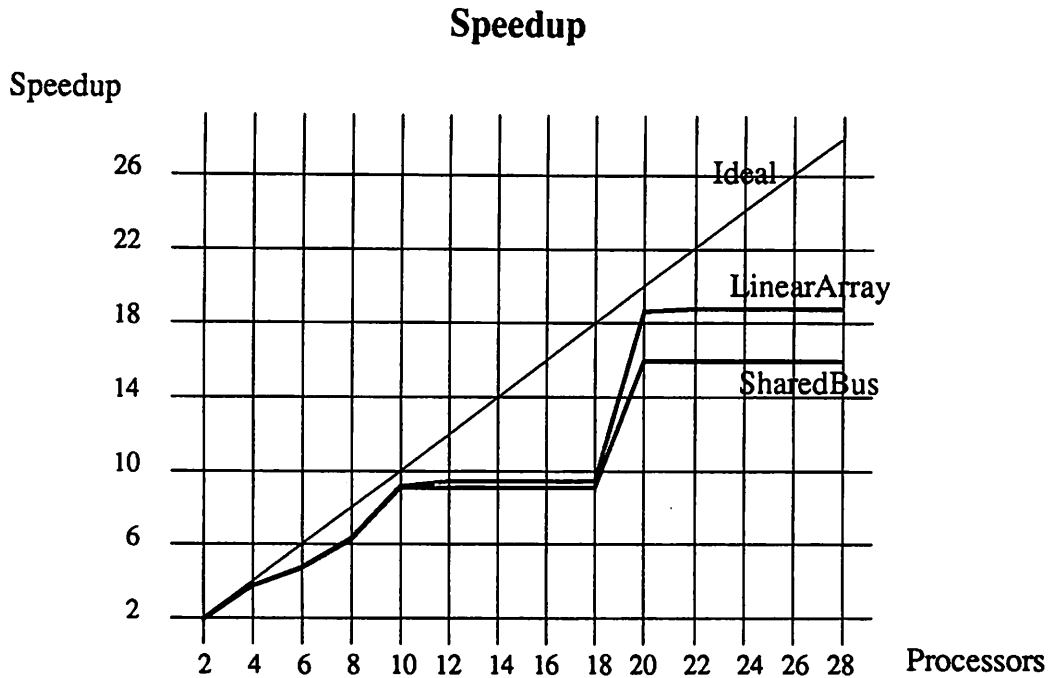
Figure 8.8 shows the Cordic algorithm scheduled onto 4 DSP32C processors. The speedup is 3.76, for an average processor utilization of 94.16%. The accompanied



**FIGURE 8.9: Cordic Example Scheduled on 4 Processors**

table gives the buffer memory usage for each processor. It is assumed that the array of correction angles is sent along with the intermediate results to the next pipeline stage. The processors are thus required to buffer the array for correct execution. The farther a processor is situated in the pipeline, the larger the buffer.

In contrast to the Histogram example, the Cordic program is communication intensive. With a pipeline configuration as above, each processor in the pipeline will send its results to the successor processor at the end of its computation. A linear array architecture allows these neighbor-to-neighbor communications to occur simultaneously, while a single shared-bus architecture is soon saturated. Figure 8.10 plots the speedup for the Cordic algorithm on the SMART machine with the bus configured as a linear array and a shared-bus, respectively. Again, the ideal speedup is included for comparison. The first thing we notice is the “stair case” effect of the speedup curves. Currently, McDAS does not decompose loops at the middle of an iteration. No attempt will be made, for instance, to assign 1.5 iterations of a loop on one processor and the remainder on another. As a result, when there are 12 to 18 processors available, the scheduler still has to assign at least one processor 2 iterations to execute.

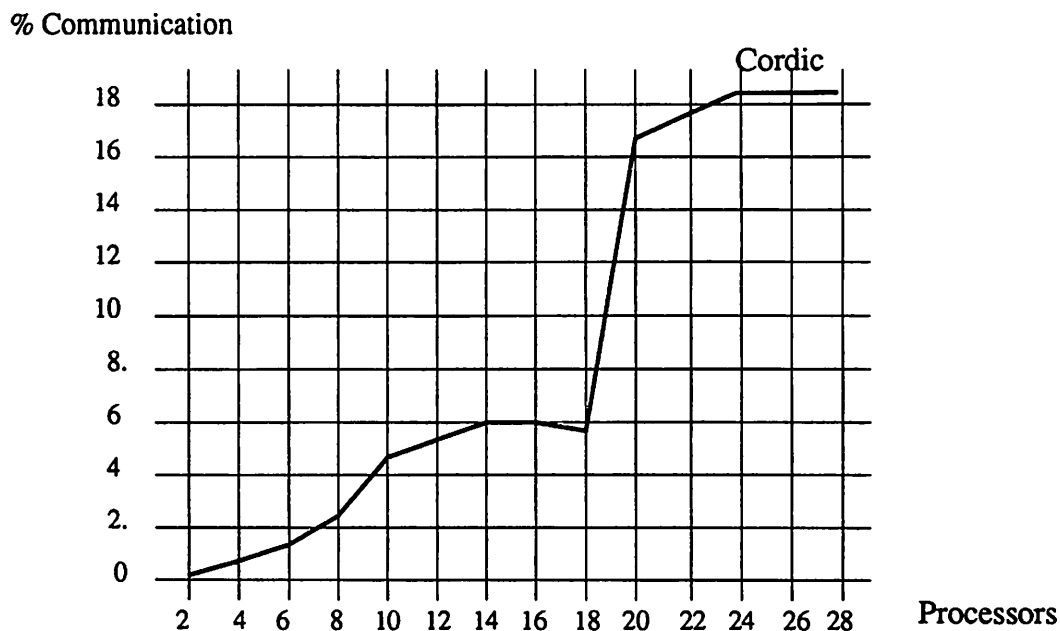


**FIGURE 8.10 : Speedup for Cordic Example**

The throughput remains constant until enough processors are available to allow each iteration to be executed by its own processor. This occurs at 20 processors, after which a great leap in throughput is attained. The decision to enforce iteration boundaries is made to keep the number of nodes considered by the scheduler low and maintain reasonable user response time. It also keeps program code compact. Because of this, the computation time of one iteration of a loop may represent the finest granularity level in the application (except when the number of iterations is less than the number of available processors, for which each iteration will be expanded to its subgraph). For most examples that we have encountered (including the Histogram), this value is much smaller than the overall computation time and hence does not greatly affect the load balancing. The Cordic example, however, is not computationally-intensive, allowing this phenomenon to hinder the load balancing.

The amount of communication among the processors is shown in Figure 8.11. When the number of processors is between 2 and 12, as more processors are utilized,

## Communication



**FIGURE 8.11 : Percent Communication on Cordic Example.**

the amount of interprocessor communication increases. From 12 to 18 processors, the scheduler is not utilizing the additional processors well; they are either idle or only slightly used. As a result, there is no substantial increase in the amount of communication. When the processor count reaches 20, each processor is assigned an iteration, and is heavily utilized. All 20 processors are now communicating with their neighbor as opposed to 10. This results in a great surge of interprocessor communications as shown in Figure 8.11. Architectures which are not able to cope with this communication demand will suffer in performance. For neighbor-to-neighbor communications, a linear array architecture can process these data transfers simultaneously, resulting in a good speedup. A single shared bus architecture however, will perform much worse as the shared bus forces all of these data transfers to be processed sequentially. By contrast, the communication demand of the Histogram example averages only 2%, making the processor interconnection not as important an issue.

Finally, Table 8.1 shows the buffer memory usage of the Cordic example on SMART as a function of the available processors. Since the application is extensively

# Procs	# Pipelines	Size of Maximum Buffer	Size of Total Buffer
2	2	86	86
6	6	106	308
10	10	206	1134
14	11	206	1165
18	11	206	1168
22	21	406	4302

TABLE 8.2

pipelined, the buffer memory requirements of the processors in the last stages of the pipeline can be quite large.

### 8. 2. 3 Scheduling Different Applications

The scheduling algorithm has been tested on a variety of DSP examples with different types of concurrency and communication patterns. The goal is to see how the algorithm exploits the concurrency to achieve speedup. For all examples discussed in this section, a Sequent shared-bus multiprocessor composed of 8 processors is used as the target system.

Table 8.3 shows the scheduling results for a number of examples which do not contain global recursions. Those that are will be discussed later. A few entries need explanation.  $F(G)$  and  $H(G)$  give the number of nodes in the totally flattened graph and in the hierarchical graph, respectively.  $R(G)$  gives the number of nodes that the algorithm considered during its search.  $F(G)$  characterizes the computation requirement of the examples, while  $H(G)$  illustrates the compactness of the hierarchical description.

$R(G)$  shows how the top down hierarchical search allows the scheduler to minimize the number of nodes considered during scheduling.

Example	F(G)	H(G)	R(G)	Concurrency	# Pipelines	Speedup
DTW	1.7e8	98	15	Pipe/Par	2	7.08
Matrix Mult.	2.0e6	24	11	Pipe/Par	2	6.64
256-pt DFT	7.6e5	35	9	Pipe/Par	3	6.94
Pitch Extractor	1.2e5	270	22	Pipe/Par	3	7.53
2-Norm	1926	23	12	Pipe/Par	4	7.61
Cordic	494	45	24	Pipeline	7	6.37
8-pt DCT	87	62	75	Pipe/Par	5	3.26

TABLE 8.3

The dynamic time warp (DTW) algorithm [Sak78] is used in speech recognition to compute a match between an unknown signal and a library of templates. The algorithm above performs 1000 template matchings and outputs the score. The dominant concurrency lies in the parallel template matching task. The matrix multiplication application multiplies a 64x64 matrix by a 64x64 matrix, and outputs the resultant matrix. Each element of the resultant matrix can be calculated independently, yielding parallelism. The pitch extractor [Slu80] starts with a Fourier transformed speech signal, performs an amplitude calculation, searches for local maxima in the spectrum to derive a candidate pitch, and matches the pitch against a set of templates. The amplitude calculation and local maxima search are sequential operations, while the template matching contains parallelism. For more fine-grain examples, the 2-norm calculation squares each element of a vector, and then sum them. The first task can be executed in parallel, while the second must be done sequentially. The cordic calculation is the same example as described in the last section, and involves an iterative computation to convert cartesian coordinates to polar coordinates. Finally, the DCT example performs an 8-point discrete cosine transform in 5 stages. The stages are

expanded in line, and thus there are no loops in the description. This explains the large number of nodes present in  $R(G)$  for the example.

In almost all cases, both temporal and spatial concurrency are exploited using a combination of pipelining and parallel execution. The amount of pipelining and parallelism is given by the # Pipelines column. Since there are 8 processors, those which use few pipeline stages must have many processors working in parallel. As we can see, there is a good mix in general. Finally, the last column shows the speedup obtained by the scheduler. It shows that although the examples are quite diverse in computational complexity and concurrency pattern, the speedup obtained is consistently good. One exception is the DCT example, which only achieves a speedup of 3.26. Although it has sufficient concurrency, the amount of communication as compared to computation (it has only 87 primitive operations) is quite significant. This is explored more thoroughly in Table 8.4, where additional statistics derived from the scheduling results are analyzed.

Example	# Iter	# Scheduling Invocations / Iter	CPU (sec)	% Comm. Overhead	Max Buffer Size
DTW	15	5	30.2	1.53%	807,928
Matrix Mult	14	1	5.8	5.19%	36,864
256-pt DFT	13	1	8.2	0.27%	1437
Pitch Extractor	15	3	21.7	1.12%	335
2-Norm	14	1	11.2	0.2%	2032
Cordic	14	8	12.6	4.88%	146
8-pt DCT	10	25	89.3	51.0%	18

**TABLE 8.4**

The percent communication overhead tells how much time the processors are idle waiting for the bus to be available for communication or synchronization. The DCT



has over 50% communication overhead, giving each processor an average utilization of 40.8%. If the communication cost on the Sequent is reduced to zero, scheduling reveals that the average processor utilization would jump to 80.2%. The analysis shows that the Sequent machine is not efficient for fine-grain and communication intensive computations because its interprocessor communication mechanism is somewhat expensive.

The # Iter column gives the number of iterations the scheduler took to find the minimal stagetime. All examples needed no more than 15 iterations to converge. As discussed in Chapter 6, the scheduling algorithm may perform a number of scheduling invocations at each iteration. One reason may be that the path merging mechanism was invoked to improve processor utilization. The other source can come from successive decomposition of large nodes in cycles to meet the cycle scheduling bound. The latter feature will come in when examples with cycles are analyzed. For the examples here, all scheduling invocations stems from path merging. The DCT example, with so little hierarchy in the description, was a prime candidate for path merging. There average number of path merging steps for the example equals 25. The number of iterations and the number of path merges together dictate the scheduling CPU time. The CPU measurements shown are based on a Sun Sparc II, and include the extensive logging of schedule data for analysis.

#### **8. 2. 4 Scheduling Applications with Global Recursions**

We analyzed here three examples with global feedback cycles: An echo Canceller, an adaptive differential pulse code modulator (ADPCM), and a decision feedback equalizer (DFE). An echo canceller [Hon84] is used to cancel out the echo of the talker or the receiver in the telephone network. This is often done by using a least mean square (lms) filter to estimate the echo from the input signal and using it to remove the echo from the output signal. The lms filter contains a single cycle feedback.

ADPCM [Hon84] is a technique to reduce the bit rate needed to transmit a signal by sending an error signal (the difference in the present and past signal values), rather than the signal value itself. Since the error is usually small, less bits are required to represent the error signal. At the receiver's end, the original signal is recovered using feedback. Finally, a DFE [Hon84] filter is often used to help equalize the frequency-dependent channel attenuation in data transmission.

The global delay recursion prohibits pipelining. However, each example contains iterations in the cycle which can be executed in parallel. As a result, retiming and parallel execution are used to increase performance. Since the speedup is limited by the cycle scheduling bound (chapter 6.4), a comparison to the ideal speedup is no longer meaningful. Table 8.5 shows the scheduling results for these examples.

Example	Concurrency	# Pipelines	Stagetime	Cycle Bound	Speedup
ADPCM	Retime/Par	2	1381.5	1362.8	3.72
Echo Canc	Retime/Par	2	12157	6649	2.34
DFE	Retime/Par	1	1500.8	1238.5	4.22

**TABLE 8.5**

For the ADPCM and DFE examples, the scheduling algorithm performs quite well as the pipeline stagetime is brought down close to the cycle scheduling bound. This is possible because there are enough free processors to fully exploit the parallel iterations in the cycle. In the echo canceller case however, the number of available processors is insufficient to fully exploit all the parallelism in the iteration node. As a result, each processor allocated to the iteration node still has to perform a number of iterations sequentially. Hence the stagetime is still considerably larger than the cycle bound. This analysis is confirmed when these examples are scheduled with a larger number of processors. The ADPCM and DFE examples fail to improve, while for the echo canceller example, the scheduler succeeds to further minimize the stagetime. In

principle, the scheduling of applications with cycles is thus constrained not only by the cycle bound, but remains constrained by the available resources also, as expected.

## 8.3 SUMMARY

In this chapter, the scheduling and code generation tools in McDAS are evaluated with a number of benchmark DSP examples. Two multiprocessor machines are used as target architectures: The Sequent and SMART machines. The Sequent computer serves not only as a sample architecture for the scheduler, but is also currently the only machine targeted by the code generator. Hence, applications which are scheduled can be executed to verify the code generator and the estimation routines. The SMART machine, on the other hand, with its configurable bus, allows for the exploration of a wide range of design implementations.

The results obtained show the scheduler's ability to exploit different types of concurrency to achieve good speedup. For examples with spatial concurrency, parallel execution is used. For examples with temporal concurrency, pipelining is used. In examples with cycles, retiming is employed. Often, a combination of these techniques are used to obtain the greatest speedup gain.

The ability to traverse hierarchy allows the scheduler to adapt to the number of available processors. The more processors, the finer the granularity being exploited. The top down hierarchical search strategy is shown to reduce the search space considerably, enabling fast scheduling performance. Finally, comparisons between the estimated and the actual execution times of tasks on the Sequent confirms the practical use of the McDAS system as a multiprocessor compilation environment.

# CONCLUSION

In this chapter, we review the major results presented in the previous chapters. While the presented methods and algorithms adequately address many key issues in DSP multiprocessor implementation, more work is needed to make the compilation environment complete. An outline of such work is proposed for future research.

## 9.1 SUMMARY

The goal from the start of this research has been to develop a multiprocessor scheduling algorithm for DSP implementation which can exploit all available concurrency styles, at any level of granularity necessary. This has to be done within a compilation environment which is able to address all practical constraints of a multiprocessor system such as communication delays, and memory and processor availability constraints.

The goal is achieved by attacking the problem on two fronts. On one front, estimation is used to collect and store as much information about the input application and the target architecture as possible. On the other front, a scheduling algorithm is developed to exploit all the information available to obtain the highest quality solution. The results of the first objective are the hierarchical control/data flowgraph (CDFG) representation and the methodologies for estimating computation and communication

delays. These were discussed in Chapters 3, 4, and 5 respectively. The CDFG format stores all levels of hierarchy in the application in an efficient manner, allowing the scheduler to quickly traverse the different granularity available in the application. The computation estimation model gives the scheduler precise computation times and memory requirements of nodes at any level of hierarchy. The estimation is based on benchmarking programs to obtain computation times of primitive operators, and accumulating these costs in a hierarchical fashion for more complex computations. A nice feature of this technique is the inclusion of the effects of the underlying compiler technology in its cost. The effectiveness of the proposed strategy is validated by comparing the estimated and actual computation times for a number of example programs. In all benchmarks for both the Sequent and SMART multiprocessor systems, an error margin of  $< 5\%$  is achieved. Finally, the communication model uses a time-slot bus reservation strategy to accurately estimate the communication delay of a data transfer between two processors, taking bus congestion into consideration.

The scheduling algorithm presented in Chapter 6 is the result of the effort to meet the second objective. It can simultaneously exploit both spatial and temporal concurrency to achieve speedup. This can be done at any level of hierarchy in the flowgraph, allowing the scheduler to consider the concurrency at a level of granularity consistent with the amount of available processors. Furthermore, the scheduler is able to accept architectural constraints such as maximum bounds on the number of processors and amount of memory available, as well as the architecture topology. The results on a set of benchmarks demonstrate the scheduler's ability to achieve near optimal speedups across a wide range of applications.

To increase the applicability of the scheduler, a software environment is developed to provide a complete compilation path for DSP applications, from behavioral specification to code execution. The environment allows a designer to experiment with different architectures, and quickly implement designs. A number of

graphic display tools are available to provide analysis of the flowgraph structure and scheduling results.

The CDFG database serves as a central repository, on which compilation tasks, such as computation estimation, scheduling, and code generation are executed. This modularity in the design environment allows additional tools to be easily integrated into the compilation process. As an example, a set of flowgraph optimizing transformations to perform common subexpression elimination, dead code elimination, and manifest expression reduction has been incorporated. These transformations are automatically invoked after the CDFG flowgraph generation. Another flowgraph transformation which has been implemented is the multirate transformation. It converts a CDFG with multiple sampling rates into one with a single sampling rate. This is done by clustering operations with the sample sampling rate into a process, and repeatedly invoking each process according to its rate. This transformation is automatically applied to multirate applications. Many additional tools, especially flowgraph transformations, can make a significant contribution to the compilation environment. These will be discussed next as part of the discussion on directions for future work.

## 9.2 FUTURE RESEARCH

As often occurs in research, an attempt to answer one set of challenging questions produces as a side effect a number of even more challenging questions. The development of the McDAS environment offers no exception. Even though the current system already provides the designer with the capability to carry an application from specification to design exploration and implementation, the concepts presented hereafter can help to enhance those capabilities and turn McDAS into an even more powerful multiprocessor compilation environment.

A number of topics directly in line with the work in McDAS have already been alluded to earlier. These as well as some other directions will be summarized here.

### **9.2.1 Data Dependency Analysis**

In the generation of a flowgraph from a textual description, careful data dependency analysis is required to correctly connect data edges to nodes which are dependent on them. The main challenge is the dependency analysis of multidimensional signals such as arrays or matrices. This requires a careful examination of the range covered by the array indices. As an example, a read operation of an array, at some specified index, must trail a write operation to the same index of the array. The problem lies in a lack of a concise way to represent a range of coverage of a read/write operation, and a lack of an efficient algorithm to detect a coverage intersection.

In Chapter 2, a number of research efforts to tackle this problem were presented. These include the SUIF project [May91], the Paraphrase project [Kuc84], and many others [Li90][All87]. For example, it has been determined that, for certain special case inputs, efficient algorithms are available to determine exact dependence [Li90]. In [May91], a parallelizing compiler based on a systematic invocation of special case algorithms is presented. The application of these techniques to the CDFG generation module may reduce the number of dependencies which are inserted. This in turn will maximize the concurrency available in the flowgraph.

### **9.2.2 DSP Code Generation**

DSP processors are specialized processors which often use extensive pipelining, multiple independent memories and functional units, hardware looping, and other innovative techniques to allow real-time processing. While this is highly desirable, it also makes it difficult to generate code which can fully exploit their

capability. The main constraints come from the restrictions imposed by the pipelined execution and the parallel memory accesses.

In Chapter 7, a discussion on DSP code generation from a high level description was presented. All techniques attempt to generate code which is optimized for the target architecture. One offered a register allocation scheme which can remove many unnecessary memory accesses between operations, while another presented a code emission scheme which can better exploit the instruction-set of the target processor. This technique attempts to use single-cycle multiply-accumulate instructions, modular addressing for delay lines, zero-overhead looping, and parallel memory accesses.

There are two main approaches. One approach generates C as an intermediate language. The main drawback is the lack of support for DSP primitives in C usually results in inefficient code. The other approach directly synthesizes DSP code from a flowgraph specification. The work by Genin [Gen89] and Kim [Kim90] are important contributions in this area. For larger granularity DSP blocks, the use of a library of hand coded DSP blocks [Pow92] provides a simple and effective compromise. In our opinion, direct code generation from a DSP flowgraph holds the greatest promise due to its efficiency and generality.

### **9.2.3 Computation and Memory Estimation**

While the benchmarking technique presented in this thesis is quite effective, it has one main drawback, its simple code generation model. Each CDFG primitive operation is assumed to generate a set of assembly instructions which include the fetching of input operands from memory, the execution of the operation, and the storing of the output operand into memory. With this model, each CDFG operation takes the same amount of time and memory to execute, no matter how and where it is used. This allows for fast and accurate estimations. For an un-optimized compiler, this is usually the case. However, optimizing compilers such as those described in the above



subsection would deviate significantly from the code generation model above, making accurate estimations of computation time and memory usage more difficult.

To incorporate these optimizations into the estimation routines, it is necessary to model more accurately the underlying code generation strategy to take into account the register allocation, the pipelined execution, as well as the parallel data accesses. This work is currently being pursued at Georgia Tech [Cur92], with very promising results. The main drawback of this approach is that very low-level interactions must be modelled, yielding an estimation routine which is complex, slow, and not reusable. At larger granularity, the use of predefined optimized DSP library blocks can significantly reduce the overall estimation error.

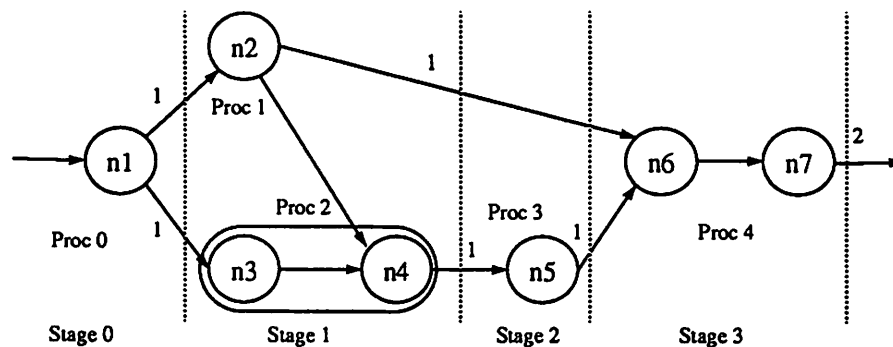
## **9.2.4 Scheduling for Heterogeneous System**

It is not uncommon for the target multiprocessor to be composed of different core processors. Usually, this occurs when each type is faster at performing some particular task than the rest. To exploit this advantage, a scheduling algorithm must attempt to schedule tasks onto those processors which can execute them the fastest. This can add another dimension of complexity to the existing scheduling problem because it may be advantageous to separate nodes which communicate heavily with each other to exploit specialized hardware features. Thus in addition to the trade-off between exploitation of communication cost and parallelism and granularity, this environment introduces a new trade-off between communication cost and varying processor computation speeds.

For classical schedulers which attempt to minimize the completion time of a single execution of the algorithm, the extension to heterogeneous processors has been addressed [Sih89]. In this formulation, the computation cost for a node is different for each processor type in the architecture, and the scheduler is extended to look at each

case. The main issue is the increase in the complexity of the cost function and the search space.

The same approach is not possible with the scheduling algorithm presented in this thesis. Because pipelining is exploited, a processor, once assigned to a pipeline stage, is restricted to perform only computations which can be scheduled on that pipeline stage, and no others. Consider the flowgraph as shown in Figure 9.1. Suppose



**FIGURE 9.1: Pipelining in Heterogeneous Systems Unsuccessful.**

that Proc 1 is of different type than the other processors, and suppose node n7 executes faster on Proc 1 than all other processors. It still cannot be scheduled on Proc 1, even if Proc 1 is insufficiently utilized. The pipeline clustering forces node n7 to be executed in stage 3, while Proc 1 is already assigned to stage 1. This natural grouping of neighboring computations to neighboring pipeline stages hinders the effective exploitation of pipelining on a heterogeneous environment. However, when there exist more than one processor per stage, the extension as discussed above can make a significant contribution.

### 9.2.5 Scheduling Data-dependent Computations

In compile-time scheduling, the quality of the solution is directly linked to the ability to accurately model what will occur at run-time. The better the modelling, the

more accurate the scheduling decisions. Data-dependent computations such as conditional and while loop constructs dramatically reduce a scheduler's ability to accurately model the run-time behavior.

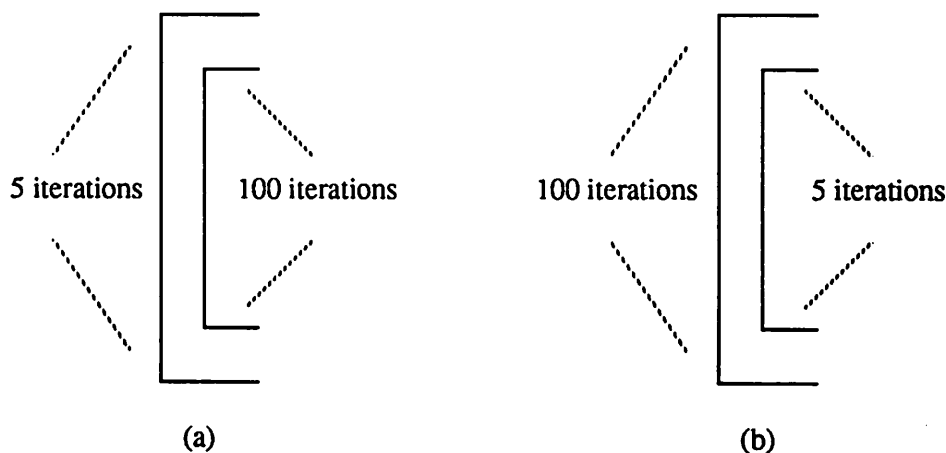
The conventional solution is to reject static scheduling and incur the (substantial) cost of dynamic scheduling. However, a number of research projects are underway to minimize the amount of dynamic scheduling needed [Lee88][Loe88]. This approach, coined *quasi-static* scheduling by Lee [Lee88], retains as much static scheduling as possible, and invokes dynamic scheduling only when absolutely necessary. To obtain a high quality quasi-static schedule, Ha [Ha92] proposes a set of possible local schedules or *profiles* of each data-dependent computations at compile time, and selects the profile which minimizes the expected run-time cost. The derivation of the profiles is dependent upon having a probability distribution of the run-time behavior of the computation. Ha's key contributions are the derivation of these distributions. The technique is used to schedule a number of dynamic constructs such as data-dependent iterations, recursions, and conditionals, with very promising results.

## 9.2.6 Loop Transformations

Transformations are changes in the flowgraph structure which can improve the final implementation, without altering the input-output relationships. The improvement may be in an increase in performance or concurrency, or a reduction in memory usage. In Chapter 4, a number of compiler optimizing transformations are introduced as part of the Silage to flowgraph translation process. In this section, we examine a number of loop transformations which can be useful in the McDAS environment.

Manipulating loop structures can greatly influence the final scheduling outcome. Examples of loop transformations include loop unrolling, loop merging, and loop interchange. Loop unrolling is already an essential part of the scheduling algorithm in this thesis, and is used to unroll iterations to expose more concurrency.

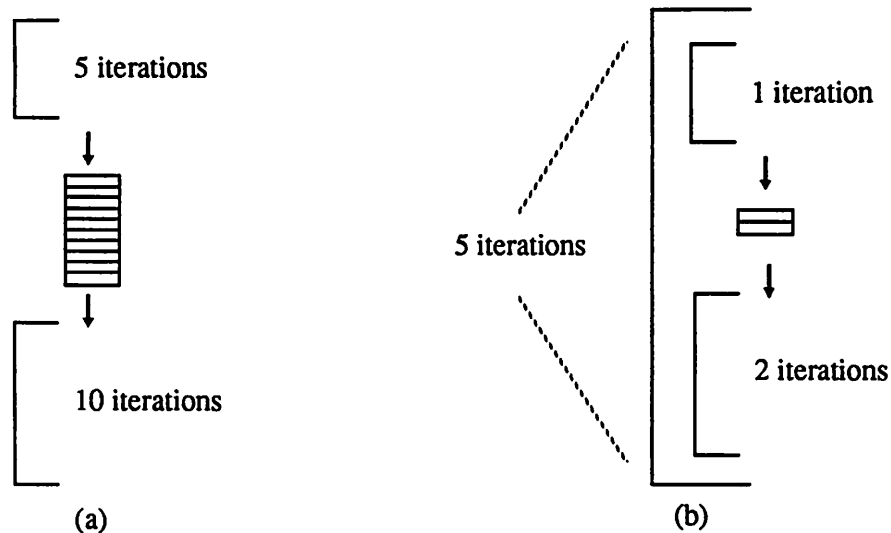
Loop interchange, a transformation where two nested loops are interchanged, can also significantly improve the scheduler. For example, the current implementation of the loop unrolling in the scheduler breaks a loop into a number of sub-nodes, each with a subset of the loop. No attempt is made to share an iteration between sub-nodes. As a result, the size of a single iteration often becomes a lower bound on the granularity level. Sometimes this granularity level is large enough to produce a poor load balancing. For example, consider a nested loop as shown in Figure 9.2a, where the outer



**FIGURE 9.2: Loop Interchange to Improve Load Balancing**

loop has 5 iterations and the inner loop has 100 iterations. Assume there are 4 available processors. The scheduler only sees 5 iterations to partition. A poor load balancing results since one processor is assigned 2 iterations, while the rest are assigned a single iteration. If a loop interchange is applied however (Figure 9.2b), the scheduler now sees 100 iterations to partition. A partition of 25 iterations for each processor yield a perfect load balancing. Nested loops occur often enough in signal processing to warrant the inclusion of the loop interchange transformation.

Loop merging, on the other hand, involves collapsing two consecutive loops into a single loop to save buffer memory storage. Consider Figure 9.2a where loop 1 has 5 iterations and loop 2 has 10 iterations. Each iteration of loop 1 produces 2 data samples while each iteration of loop 2 consumes 1 data sample. If the loops are



**FIGURE 9.3: Loop Merging to reduce memory requirements**

executed as shown, a buffer of 10 samples are accumulated at the end of loop 1. However, if the two loops are merged as in Figure 9.2b, the amount of buffer memory is reduced to 2.

Finally, it is possible to perform loop unrolling on the infinite time loop itself to expose more concurrency. Consider an application consisting only of a parallel loop of 4 iterations, and the available number of processors is 8. Instead of decomposing each iteration further to take advantage of the available processors, the time loop can be unrolled to expose the computation of 2 samples. Since this will have 2 parallel loops of 4 iterations, the 8 processors can be fully utilized.

For applications which are globally recursive, unrolling the time loop can sometime lead to an implementation with a higher throughput than is previously possible. A number of research projects are underway to investigate this transformation [Par89][Pot92]. The approach involves unrolling the time loop and applying a number of algebraic transformations in a certain order to move computations out of the recursion. While these techniques hold great promise, they currently only address fine-grain linear recursive applications. An extension to large-grain non-linear systems would greatly improve the applicability of this transformation.

## 9.3 SUMMARY

In this chapter, the main contributions of the thesis are summarized, and a number of topics which need further examination and research are discussed. In particular, enhancements to the scheduling and code generation strategies as well as several key loop transformations are proposed.

## REFERENCES

- [Ack79] W.B. Ackerman, J.B. Dennis: "VAL -- A Value-Oriented Algorithmic Language: Preliminary Reference Manual," MIT Laboratory for Computer Science Technical Report TR-218, MIT, Cambridge, Mass., June 1979.
- [Ack82] W.B. Ackerman: "Data Flow Languages," *Computer*, vol. 15, no. 2, February, 1982.
- [Ada74] T.L. Adam, K.M. Chandy, J.R. Dickson: "A Comparison of List Schedules for Parallel Processing Systems," *Communications of the ACM*, vol. 17, 1974, pp. 685-690.
- [All87] J.R. Allen, K. Kennedy: "Automatic Translation of FORTRAN programs to Vector Form," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 4, October, 1987.
- [Alr86] H. Alrutz: "Implementation of a Multi-Pulse Coder on a Single Chip Floating-Point Signal Processor," ICASSP'86, Tokyo, Japan, pp2367-2370.
- [Ann87] M. Annaratone, et al: "Warp Architecture: From Prototype to Production," *Proceedings of the 1987 National Computer Conference*, Chicago, Illinois, June 1987.

- [Arc86] J. Archibald, J. Baer: "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model," *ACM Transactions on Computer Systems*, vol. 4, no. 4, 1986.
- [Arv78] K. Arvind, P. Gostelow, W. Plouffe: "An Asynchronous Programming Language and Computing Machine," Department of Information and Computer Science Technical Report 114a, University of California, Irvine, December 1978.
- [Ash77] E.A. Ashcroft, W.W. Wadge: "Lucid, a Non-Procedural Language with Iteration," *Communication of the ACM*, vol. 20, no. 7, July 1977, pp. 519-526.
- [Att88] "WE DSP32C Digital Signal Processor," *AT & T Information Manual*, December 1988.
- [Bar91] T. Barnwell III, V. Madisetti, S. McGrath: "The Georgia Tech Digital Signal Multiprocessor," submitted to *IEEE Transactions on Signal Processing*, July 1991.
- [Bau90] K. Baudendistel, J.H. McClellan: "Code Generation for the AT&T DSP32", *ICASSP 90*, Albuquerque, NM.
- [Bok81] S.H. Bokhari: "Shortest Tree Algorithm for Optimal Assignments Across Space and Time in a Distributed Processor System," *IEEE Transactions on Software Engineering*, vol. SE-7, no. 6, November 1981.
- [Bok88] S.H. Bokhari: "Partitioning Problems in Parallel, Pipelined, and Distributed Computing," *IEEE Transactions on Computers*, vol. 37, no. 1, January 1988.
- [Bac78] J. Backus: "Can Programming Be Liberated from the Von Neumann Style? A Functional Style and Its Algebra of Programs," *Communications of the ACM*, vol. 21, no. 8, August 1978.



- [Ber91] G. Berry: "A hardware implementation of pure ESTEREL," *ACM Workshop on Formal Methods in VLSI Design*, January, 1991.
- [Cam90] J.P. Campbell Jr, V. Welch, T. Tremain: "An Expandable Error-Protected 4800 bps CELP coder," *ICASSP 1985*.
- [CCI89] Description of Reference Model 8 (RM8), Document 525, CCITT SGXV, Working Party XV/4, Specialist Group for Visual Telephony, 1989.
- [Che84] S.C. Chen: "Large-Scale and High-Speed Multiprocessor System for Scientific Applications: Cray X-MP Series," In *High-Speed Computation*, NATO ASI Series, vol. F7, J.S. Kowalik Ed. Springer-Verlag, New York, 1984, pp. 59-67.
- [Che92] D.C. Chen, et al.: "An Integrated System for Rapid Prototyping of High Performance Algorithm Specific Data Paths," submitted to *International Conference on Application-Specific Array Processors*, Berkeley California, August 1992.
- [Chu80] W.W. Chu, L.J. Holloway, M.T. Lan, K. Efe: "Task Allocation in Distributed Data Processing," *Computer*, November 1990, pp. 57-69.
- [Cof72] E.G. Coffman Jr, R.L. Graham: "Optimal Scheduling for Two Processor Systems," *Acta Informatica*, vol. 1, 1972, pp. 200-213.
- [Cof76] E.G. Coffman Jr. (Ed.): *Computer and Job-Shop Scheduling Theory*, John Wiley & Sons, New York, 1976.
- [Coo71] S.A. Cook: "The Complexity of Theorem Proving Procedures," *Proc. Third ACM Symposium on Theory of Computing*, pp. 24-30, 1971.
- [Cov87] C.D. Covington, G.E. Carter, D.W. Summers: "Graphic Oriented Signal Processing Language - GOSPL," *ICASSP 1987*.

- [Cro83] R.E. Crochiere, L.R. Rabiner: "Multirate Digital Signal Processing," Englewood Cliffs, New Jersey, Prentice Hall, 1983.
- [Cur92] B.A. Curtis, V.K. Madiseti: "Task Scheduling in the Georgia Tech Digital Signal Multiprocessor," *ICASSP'92*, San Francisco, CA, March 1992.
- [Dau87] J.W. Daugherty: "Using the DSP32 in Speech Processing Applications," *AT&T Memorandum*, Dept. 54127, May 1987.
- [Dar81] J. Darlington, M. Reeve: "Alice -- A Multiprocessor Reduction Machine for the Parallel Evaluation of Applicative Languages," *Proc. 1981 ACM Conf. Functional Programming Languages and Computer Architecture*, 1981, pp. 65-75.
- [Dav86] J. Davies, et al.: "The KAP/205: An Advanced Source-to-Source Vectorizer for the Cyber 205 Supercomputer," *Proceedings of the 1986 International Conference on Parallel Processing* (St. Charles, Illinois, Aug. 19-22). IEEE Press, New York, 1986, pp. 827-832.
- [Fly72] M.J. Flynn: "Some Computer Organizations and their Effectiveness," *IEEE Transactions on Computers*, September 1972, C-21(9), pp. 948-960.
- [For88] H.R. Forren: "Multiprocessor Design Methodology for Real-Time DSP Systems Represented by Shift-Invariant Flow Graphs," PhD thesis, Georgia Institute of Technology Technical Report, 1988.
- [Fre89] K. Frenkel: "HDTV and the Computer Industry," *Communications of the ACM*, November 1989.
- [Gar79] M.R. Garey and D.S. Johnson: "Computers and Intractability: A Guide to the Theory of NP-Completeness," W.H. Freeman and Company, San Francisco, 1979.

- [Gel92] P.R. Gelabert, T.P. Barnwell III: "Optimal Automatic Periodic Multiprocessor Compiler for Multi-Bus Networks," *ICASSP'92*, San Francisco, CA, March 1992.
- [Gen89] D.R. Genin, et al.: "System Design, Optimization, and Intelligent Code Generation for Standard DSP," *ISCAS*, May 1989.
- [Gir87] E. F. Girczyc: "Loop Winding -- A Data Flow Approach to Functional Pipelining," *ISCAS'87*, pp. 382-385.
- [Gon77] M.J. Gonzalez, Jr.: "Deterministic Processor Scheduling," *Computing Surveys*, vol. 9, no. 3, September 1977.
- [Gra69] R.L. Graham: "Bounds on certain multiprocessing anomalies," *SIAM Journal of Applied Mathematics*, vol. 17, no. 2, March 1969, pp. 416-429.
- [Gue91] P.L. Guernic, T. Gautier, M. Borgne, C. Maire: "Programming Real-Time Applications with SIGNAL," IRISA, Campus de Beaulieu, France.
- [Gur80] J. Gurd, I. Watson: "Data Driven Systems for High Speed Parallel Computing: Part 1: Structuring Software for Parallel Execution; Part 2: Hardware Design," *Computer Design*, June and July 1980, pp. 91-100, 97-106, respectively.
- [Ha92] Soonhoi Ha, "Compile-Time Scheduling of Dataflow Program Graphs with Dynamic Constructs," *Ph.D. Thesis* in preparation, University of California, Berkeley, 1992.
- [Har86] D. Harrison: "Data Management and Graphics Editing in the Berkeley Design Environment," *Proceedings IEEE International Conference on Computer Aided Design*, November 1986.

- [Hil85] P. Hilfinger: "SILAGE, A High Level Language and Silicon Compiler for Digital Signal Processing," *Proceedings IEEE CICC Conference*, Portland, May 1985.
- [Hon84] M.L. Honig, and D.G. Messerschmitt: *Adaptive Filters: Structures, Algorithms, and Applications*, Kluwer Academic Publishers, 1984.
- [Hu61] T.C. Hu: "Parallel Sequencing and Assembly Line Problems," *Operations Research*, vol. 9, no. 6, 1961, pp. 841-848.
- [Hwa89] J.J. Hwang, Y.C. Chow, F.D. Anger, C.Y. Lee: "Scheduling Precedence Graphs in Systems with Interprocessor Communication Times," *SIAM Journal of Computing*, vol. 18, 1989, pp. 244-257.
- [Int86] "iPSC Program Development Guide," *Intel Corporation*, November, 1986.
- [Jac90] Geert Jacobs: "Multirate Digital Signal Processing," *EDC Internal Document*, Nov. 1990.
- [Joh75] D.B. Johnson: "Finding All the Elementary Circuits of a Directed Graph," *SIAM Journal of Computing*, vol. 4, no. 1, March 1975.
- [Kan87] R. Kannan: "Minkowski's Convex Body Theorem and Integer Programming," *Mathematics of Operations Research*, vol. 12, no. 3, August 1987.
- [Kar72] R.M. Karp: "Reducibility Among Combinatorial Problems," *Complexity of Computer Computations* (R.E. Miller, Ed.), Plenum Press, 1972, pp. 85-103.
- [Kem90] D. Kemp, R. Sueda, T. Tremain: "An Evaluation of 4800 bps Voice Coders", *ICASSP 1990*, S4.21, pp200-203.
- [Kim90] B.M. Kim, T.P. Barnwell III: "Resource Allocation and Code Generation for Pointer-Based Pipelined DSP Multiprocessors," *ISCAS'90*.

- [Koh75] W.H. Kohler: "A Preliminary Evaluation of the Critical Path Method for Scheduling Tasks on Multiprocessor Systems," *IEEE Transactions on Computers*, vol. C-24, 1975, pp. 1235-1238.
- [Koh89] W. Koh, A. Yeungk, P. Hoang, J. Rabaey: "A Configurable Multiprocessor System for DSP Behavioral Simulation," *ISCAS Symposium*, May 1989.
- [Kri87] R. Krishnamurti: "Reconfigurable Parallel Architectures for Special Purpose Computing," PhD thesis, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA, 1987.
- [Kro92] T. Krol, J.V. Meerbergen, C. Niessen, W. Smits, J. Huisken: "The Sprite Input Language, An Intermediate format for High Level Synthesis," *Philips Research Laboratories*, The Netherlands.
- [Kuc84] D.J. Kuck, R.H. Kuhn, B. Leasure, M. Wolfe: "The Structure of an Advanced Retargetable Vectorizer," in *Tutorial on Supercomputers: Designs and Applications*, K. Hwang, Ed., IEEE Press, New York, 1984, pp. 163-178.
- [Lee85] S. Lee, C. Hodges, T. Barnwell III: "An SSIMD Compiler for the Implementation of Linear Shift-Invariant Flow Graphs," *Proceedings ICASSP*, March 1985.
- [Lee87] E.A. Lee, D.G. Messerschmitt: "Synchronous Data Flow," *IEEE Proceedings*, September 1987.
- [Lee88] E.A. Lee: "Recurrences, Iterations, and Conditionals in Statically Scheduled Block Diagram Languages," *VLSI Signal Processing III*, IEEE Press, 1988.
- [Lee89a] E.A. Lee: "An Informal Study of Block Diagram Linguistics," U.C. Berkeley Internal Report, November 1989.

- [Lee89b] E.A. Lee: "Consistency in Dataflow Graphs," U.C. Berkeley Internal Report, November 1989.
- [Lee89c] E.A. Lee, et al: "Gabriel: A Design Environment for DSP," *IEEE Transactions on ASSP*, vol. 37, no. 11, November 1989.
- [Lee89d] E.A. Lee, S. Ha: "Scheduling Strategies for Multiprocessor Real-Time DSP," *GLOBECOM*, Dallas Texas, November 1989.
- [Lee90] E.A. Lee, J.C. Bier: "Architectures for Statically Scheduled Dataflow," UCB/ERL M89/129, May 1990.
- [Lei83] C. E. Leiserson, F. M. Rose, J. B. Saxe: "Optimizing Synchronous Circuitry by Retiming," *Proceedings of the Third Caltech Conference on VLSI*, Computer Science Press, 1983, pp. 23-26.
- [Len78] J.K. Lenstra, A.H. Rinnooy Kan: "Complexity of Scheduling under Precedence Constraints," *Operations Research*, vol. 26, no. 1, January 1978.
- [Loe88] C. Loeffler, A. Ligtenberg, H. Bheda, and G. Moschytz: "Hierarchical Scheduling system for Parallel Architectures," *Proceedings of Euco*, Grenoble, September, 1988.
- [Li90] Z. Li, P. Yew: "Practical Methods for Exact Data Dependency Analysis," *Proceedings of the Second Workshop on Languages and Compilers for Parallel Computing*, 1989.
- [Lov88] T. Lovett, S. Thakkar: "The Symmetry Multiprocessor System," *Proc. 1988 Int'l Conf. of Parallel Processing*, University Park, Pennsylvania, 303-310.
- [May91] D.E. Maydan, J.L. Hennessy, M.S. Lam: "Efficient and Exact Data Dependence Analysis," *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991.

- [McC89] C. McCreary, H. Gill: "Automatic Determination of Grain Size for Efficient Parallel Processing," *Communications of the ACM*, Vol. 32, No. 9, September 1989.
- [Mes84] D.G. Messerschmitt: "A Tool for Structured Functional Simulation," *IEEE J. Select. Areas Commun.*, vol. SAC-2, January 1984.
- [Mot90] "DSP96002 User's Manual," *Motorola*, 1990.
- [Nag84] S. Nagashima, Y. Inagami, T. Odaka, S. Kawabe: "Design Consideration for a high speed vector processor: The Hitachi S-810," *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers. ICCD 84.* (Port Chester, N.Y. Oct. 8-11). IEEE Press, New York, 1984, pp. 238-243.
- [Ost86] A. Osterhaug: "Guide to Parallel Programming on Sequent Computer Systems," *Sequent Computer Systems, Inc.*, 1986.
- [Pad86] D.A. Padua, M.J. Wolfe: "Advanced Compiler Optimizations for Supercomputers," *Communications of the ACM*, vol. 29, no. 12, December 1986.
- [Par89] K.K. Parhi: "Rate-Optimal Fully-Static Multiprocessor Scheduling of Data-Flow Signal Processing Programs," *ISCAS*, May 1989.
- [Rav83] C.V. Ravishankar, J.R. Goodman: "Cache Implementation for Multiple Microprocessors," *Proceedings Compton Spring'83*, February 1983, pp. 346-350.
- [Par89] K.K. Parhi, D.G. Messerschmitt: "Pipeline Interleaving and Parallism in Recursive Filters - Parts I & II", *IEEE Transactions on Signal Processing*, pp. 1099-1117 & pp. 1118-1134, July 1989.

- [Pap84] M.S. Papamarcos, J.H. Patel: "A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories," *Proc. Eleventh International Symposium on Computer Architecture*, June 1984, pp. 348-359.
- [Pot92] M. Potkonjak, J. Rabaey: "Fast Implementation of Recursive Programs Using Transformations," *ICASSP*, March 1992.
- [Pow92] D.B. Powell, E.A. Lee, W.C. Newman: "Direct Synthesis of Optimized DSP Assembly Code from Signal Flow Block Diagrams," *ICASSP'92*, San Francisco, CA.
- [Pri91] H. Printz: "Automatic Mapping of Large Signal Processing Systems to a Parallel Machine," PhD thesis, Carnegie Mellon Univ., May 1991.
- [Rab91] J. Rabaey, C. Chu, P. Hoang, M. Potkonjak: "Fast Prototyping of Data Path Intensive Architecture," *IEEE Design and Test*, vol. 8, no. 2, pp. 40-51, 1991.
- [Sak78] H. Sakoe, and S. Chiba: "Dynamic Programming Algorithm Optimization for Spoken Word Recognition," *IEEE Trans. Acoustics, Speech, and Signal Processing*, vol. ASSP-26, pp. 43-49, 1978.
- [Sar89] V. Sarkar: "Partitioning and Scheduling Parallel Programs for Multiprocessors," Research Monographs in Parallel and Distributed Computing, MIT Press, 1989.
- [Sch85] D.A. Schwartz: "Synchronous Multiprocessor Realizations of Shift-Invariant Flow Graphs," PhD thesis, Georgia Institute of Technology, 1985.
- [Sch88] C. Scheers: "User's Manual for the S2C Silage To C Compiler," Internal Document, IMEC Laboratory, Leuven Belgium, August 1988.
- [Sed88] R. Sedgewick: "Algorithms," Addison-Wesley, 1988.



- [Sha87] K.S. Shanmugan, G.J. Minden, E. Komp, T.C. Manning, E. R. Wiswell: "Block-Oriented System Simulator (BOSS)," Telecommunication Lab., Univ. Kansas, Internal Memo, 1987.
- [Sih89] G. Sih, E.A. Lee: "Dynamic-Level Scheduling for Heterogeneous Processor Networks," *IEEE Symposium on Parallel and Distributed Processing*, December 1989.
- [Sih90] G. Sih, E.A. Lee: "A Multiprocessor Scheduling Strategy," Electronics Research Laboratory Internal Report, U.C. Berkeley, 1990.
- [Ski88] D. Skillicorn: "A Taxonomy for Computer Architectures," *Computer*, November 1988.
- [Slu80] R. J. Sluyter, N. J. Kotmans, A. V. Leeuwaarden: "A Novel Method for Pitch Extraction from Speech and a Hardware Model Applicable to Vocoder Systems," *ICASSP'80*, pp. 45-48.
- [Sto77] H.S. Stone: "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," *IEEE Transactions on Software Engineering*, vol. SE-3, no. 1, January 1977.
- [Sto85] S.J. Stolfo, D. Miranker: "DADO: A Parallel Processor for Expert Systems," *Advanced Computer Architecture* (Agrawal, ed.) IEEE Computer Society Press, 1985.
- [Sun91] J.S. Sun, R.W. Brodersen: "System Module Interface Design in Siera," Internal Document, U.C. Berkeley, November 1991.
- [Tex92] "TMS320C30 C Compiler Manual," *Texas Instruments*, 1992.
- [Thi87] "Connection Machine Model CM-2 Technical Summary," *Thinking Machines Corporation*, April, 1987.

- [Ull75] J.D. Ullman: "NP-Complete Scheduling Problem," in *Journal of Computer and System Sciences*, vol. 10, 1975, pp. 384-393.
- [Wal87] R.A. Walker, D.E. Thomas: "Design Representation and Transformation in the System Architect's Workbench," *ISCAS'87*, pp. 166-169.
- [Wal89] G. K. Wallace, "Technical Description of the Proposed JPEG Baseline Standard for Color Image Compression," *EI'89*, Boston, October, 1989.
- [Wan88] E. Wang: "A Compiler for Silage," Master's Thesis, U.C. Berkeley, 1988.
- [Wat92] G. F. Watson: "Solid State," *IEEE Spectrum*, January 1992.
- [Yu84] W.H. Yu: "LU Decomposition on a Multiprocessing System with Communication Delay," PhD thesis, U.C. Berkeley, 1984.
- [Zis87] M.A. Zissman, G.C. O'Leary: "A Block Diagram Compiler for a Digital Signal Processing MIMD Computer," *ICASSP 1987*.

# Appendix **A**

## Flowgraph Implementation

This appendix describes the implementation of the CDFG flowgraph. The CDFG is stored in the OCT database [Har86], which has been extended to support a flowgraph policy. This policy is meant to capture only the structure of the flowgraph, not its behavior. As such, it describes the only interconnection of nodes and edges. However, each node has a pointer to where specific information on the behavior of the node is found. The OCT schematic editor VEM [Har86] can be used to display the flowgraph schematics. A library of behavioral primitives (addition, multiplication, delay, decimation, etc.) is provided as a start, and is shown in Table A.1. The user can easily add his own primitives to the library. By storing only the structure, the same representation can be used to support a wide number of projects such as HYPER [Rab91], CADDI [Che92], and Aloha [Sun91]. In addition, an ASCII format flowgraph description language, supported by a set of C data-structures, is available. This ASCII flowgraph language (AFL) has a one to one correspondence to the OCT policy, and serves as an easy readable user-interface into the OCT database. A tool to translate the AFL format to OCT, and vice-versa, has been developed. Developers not familiar with OCT can work with the AFL C data-structures, and interface to OCT when needed.

TABLE A.1: Table of Flowgraph Operators

Description	Function	Inputs	Outputs	Arguments
add	“+”	(In1 In2)	(Out)	None
subtract	“-”	(In1 In2)	(Out)	None
multiply	“*”	(In1 In2)	(Out)	(coef)
divide	“/”	(In1 In2)	(Out)	None
negate	“-”	(In)	(Out)	None
inc	“++”	(In)	(Out)	None
dec	“--”	(In)	(Out)	None
shift_left	“<<”	(In LShift)	(Out)	(shift)
shift_right	“>>”	(In RShift)	(Out)	(shift)
and	“&”	(In1 In2)	(Out)	None
or	“ ”	(In1 In2)	(Out)	None
exor	“^”	(In1 In2)	(Out)	None
inv	“!”	(In)	(Out)	None
equal	“=”	(In)	(Out)	None
cond_eq	“==”	(In1 In2)	(Out)	None
cond_ne	“!=”	(In1 In2)	(Out)	None
cond_g	“>”	(In1 In2)	(Out)	None
cond_ge	“>=”	(In1 In2)	(Out)	None
cond_l	“<”	(In1 In2)	(Out)	None
cond_le	“<=”	(In1 In2)	(Out)	None
bit	“bit”	(In)	(Out)	(bit)
bitselect	“bitselect”	(In)	(Out)	(bit width)
bitmerge	“bitmerge”	(In1 In2)	(Out)	None
read	“read”	(Index)	(Out)	(array_name index)
write	“write”	(In Index)	None	(array_name index)
mux	“mux”	(Cond In1 In2)	(Out)	None

**TABLE A.1: Table of Flowgraph Operators**

Description	Function	Inputs	Outputs	Arguments
constant	“const”	None	(Out)	(value)
cast	“cast”	(In)	(Out)	(type)
func	“func”	(*)	(*)	(*)
if	“if”	Cond *)	(*)	None
else	“else”	(Cond *)	(*)	None
iteration	“iteration”	(*)	(*)	(index min max step)
do	“do”	(*)	(*)	(avg)
exit	“exit”	(Cond In)	None	None
nop	“nop”	(In)	(Out)	None
delay	“@”	(In delay init)	(Out)	(delay init)
lpdelay	“#”	(In delay init)	(Out)	(delay init)
input	“input”	None	(Out)	None
output	“output”	(In)	(Out)	None
upsampling	“upsample”	(In scale phase)	(Out)	(scale phase)
dnsampling	“dnsample”	(In scale phase)	(Out)	(scale phase)
interpolate	“interpolate”	(In scale phase)	(Out)	(scale phase)
decimate	“decimate”	(In scale phase)	(Out)	(scale phase)
timemux	“timemux”	(*)	(Out)	(cnt)
timedemux	“timedemux”	(In phase)	(*)	(cnt)

## A.1 Flowgraph Structure

In this section, we describe the structure of the CDFG. The most interesting feature is its support for structure hierarchy. The AFL format is described first, followed by the corresponding representation in OCT.

### A.1.1 AFL Format

A CDFG is described as a composition of nodes, data edges, and control edges.

Each graph contains the following information:

```
(graph
  (name string)
  (class string)
  (arguments list)
  (attributes list)
  (model list)
  (in_edges edge-list)
  (in_control edge-list)
  (out_edges edge-list)
  (out_control edge-list)
  (nodelist node-definitions)
  (edgelist edge-definitions)
  (controllist edge-definitions)
)
```

Each graph has a name. The class of a graph is either “MODULE” or “LEAF” depending on whether it is composed of still smaller nodes or if it is a leaf-node in the hierarchy. Arguments of a graph are parameters which influence the behavior of the graph. An example of arguments of a graph is the default value of the delay in a DELAY leaf graph. Attributes of a graph, on the other hand, store information which are tool-specific. An example of an attribute of a graph may be the processor for which this graph will be executed on. The model field points to the behavioral definition of the graph. To define an interface to the next hierarchy level, it is necessary to declare the in\_edges, in\_control, out\_edges, and out\_control of the graph. Finally, the list of nodes, data edges, and control edges which belong to the graph are given in nodelist, edgelist, and controllist.

A node is defined as follows:

```

(node
  (name string)
  (class string)
  (master string)
  (arguments list)
  (attributes list)
  (in_edges edge-list)
  (in_control edge-list)
  (out_edges edge-list)
  (out_control edge-list)
)

```

Each node is an instance of an operator, which is defined in the **master** field. The operator may be primitive, such as add, multiply, etc., or it can be a graph itself, in which case, the node is called a *hierarchical* node. Arguments of a node may include the value for a constant node, the iteration index and bounds for an iteration node, etc. Attributes of a node may be its level in a level labelling calculation or a computation time cost, etc. Finally, an edge is defined as follows:

```

(edge
  (name string)
  (class string)
  (arguments list)
  (attributes list)
  (in_nodes edge-list)
  (out_nodes edge-list)
)

```

An argument of an edge is its type, which is taken from the Silage data types. It can either be a boolean, integer, fixed-point, or floating point. An attribute of an edge may be its register allocation.

There is a great deal of redundancy in the CDFG structure to allow the tools to traverse the graph in the most efficient way possible.

### A.1.2 OCT Format

The OCT format has a 1-to-1 correspondence with the AFL format described above. Thus, only a overview will be given here. In essence, a cell, representing a graph, is composed of cell instances, representing nodes, and nets, representing data edges. Terminals serve as interfaces between cell instances and nets, and contains a DIRECTION property. Arguments are stored in the PARAMETERS bag, and attributes are stored in the ATTRIBUTES bag. The MODEL bag, attached to a cell facet, shows where to find the behavioral description of the cell. Control precedences are stored in a PRECEDENCE bag, attached to the cell facet. The relationship is shown in Figure A.1

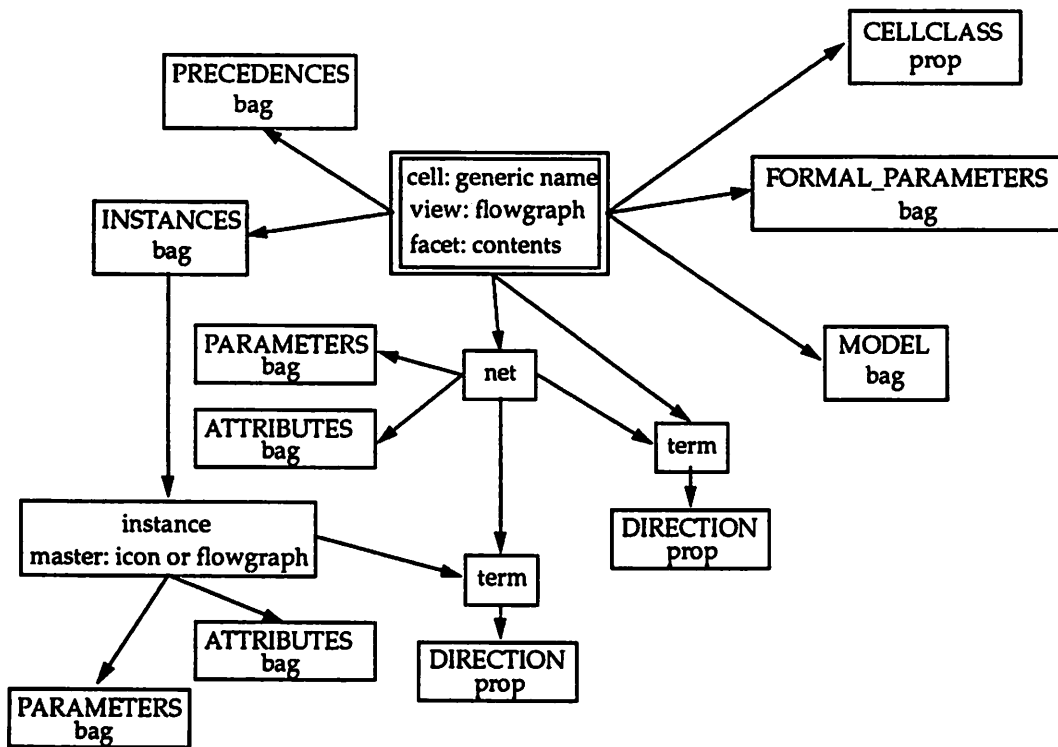


FIGURE A.1 : OCT Flowgraph Policy

## A.2 C Data Structure

This section gives the C data structures which support the AFL format.



```
/*
* Definitions of data structures used to represent flow graphs
*/

/* Generic pointer structure */
typedef char *pointer;

/* Pointer definitions */
typedef struct LIST *ListPointer;
typedef struct NODE *NodePointer;
typedef struct EDGE *EdgePointer;
typedef struct GRAPH *GraphPointer;

/* Generic LIST structure */

typedef enum { IntNode, RealNode, CharNode, ListNode, GraphNode, EdgeNode,
              NodeNode, ArrayNode, NullNode }
              EntryType;

typedef struct LIST {
    char *Label;
    EntryType Type;
    pointer Entry;
    ListPointer Next;
} LIST;

/* Graph describing data structures */

typedef struct GRAPH {
    char *Name;
    char *Class;
    EdgePointer EdgeList;
    NodePointer NodeList;
}
```

```

EdgePointer ControlList;
ListPointer Parents;
ListPointer InEdges;
ListPointer OutEdges;
ListPointer InControl;
ListPointer OutControl;
ListPointer Arguments;
ListPointer Attributes;
ListPointer Model;
GraphPointer Next;
pointer Extension; /* Attach your program structures here */
int Token; /* Bogus field available to the service Programs (for ordering e.g.) */
} GRAPH;

```

```

typedef struct NODE {
    char *Name;
    char *Class;
    GraphPointer Master;
    ListPointer InEdges;
    ListPointer OutEdges;
    ListPointer InControl;
    ListPointer OutControl;
    ListPointer Arguments;
    ListPointer Attributes;
    NodePointer Next;
    pointer Extension; /* Attach your program structures here */
    int Token; /* Bogus field available to the service Programs (for ordering e.g.) */
} NODE;

```

```

typedef struct EDGE {
    char *Name;
    char *Class;
    ListPointer InNodes;

```

```

ListPointer OutNodes;
ListPointer Arguments;
ListPointer Attributes;
EdgePointer Next;
pointer Extension; /* Attach your program structures here */
int Token; /* Bogus field available to the service Programs (for ordering e.g. */
} EDGE;

```

### A.3 A Sample AFL Flowgraph

Given the following Silage program:

```

#define word fix<16,2>
#define word2 fix<32,4>

func main(A, B : word) K, L: word =
begin
  K = joe(A,B);
  L = joel(A,B);
end;

func joe(C, D : word) E: word =
begin
  E = C + D;
end;

func joel(C, D : word) F: word =
begin
  F = C - D;
end;

```

The corresponding AFL CDFG is given as:

```

(GRAPH
  (NAME main)
  (CLASS MODULE)
  (MODEL (
    (model_class internal)
    (model_name func))
  )
  (NODELIST
    (NODE
      (NAME n2)
      (CLASS Hierarchy)
      (MASTER joe)
      (ATTRIBUTES
        ((Silage ("func.sil" 6 15 )))
      )
    )
  )
)

```

```

      (IN_EDGES (A B ))
      (OUT_EDGES (K ))
    )
  (NODE
    (NAME n3)
    (CLASS Hierarchy)
    (MASTER joe1)
    (ATTRIBUTES
      ((Silage ("func.sil" 7 16 )))
    )
    (IN_EDGES (A B ))
    (OUT_EDGES (L ))
  )
)
(NODE
  (NAME n4)
  (CLASS data)
  (MASTER input)
  (OUT_EDGES (A ))
)
(NODE
  (NAME n5)
  (CLASS data)
  (MASTER input)
  (OUT_EDGES (B ))
)
(NODE
  (NAME n6)
  (CLASS data)
  (MASTER output)
  (IN_EDGES (K ))
)
(NODE
  (NAME n7)
  (CLASS data)
  (MASTER output)
  (IN_EDGES (L ))
)
)
(EDGELIST
  (EDGE
    (NAME A)
    (CLASS input)
    (ARGUMENTS
      ((type fix<16,2>))
    )
    (ATTRIBUTES
      ((Silage ("func.sil" 4 11 ))(lhs 1))
    )
    (IN_NODES (n4 ))
    (OUT_NODES (n2 n3 ))
  )
  (EDGE
    (NAME B)
    (CLASS input)
    (ARGUMENTS
      ((type fix<16,2>))
    )
    (ATTRIBUTES

```

```

        ((Silage ("func.sil" 4 14 ))(lhs 1))
    )
    (IN_NODES (n5 ))
    (OUT_NODES (n2 n3 ))
)
(EDGE
  (NAME K)
  (CLASS output)
  (ARGUMENTS
    ((type fix<16,2>))
  )
  (ATTRIBUTES
    ((lhs 1)(Silage ("func.sil" 4 29 )))
  )
  (IN_NODES (n2 ))
  (OUT_NODES (n6 ))
)
(EDGE
  (NAME L)
  (CLASS output)
  (ARGUMENTS
    ((type fix<16,2>))
  )
  (ATTRIBUTES
    ((lhs 1)(Silage ("func.sil" 4 32 )))
  )
  (IN_NODES (n3 ))
  (OUT_NODES (n7 ))
)
)
)
)
(GRAPH
  (NAME joe)
  (CLASS MODULE)
  (MODEL (
    (model_class internal)
    (model_name func))
  )
  (IN_EDGES (C D ))
  (OUT_EDGES (E ))
  (NODELIST
    (NODE
      (NAME n1)
      (CLASS data)
      (MASTER add)
      (ATTRIBUTES
        ((Silage ("func.sil" 12 17 )))
      )
      (IN_EDGES (C D ))
      (OUT_EDGES (E ))
    )
  )
)
)
(EDGELIST
  (EDGE
    (NAME C)
    (CLASS input)
    (ARGUMENTS
      ((type fix<16,2>))
    )
  )
)

```

```

    )
    (ATTRIBUTES
      ((Silage ("func.sil" 10 10 ))(lhs 1))
    )
    (IN_NODES (parent ))
    (OUT_NODES (n1 ))
  )
  (EDGE
    (NAME D)
    (CLASS input)
    (ARGUMENTS
      ((type fix<16,2>))
    )
    (ATTRIBUTES
      ((Silage ("func.sil" 10 13 ))(lhs 1))
    )
    (IN_NODES (parent ))
    (OUT_NODES (n1 ))
  )
  (EDGE
    (NAME E)
    (CLASS output)
    (ARGUMENTS
      ((type fix<16,2>))
    )
    (ATTRIBUTES
      ((lhs 1)(Silage ("func.sil" 10 28 )))
    )
    (IN_NODES (n1 ))
    (OUT_NODES (parent ))
  )
)
)
)
(GRAPH
  (NAME joe1)
  (CLASS MODULE)
  (MODEL (
    (model_class internal)
    (model_name func))
  )
  (IN_EDGES (C D ))
  (OUT_EDGES (F ))
  (NODELIST
    (NODE
      (NAME n0)
      (CLASS data)
      (MASTER minus)
      (ATTRIBUTES
        ((Silage ("func.sil" 17 17 )))
      )
      (IN_EDGES (C D ))
      (OUT_EDGES (F ))
    )
  )
)
)
(EDGELIST
  (EDGE
    (NAME C)

```

```

    (CLASS input)
    (ARGUMENTS
      ((type fix<16,2>))
    )
    (ATTRIBUTES
      ((Silage ("func.sil" 15 11 ))(lhs 1))
    )
    (IN_NODES (parent ))
    (OUT_NODES (n0 ))
  )
(EDGE
  (NAME D)
  (CLASS input)
  (ARGUMENTS
    ((type fix<16,2>))
  )
  (ATTRIBUTES
    ((Silage ("func.sil" 15 14 ))(lhs 1))
  )
  (IN_NODES (parent ))
  (OUT_NODES (n0 ))
)
(EDGE
  (NAME F)
  (CLASS output)
  (ARGUMENTS
    ((type fix<16,2>))
  )
  (ATTRIBUTES
    ((lhs 1)(Silage ("func.sil" 15 29 )))
  )
  (IN_NODES (n0 ))
  (OUT_NODES (parent ))
)
)
)
)

```

# Appendix **B**

## Silage To Flowgraph Implementation

This appendix describes the implementation of the basic Silage To CDFG translator. It is performed in two phases: The first phase is the Silage front end, which parses the Silage program and builds the necessary data structures for the construction of a flowgraph. The second phase, the CDFG generator, builds a CDFG flowgraph from the data structures.

### B.1 Silage Frontend

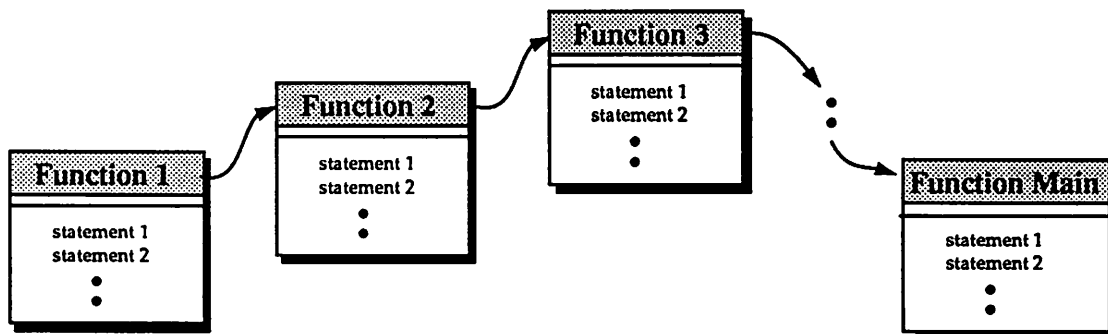
The Silage front end is derived from the IMEC Silage simulator. There are nine phases in the Silage front end. They are listed below in order of execution.

- **Parser():** This module uses the Lex (the Unix scanner generator), and YACC (the Unix LALR parser generator) program to perform the usual lexical and syntactic analysis on the input Silage program. The output is a syntax tree.
- **pass1():** This module creates a linked list of function definition templates from the parse tree.
- **pass2():** This module fills in all input, output, and local parameters of each function.



- **pass3():** This module builds the templates for every statement and iteration block of each function. It also builds the left hand side (LHS) of each statement.
- **pass4():** This module builds the right hand side (RHS) of each statement.
- **pass5():** This module orders the function definitions from the lowest level to the highest level function. The main function is placed last. No recursive definitions are allowed.
- **pass6():** This module topologically orders the statements within each function definition. Because the range of the array indices has not been analyzed yet, it is not possible to determine dependencies between statements containing arrays. Thus, any function that has iterations or arrays are not *processed*. The user is expected to have written ordered Silage statements in these cases.
- **pass7():** This module evaluates all manifest expressions, including those inside iterations to determine the range of all array indices and all delay values of delayed signals. This range is needed for the data dependency analysis of array signals.
- **pass8():** This module performs the type checking of signals. It consists of a *type deduction* phase and a *type enforcement* phase. For each function definition, the routine traverses the statements from input to output and deduces the type of the LHS from the types and operations of the RHS. When deduction is not possible, the type is left as *undefined*. The type enforcement phase then traverses the statements from output back to input and enforces the type of the result back on the inputs. If the type enforced is different than the type deduced, and there is no *cast* function, an error is declared. If the deduced type is undefined, the enforced type is taken. Within this process, all constants are assigned a type suitable for the operator being applied to it. Functions which were not ordered in pass6() are also ignored here.

Of all the nine phases, the type checking phase of pass 8 is the weakest. It was removed and reimplemented in the flowgraph generator phase, where the flowgraph structure allows for a simpler implementation. The Silage front end terminates with a linked-list of function definitions, where each function itself contains a list of Silage statements. The functions are sorted with the lowest level functions first, as shown in Figure B.1. This data structure is sufficient for the synthesis of a CDFG.



**FIGURE B.1 : Silage Function Definition List**

## B.2 CDFG Generator

The CDFG generator makes a 1-pass traversal (with back patching) of the linked-list of function definitions, and constructs a hierarchical flowgraph description of the Silage program. In the following sections, we will describe the algorithm and the data structure organizations.

### B.2.1 The Algorithm

The CDFG generator is a collection of routines to generate a CDFG from the Silage Function Definition list. It assumes that the functions are ordered from the lowest level to the highest level, and that statements inside the function definitions are ordered according to data dependency. The essence of the program is given in the following pseudo-code:

```
ConstructGraph(FunctionList) {
```

```

For each Function in FunctionList do
    ConstructFunction(Function);
    Store Function in FunctionTable;
}

ConstructFunction(Function) {
    Allocate a new Graph to contain the following Edges and Nodes {
        For each Parameter in Function->InputParameters do
            Edge = CreateAndRegisterEdge(Parameter);
        For each Statement in Function do
            ConstructStatement(Statement);
    }
}

ConstructStatement(Statement) {
    if (Statement->Type == Single)
        ConstructSingleStatement(Statement->Single);
    if (Statement->Type == Loop)
        ConstructIterationStatement(Statement->Loop);
}

ConstructIterationStatement(Loop) {
    Node = CreateAndRegisterNode("Iteration");
    Allocate a new Graph to contain the following Edges and Nodes {
        For each Statement in Loop do
            ConstructStatement(Statement);
    }
    DetermineDependency(Graph);
    Node->Subgraph = Graph;
}

ConstructSingleStatement(Single) {
    RightEdge = ConstructExpression(Single->RHS);
    LeftEdge = ConstructExpression(Single->LHS);
    Node = CreateAndRegisterNode("Equal");
    Connect RightEdge to Node;
    Connect Node to LeftEdge;
}

ConstructExpression(Exp) {
    switch(Exp->Type) {

```

```

    case IDENTIFIER: return(CreateAndRegisterEdge(Exp));
    case PLUS: return(ConstructBinaryExp(Exp));
    ....
    case FUNC: return(ConstructFuncExp(Exp));
    case READ: return(ConstructReadExp(Exp));
    ...
  }
}

```

```

ConstructBinaryExp(Exp) {
  RightEdge = ConstructExpression(Single->RHS);
  LeftEdge = ConstructExpression(Single->LHS);
  Node = CreateAndRegisterNode(Exp->Type);
  OutputEdge = CreateAndRegisterEdge(NewName());
  Connect RightEdge, LeftEdge to Node;
  Connect Node to OutputEdge;
  return(OutputEdge);
}

```

```

CreateAndRegisterEdge(Name) {
  if (Name is registered in EdgeTable) return(EdgeTable[Name]);
  Edge = NewEdge(Name);
  Store Edge in EdgeTable[Name];
  Store Edge in Graph;
  return(Edge);
}

```

```

CreateAndRegisterNode(Name) {
  Node = NewNode(NewName());
  Node->Subgraph = LeafGraph[Name];
  Store Node in Graph;
  return(Node);
}

```

The key routine is `ConstructExpression()`, which for a given expression, creates a node for the operator as well as input and output edges for the operator's inputs and outputs. The resultant output edges are returned. The routine is implemented in a recursive way to process arbitrarily complex expressions. As the nodes and edges

are generated from the function definitions, the position of the Silage code (line number, character number, filename) is copied over for error reporting.

### Hierarchy Construction

Iterations and Function calls correspond to a change in hierarchy level in a CDFG. The mechanism for constructing an iteration was discussed above, while constructing a function call is as follows:

```
ConstructFuncExp(Func) {
    Node = NewNode(NewName());
    if (Func is not registered in FunctionTable) Error("Function not defined");
    Node->Subgraph = FunctionTable[Func];
    Build Node->InEdges, Node->OutEdges;
    return(Node->OutEdges);
}
```

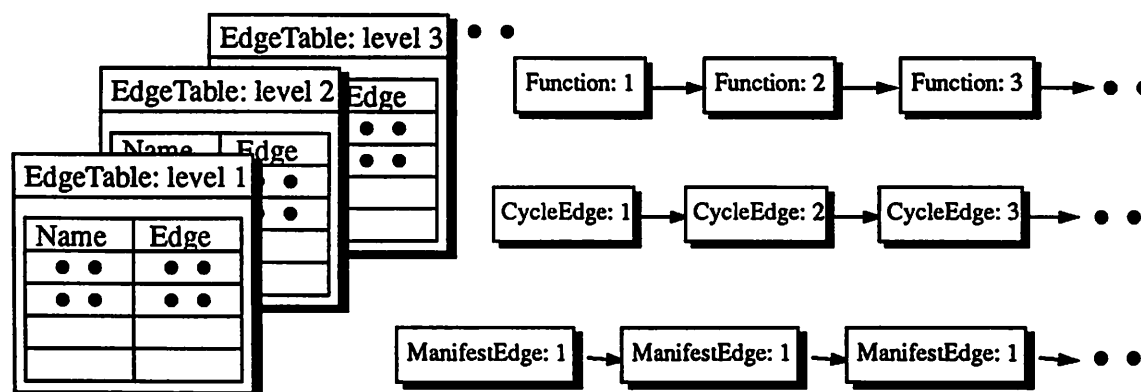
While both iteration and function constructs are expressed in the next level hierarchy by a hierarchical node, the two differ in the following points: First, the iteration subgraph is generated on the fly as the iteration statement is encountered. The program temporarily descends one level of hierarchy, builds the subgraph, and returns. The function call subgraph, on the other hand, has already been generated when the program encountered the function's definition on the FunctionList. Hence, the linking of the function call node to its subgraph reduces to a table lookup. Note that it is possible to have many function call nodes pointing to the same subgraph, while there is a one-to-one correspondence between an iteration node and its subgraph. Secondly, the function call subgraph shows the exact computation of the function call, while the iteration subgraph conveys only one iteration of the iteration node. Furthermore, attached to the iteration node is an attribute stating how the iterations are related to each other. Iterations are also classified into two types: *parallel* dependency, where the iterations are independent of each other, and *serial* dependency, where the iterations must be executed serially to ensure correct execution. The dependency in this case can be between two successive iterations or two iterations that are farther apart. The

iteration loop will always be executed in sequence when there is serial dependency, independent of the scope of the dependency. Finally, there is a formal interface between a call to a function and the function itself, specifically, the function arguments. These arguments serve as the inputs to the function, and all signals defined in the function must derive from them. On the contrary, there is no such interface between an iteration node and its subgraph. A signal defined inside an iteration, for instance, may use other signals which were defined previously outside of the iteration. These signals, then, must be located by the CDFG generator and imported into the iteration subgraph. For nested iterations, this can mean importing them across many levels of hierarchy. Lastly, the scope of the signals defined in a called function are disjoint from the scope of the signals defined in the calling function. Hence, signals with the same name can be defined. In the iteration case, of course, this would be a violation of the single-assignment rule, as the scope of the signals inside or outside of the iteration is still the same. In the next section, we discuss the data structures used by the CDFG generator which support this detection.

### B.2.2 The Data Structure

The CDFG generator uses a number of tables and lists to keep track of the nodes and edges that are being generated. They are used for error detection and correction, as well as for cycle generation. The tables and lists, as shown in Figure B.2, include the `EdgeTable`, the `FunctionList`, the `CycleEdgeList`, and the `ManifestEdgeList`.

The `EdgeTable` is an array of hash tables indexed by the level of nested iterations in the function definition. The `EdgeTable` is cleared at the beginning of each new function definition. There is a hash table for each graph or subgraph created, each storing all the edges that make up the graph. When an iteration is entered, the level is incremented, and a new hash table is used to store the edges in the new subgraph. When the `CreateAndRegisterEdge()` routine is invoked, we first check to see whether the edge has already existed in the `EdgeTable`. To perform this check, we hash into the hash



**FIGURE B.2 : Data Structures of CDFG Generator**

tables starting at the current level and working backward to level one. If the proposed edge exists at the current level, we return the edge; if it exists in a previous level, we instantiate a copy of it at the current level for the current graph, and if it is new, we create a new edge and install it in the current hash table. In processing the LHS of a statement, the edge should always be new; otherwise, we have a single assignment violation. Since the EdgeTable is cleared for each new function definition, it is possible to use the same name to define two signals in two different functions.

The FunctionList keeps track of all the functions which have been translated so far. The construction of a function call construct needs to traverse this list to link the function call node to its subgraph. The CycleEdgeList is used in the construction of delay cycles, and the ManifestEdgeList is used to process array signals. These two cases are discussed in section 4.1.

# Appendix **C**

## Code Generation Results

This appendix gives the Silage program for the Histogram example, and the corresponding C program generated by McDAS. The example is scheduled on 4 processors to allow readability without sacrificing generality. The C program is executable on the Sequent machine and includes code to gather running time statistics. The code for a real-time implementation would be more basic and compact. Due to limited space, only the Histogram example is shown. Other examples can be found in `~hyper/mcdas/Demo` or `~hoang/McDAS/Examples`.



## C.1 Histogram Silage Code

```
/* Silage Description for S2C
```

```
    CALCULATION OF PROBABILITY DENSITY FUNCTION (PDF)
```

```
    Principle available from Dongping [DON87] and Aubert [AUB85].
```

```
    Calculates normalized Amplitude Probability Function for intervals of 1s of EKG,  
i.e. 100 samples. */
```

```
#define W1 fix<16,0>
```

```
#define W2 fix<8,0>
```

```
#define N 128
```

```
#define M 32
```

```
#define LOGM 5
```

```
/*    N : number of samples involved  
    x : input array of EKG samples  
    immax : intermediate maximum, used for applicative reason  
    immin : intermediate minimum, used for applicative reason  
    max : definite maximum amplitude sample of x  
    min : definite minimum amplitude sample of x  
    range : width of a single amplitude subclass  
    limit : upper limit of subclass, limit[0] only used to designate  
    lower limit of subclass l  
    k : loop variable  
    subclass[k][i] : subclasses, k-index designates the particular subclass  
    i-index designates version (applicative) contains the number of members of  
the subclass  
    class_of_zero : indicates the class of EKG baseline, intermediate  
    cz : definite class of EKG baseline  
    sb : designates startclass of baselinerange  
    baseline : intermediate value array for calculation of baselinerange  
    baselinerange : definite number of samples in the 5 subclasses around the  
baselineclass cz in percent of total. */
```

```
func main (x : W1[N]) baselinerange: W2 =  
begin
```

```
/* first part of the description determines max. and min. value of  
the input array of EKG samples */
```

```
(k : 1 .. N-1) ::
```

```
begin
```

```
    immax##1 = x[0];
```

```
    immin##1 = x[0];
```

```
    immax = if (x[k] >= immax##1) -> x[k] || immax##1 fi;
```

```
    immin = if (immin##1 >= x[k]) -> x[k] || immin##1 fi;
```

```
end;
```

```
gmax = immax;
```

```
gmin = immin;
```

```
/* second part determines the limits of the 32 amplitude subclasses  
of the PDF histogram, subclasses are initialized */
```

```
range = W1((gmax-gmin)>>LOGM);
```

```
limit[0] = gmin;
```

```
(k : 1 .. M) ::
```

```

begin limit[k] = if (k == M) -> gmax || limit[k-1] + range fi;
end;

/* third part : filling in the subclasses */

(i : 0 .. M-1) ::
begin lower_limit = limit[i];upper_limit = limit[i+1];
(k : 0 .. N-1) ::
begin count##1 = W2 (0);count = if (x[k] >= lower_limit & x[k] <
upper_limit) -> count#1 + W2 (1)|| count#1 fi;
end; subclass[i] = count;
end;

/* fourth part : determination of class of baseline */

(k : 1 .. M-1) ::
begin
class_of_zero##1 = W2(0);
class_of_zero = if (limit[k] <= 0) & (limit[k+1] > 0)
-> W2 (k)
|| class_of_zero#1
fi;
end;
lb = if class_of_zero >= 2 -> W2(class_of_zero - 2)
|| W2(0)
fi;
ub = lb + W2 (4);
(k : 0 .. M-1) ::
begin
br##1 = W2(0);
br = if (W2 (k) >= lb & W2 (k) <= ub) -> W2 (br#1 + subclass[k])
|| br#1 fi;
end;
baselinerange = br;
end;

```

## C.2 Histogram C Code

```

/*
 * Author : Phu Hoang
 * Date : Sun Apr 26 20:30:48 1992
 */

#include <stdio.h>
#include <math.h>
#include <sys/types.h>
#include <sys/times.h>
#include <usclkc.h>
#include <parallel/microtask.h>
#include <parallel/parallel.h>

#ifdef HIGHLEVEL
#include "highlevel.h"
#endif
#ifdef BITTRUE
#include "bittrue.h"
#endif
#ifdef BITFAST
#include "bitfast.h"
#endif

#define at(d, i, m) ((i+d)%m)

FILE *dfd_dump, *dfd_stat[4];

FILE *OpenFile (FileName)
char *FileName;
{
    FILE *FD;
    FD = fopen (FileName, "r");
    if (FD == NULL) {
        fprintf(stderr, "Can't open file : %s\n", FileName);
        exit (-1);
    }
    return (FD);
}

FILE *CreateFile (FileName)
char *FileName;
{
    FILE *FD;
    FD = fopen (FileName, "w");
    if (FD == NULL) {
        fprintf(stderr, "Can't open file : %s\n", FileName);
        exit (-1);
    }
    return (FD);
}

/* Declaring Global Buffers */

shared Sig_Type x[2][128];
shared Sig_Type limit[2][33];

```

```

shared Sig_Type subclass[2][32];

shared slock_t subclass_31;
shared slock_t subclass_21;

/* Initialize Global Buffers */

void InitPipeLines()
{
    Sig_Type c0;
    int i_0, i_1;

    Float2Fix (0.0, c0, 16, 0);
    for(i_0=0; i_0 < 2; i_0++)
        for(i_1=0; i_1 < 128; i_1++)
            FixAssign (c0, x[i_0][i_1]);
    for(i_0=0; i_0 < 2; i_0++)
        for(i_1=0; i_1 < 33; i_1++)
            FixAssign (c0, limit[i_0][i_1]);
    for(i_0=0; i_0 < 2; i_0++)
        for(i_1=0; i_1 < 32; i_1++)
            FixAssign (c0, subclass[i_0][i_1]);
}

/* Initialize Semaphores */

void InitSemaphores()
{
    s_init_lock (&subclass_31);
    s_lock (&subclass_31);
    s_init_lock (&subclass_21);
    s_lock (&subclass_21);
}

main ()
{
    int i;
    char str[20];
    void InitPipeLines(), InitSemaphores(), m_fork(), m_kill_procs(), histogram();

    /* Creating Display files */
    dfd_dump = CreateFile("histogram#0.dmp");
    for (i=0; i<4; i++) {
        sprintf(str, "histogram#0.stat%d", i);
        dfd_stat[i] = CreateFile(str);
    }
    /* Initialize PipeLines */
    InitPipeLines();
    /* Initialize Semaphores */
    InitSemaphores();
    /* Initialize Statistic clock*/
    usclk_init();
    /* Reserve Processors */
    m_set_procs(4);
    /* Fork Processes */
    m_fork(histogram);
    /* Kill Processes */
    m_kill_procs();
}

```

```

/* Closing Display files */
fclose(dfd_dump);
for (i=0; i<4; i++)
    fclose(dfd_stat[i]);
}

void histogram ()
{
    int nprocs, proc;
    void subgraph_0(), subgraph_1(), subgraph_2(), subgraph_3();

    nprocs = m_get_numprocs();
    proc = m_get_myid();
    printf("Proc %d active...\n", proc);
    fflush(stdout);
    switch(proc) {
        case 0 : subgraph_0(); break;
        case 1 : subgraph_1(); break;
        case 2 : subgraph_2(); break;
        case 3 : subgraph_3(); break;
        default : break;
    }
}
/*****
* Sub-Program 0
*****/
/* Declare Buffer indices...*/
int x0_C = 0;
int limit0_C = 0;
int subclass0_C = 0;

/* Delay structure definition goes here...*/

/* Declaring Input and Output File Descriptors */
FILE *fd_x;

/* Declaring FixedPoint Constants as globals */
Sig_Type e17, e23;

/* Initialize the FixedPoint globals */
Initsubgraph_0FixedLeafs ()
{
    Float2Fix (0.000000, e17, 8, 0);
    Float2Fix (1.000000, e23, 8, 0);
}

/* Initializing Delay structure */

ReadArray_x (pIn)
Sig_Type pIn[128];
{
    int i_0;
    for (i_0=0; i_0<128; i_0++)
        if (Read_x (&pIn[i_0]) < 0)
            return(-1);
    return(0);
}

```

```

Read_x (pIn)
Sig_Type *pIn;
{
    float d;
    if (fscanf (fd_x, "%f ", &d) != 1) {
        fprintf(stderr, "Reach end of file : filex\n");
        return (-1);
    } else
        Float2Fix (d, pIn[0], 16, 0);
    return(0);
}

void subgraph_0 ()
{
    Sig_Type fdum;
    int interval;
    int CycleCount, MaxCycles;
    usclk_t t32;
    MaxCycles = 10;
    Initsubgraph_0FixedLeafs ();

    /* Opening Input files */
    fd_x = OpenFile ("filex");
    interval = (int)(MaxCycles/5);

    /* Simulation for # cycles = MaxCycles */
    for (CycleCount = 1; CycleCount < MaxCycles+1; CycleCount++) {
        m_sync();
        t32 = getusclk();
        /* Calling main simulation routine */
        Sim_subgraph_0 (x, limit, subclass);
        if (--x0_C < 0) x0_C = 1;
        if (--limit0_C < 0) limit0_C = 1;
        if (--subclass0_C < 0) subclass0_C = 1;
        t32 = getusclk() - t32;
        if (CycleCount%interval == 0) {
            fprintf (dfd_stat[0], "Proc0(%d) = %lu usecs.\n", CycleCount, t32);
            fflush (dfd_stat[0]);
        }
    }

    /* Closing Input files */
    fclose (fd_x);
    exit(0);
}

Sim_subgraph_0 (x, limit, subclass)
Sig_Type x[2][128];
Sig_Type limit[2][33];
Sig_Type subclass[2][32];
{
    Sig_Type n2_immax[2];
    int n2_immax_C;
    Sig_Type n3_immin[2];
    int n3_immin_C;
    Sig_Type n4_count[2];
    int n4_count_C;
}

```

```

Sig_Type c0;

/* Declaring variables to hold temporary edges */
int k, i;
Sig_Type fdum, gmin, gmax, e4, e5, range, immax, immin;

/* statements of function body */
if (ReadArray_x(x[at(0, x0_C, 2)]) < 0)
    exit(0);
n2_immax_C = 0;
n3_immin_C = 0;
FixAssign (x[at(0, x0_C, 2)][0], n2_immax[at (1, n2_immax_C, 2)]);
FixAssign (x[at(0, x0_C, 2)][0], n3_immin[at (1, n3_immin_C, 2)]);

for (k = (1); k <= (127); k++) {
/* Declaring variables to hold temporary edges */
int idum, e2, e3;
Sig_Type fdum;
e3); FixGTE (n3_immin[at (1, n3_immin_C, 2)], 16, 0, x[at(0, x0_C, 2)][k], 16, 0,
if (e3) {
    FixAssign (x[at(0, x0_C, 2)][k], n3_immin[n3_immin_C]);
}
else {
    FixAssign (n3_immin[at (1, n3_immin_C, 2)], n3_immin[n3_immin_C]);
}
e2); FixGTE (x[at(0, x0_C, 2)][k], 16, 0, n2_immax[at (1, n2_immax_C, 2)], 16, 0,
if (e2) {
    FixAssign (x[at(0, x0_C, 2)][k], n2_immax[n2_immax_C]);
}
else {
    FixAssign (n2_immax[at (1, n2_immax_C, 2)], n2_immax[n2_immax_C]);
}
if (--n2_immax_C < 0) n2_immax_C = 1;
if (--n3_immin_C < 0) n3_immin_C = 1;
}

n2_immax_C = (n2_immax_C + 1) % 2;
n3_immin_C = (n3_immin_C + 1) % 2;
FixAssign (n3_immin[n3_immin_C], immin);
FixAssign (n2_immax[n2_immax_C], immax);
FixAssign (immax, gmax);
FixAssign (immin, gmin);
FixMinus (gmax, 16, 0, gmin, 16, 0, e4, 16, 0);
FixSR (e4, e5, 16, 0, 5);
FixCast (e5, 16, 0, range, 16, 0);
FixAssign (gmin, limit[at(0, limit0_C, 2)][0]);

for (k = (1); k <= (32); k++) {
/* Declaring variables to hold temporary edges */
int idum, e6, e7, e8, e9;
Sig_Type fdum, e11, e12;
e9 = k - 1;
FixPlus (limit[at(0, limit0_C, 2)][e9], 16, 0, range, 16, 0, e11, 16, 0);
e7 = k == 32;
if (e7) {
    FixAssign (gmax, e12);
}
}

```

```

    }
    else {
        FixAssign (e11, e12);
    }
}
FixAssign (e12, limit[at(0, limit0_C, 2)][k]);
}

for (i = (0); i <= (4); i++) {
/* Declaring variables to hold temporary edges */
int k, e14, e15;
Sig_Type fdum, lower_limit, upper_limit, count;
e15 = i + 1;
FixAssign (limit[at(0, limit0_C, 2)][e15], upper_limit);
FixAssign (limit[at(0, limit0_C, 2)][i], lower_limit);
n4_count_C = 0;
FixAssign (e17, n4_count[at (1, n4_count_C, 2)]);

for (k = (0); k <= (127); k++) {
/* Declaring variables to hold temporary edges */
int idum, e20, e21, e22;
Sig_Type fdum, e25;
FixPlus (n4_count[at (1, n4_count_C, 2)], 8, 0, e23, 8, 0, e25, 8, 0);
FixLT (x[at(0, x0_C, 2)][k], 16, 0, upper_limit, 16, 0, e21);
FixGTE (x[at(0, x0_C, 2)][k], 16, 0, lower_limit, 16, 0, e20);
e22 = e20 && e21;
if (e22) {
    FixAssign (e25, n4_count[n4_count_C]);
}
else {
    FixAssign (n4_count[at (1, n4_count_C, 2)], n4_count[n4_count_C]);
}
if (--n4_count_C < 0) n4_count_C = 1;
}

n4_count_C = (n4_count_C + 1) % 2;
FixAssign (n4_count[n4_count_C], count);
FixAssign (count, subclass[at(0, subclass0_C, 2)][i]);
if (--n4_count_C < 0) n4_count_C = 1;
}

n4_count_C = (n4_count_C + 1) % 2;
}

/*****
* Sub-Program 1
*****/
/* Declare Buffer indices ...*/
int limit1_C = 0;
int x1_C = 0;
int subclass1_C = 0;

/* Delay structure definition goes here...*/

/* Declaring FixedPoint Constants as globals */
Sig_Type e40, e38, e43, e45, e17, e23, e26, e29, e34, e47;

/* Initialize the FixedPoint globals */
Initsubgraph_1FixedLeafs ()

```



```

{
  Float2Fix (2.000000, e40, 8, 0);
  Float2Fix (2.000000, e38, 8, 0);
  Float2Fix (0.000000, e43, 8, 0);
  Float2Fix (4.000000, e45, 8, 0);
  Float2Fix (0.000000, e17, 8, 0);
  Float2Fix (1.000000, e23, 8, 0);
  Float2Fix (0.000000, e26, 8, 0);
  Float2Fix (0.000000, e29, 16, 0);
  Float2Fix (0.000000, e34, 16, 0);
  Float2Fix (0.000000, e47, 8, 0);
}

/* Initializing Delay structure */

void subgraph_1 ()
{
  Sig_Type fdum;
  int interval;
  int CycleCount, MaxCycles;
  usclk_t t32;
  MaxCycles = 10;
  Initsubgraph_1FixedLeafs ();
  interval = (int)(MaxCycles/5);

/* Simulation for # cycles = MaxCycles */
  for (CycleCount = 1; CycleCount < MaxCycles+1; CycleCount++) {
    m_sync();
    t32 = getusclk();
    /* Calling main simulation routine */
    Sim_subgraph_1 (limit, x, subclass);
    if (--limit1_C < 0) limit1_C = 1;
    if (--x1_C < 0) x1_C = 1;
    if (--subclass1_C < 0) subclass1_C = 1;
    t32 = getusclk() - t32;
    if (CycleCount%interval == 0) {
      fprintf (dfd_stat[1], "Proc1(%d) = %lu usecs.\n", CycleCount, t32);
      fflush (dfd_stat[1]);
    }
  }
  exit(0);
}

Sim_subgraph_1 (limit, x, subclass)
Sig_Type limit[2][33];
Sig_Type x[2][128];
Sig_Type subclass[2][32];
{
  Sig_Type n5_count[2];
  int n5_count_C;
  Sig_Type n6_class_of_zero[2];
  int n6_class_of_zero_C;
  Sig_Type n7_br[2];
  int n7_br_C;
  Sig_Type c0;

/* Declaring variables to hold temporary edges */
  int i, k, e39;

```

```

Sig_Type fdum, e41, e42, lb, ub, baselinerange, class_of_zero, br;

/* statements of function body */
for (i = (5); i <= (14); i++) {
/* Declaring variables to hold temporary edges */
int k, e14, e15;
Sig_Type fdum, lower_limit, upper_limit, count;

    e15 = i + 1;
    FixAssign (limit[at(1, limit1_C, 2)][e15], upper_limit);
    FixAssign (limit[at(1, limit1_C, 2)][i], lower_limit);
    n5_count_C = 0;
    FixAssign (e17, n5_count[at (1, n5_count_C, 2)]);

    for (k = (0); k <= (127); k++) {
/* Declaring variables to hold temporary edges */
int idum, e20, e21, e22;
Sig_Type fdum, e25;
        FixLT (x[at(1, x1_C, 2)][k], 16, 0, upper_limit, 16, 0, e21);
        FixGTE (x[at(1, x1_C, 2)][k], 16, 0, lower_limit, 16, 0, e20);
        e22 = e20 && e21;
        FixPlus (n5_count[at (1, n5_count_C, 2)], 8, 0, e23, 8, 0, e25, 8, 0);
        if (e22) {
            FixAssign (e25, n5_count[n5_count_C]);
        }
        else {
            FixAssign (n5_count[at (1, n5_count_C, 2)], n5_count[n5_count_C]);
        }
        if (--n5_count_C<0) n5_count_C = 1;
    }
    n5_count_C = (n5_count_C+1)%2;
    FixAssign (n5_count[n5_count_C], count);
    FixAssign (count, subclass[at(1, subclass1_C, 2)][i]);
    if (--n5_count_C<0) n5_count_C = 1;
}
n5_count_C = (n5_count_C+1)%2;
n6_class_of_zero_C = 0;
FixAssign (e26, n6_class_of_zero[at (1, n6_class_of_zero_C, 2)]);

for (k = (1); k <= (31); k++) {
/* Declaring variables to hold temporary edges */
int idum, e30, e31, e32, e35, e36;
Sig_Type fdum, e37;
    e37 = k;
    e32 = k + 1;
    FixGT (limit[at(1, limit1_C, 2)][e32], 16, 0, e34, 16, 0, e35);
    FixLTE (limit[at(1, limit1_C, 2)][k], 16, 0, e29, 16, 0, e30);
    e36 = e30 && e35;
    if (e36) {
        FixAssign (e37, n6_class_of_zero[n6_class_of_zero_C]);
    }
    else {
        FixAssign (n6_class_of_zero[at (1, n6_class_of_zero_C, 2)],
n6_class_of_zero[n6_class_of_zero_C]);
    }
    if (--n6_class_of_zero_C<0) n6_class_of_zero_C = 1;
}

```

```

n6_class_of_zero_C = (n6_class_of_zero_C+1)%2;
FixAssign (n6_class_of_zero[n6_class_of_zero_C], class_of_zero);
FixMinus (class_of_zero, 8, 0, e40, 8, 0, e41, 8, 0);
FixCast (e41, 8, 0, e42, 8, 0);
FixGTE (class_of_zero, 8, 0, e38, 8, 0, e39);
if (e39) {
    FixAssign (e42, lb);
}
else {
    FixAssign (e43, lb);
}
FixPlus (lb, 8, 0, e45, 8, 0, ub, 8, 0);
s_lock(&subclass_31);
s_lock(&subclass_21);
n7_br_C = 0;
FixAssign (e47, n7_br[at (1, n7_br_C, 2)]);

for (k = (0); k <= (31); k++) {
/* Declaring variables to hold temporary edges */
int idum, e50, e52, e53;
Sig_Type fdum, e49, e51, e55, e56;
    FixPlus (n7_br[at (1, n7_br_C, 2)], 8, 0, subclass[at(1, subclass1_C, 2)][k], 8, 0,
e55, 8, 0);
    FixCast (e55, 8, 0, e56, 8, 0);
    e51 = k;
    FixLTE (e51, 8, 0, ub, 8, 0, e52);
    e49 = k;
    FixGTE (e49, 8, 0, lb, 8, 0, e50);
    e53 = e50 && e52;
    if (e53) {
        FixAssign (e56, n7_br[n7_br_C]);
    }
    else {
        FixAssign (n7_br[at (1, n7_br_C, 2)], n7_br[n7_br_C]);
    }
    if (--n7_br_C<0) n7_br_C = 1;
}

n7_br_C = (n7_br_C+1)%2;
FixAssign (n7_br[n7_br_C], br);
FixAssign (br, baselinerange);
FixDisplay (dfd_dump, "subgraph_1 baselinerange", baselinerange, 8, 0);
fflush (dfd_dump);
}
/*****
* Sub-Program 2
*****/
/* Declare Buffer indices...*/
int limit2_C = 0;
int x2_C = 0;
int subclass2_C = 0;

/* Delay structure definition goes here...*/

/* Declaring FixedPoint Constants as globals */
Sig_Type e17, e23;

/* Initialize the FixedPoint globals */

```

```

Initsubgraph_2FixedLeafs ()
{
    Float2Fix (0.000000, e17, 8, 0);
    Float2Fix (1.000000, e23, 8, 0);
}

/* Initializing Delay structure */

void subgraph_2 ()
{
    Sig_Type fdum;
    int interval;
    int CycleCount, MaxCycles;
    usclk_t t32;
    MaxCycles = 10;
    Initsubgraph_2FixedLeafs ();
    interval = (int)(MaxCycles/5);

/* Simulation for # cycles = MaxCycles */
    for (CycleCount = 1; CycleCount < MaxCycles+1; CycleCount++) {
        m_sync();
        t32 = getusclk();
        /* Calling main simulation routine */
        Sim_subgraph_2 (limit, x, subclass);
        if (--limit2_C < 0) limit2_C = 1;
        if (--x2_C < 0) x2_C = 1;
        if (--subclass2_C < 0) subclass2_C = 1;
        t32 = getusclk() - t32;
        if (CycleCount%interval == 0) {
            fprintf (dfd_stat[2], "Proc2(%d) = %lu usecs.\n", CycleCount, t32);
            fflush (dfd_stat[2]);
        }
    }
    exit(0);
}

Sim_subgraph_2 (limit, x, subclass)
Sig_Type limit[2][33];
Sig_Type x[2][128];
Sig_Type subclass[2][32];
{
    Sig_Type n8_count[2];
    int n8_count_C;
    Sig_Type c0;

    /* Declaring variables to hold temporary edges */
    int i;
    Sig_Type fdum;

    /* statements of function body */
    for (i = (15); i <= (24); i++) {
        /* Declaring variables to hold temporary edges */
        int k, e14, e15;
        Sig_Type fdum, lower_limit, upper_limit, count;
        e15 = i + 1;
        FixAssign (limit[at(1, limit2_C, 2)][e15], upper_limit);
        FixAssign (limit[at(1, limit2_C, 2)][i], lower_limit);
        n8_count_C = 0;
    }
}

```

```

FixAssign (e17, n8_count[at (1, n8_count_C, 2)]);

for (k = (0); k <= (127); k++) {
/* Declaring variables to hold temporary edges */
int idum, e20, e21, e22;
Sig_Type fdum, e25;
  FixPlus (n8_count[at (1, n8_count_C, 2)], 8, 0, e23, 8, 0, e25, 8, 0);
  FixLT (x[at(1, x2_C, 2)][k], 16, 0, upper_limit, 16, 0, e21);
  FixGTE (x[at(1, x2_C, 2)][k], 16, 0, lower_limit, 16, 0, e20);
  e22 = e20 && e21;
  if (e22) {
    FixAssign (e25, n8_count[n8_count_C]);
  }
  else {
    FixAssign (n8_count[at (1, n8_count_C, 2)], n8_count[n8_count_C]);
  }
  if (--n8_count_C<0) n8_count_C = 1;
}

n8_count_C = (n8_count_C+1)%2;
FixAssign (n8_count[n8_count_C], count);
FixAssign (count, subclass[at(1, subclass2_C, 2)][i]);
if (--n8_count_C<0) n8_count_C = 1;
}

n8_count_C = (n8_count_C+1)%2;
s_unlock(&subclass_21);
}
/*****
* Sub-Program 3
*****/
/* Declare Buffer indices...*/
int limit3_C = 0;
int x3_C = 0;
int subclass3_C = 0;

/* Delay structure definition goes here...*/

/* Declaring FixedPoint Constants as globals */
Sig_Type e17, e23;

/* Initialize the FixedPoint globals */
Initsubgraph_3FixedLeafs ()
{
  Float2Fix (0.000000, e17, 8, 0);
  Float2Fix (1.000000, e23, 8, 0);
}

/* Initializing Delay structure */

void subgraph_3 ()
{
  Sig_Type fdum;
  int interval;
  int CycleCount, MaxCycles;
  usclk_t t32;
  MaxCycles = 10;
  Initsubgraph_3FixedLeafs ();
}

```

```

interval = (int)(MaxCycles/5);

/* Simulation for # cycles = MaxCycles */
for (CycleCount = 1; CycleCount < MaxCycles+1; CycleCount++) {
    m_sync();
    t32 = getusclk();
    /* Calling main simulation routine */
    Sim_subgraph_3 (limit, x, subclass);
    if (--limit3_C < 0) limit3_C = 1;
    if (--x3_C < 0) x3_C = 1;
    if (--subclass3_C < 0) subclass3_C = 1;
    t32 = getusclk() - t32;
    if (CycleCount%interval == 0) {
        fprintf (dfd_stat[3], "Proc3(%d) = %lu usecs.\n", CycleCount, t32);
        fflush (dfd_stat[3]);
    }
}
exit(0);
}

Sim_subgraph_3 (limit, x, subclass)
Sig_Type limit[2][33];
Sig_Type x[2][128];
Sig_Type subclass[2][32];
{
    Sig_Type n9_count[2];
    int n9_count_C;
    Sig_Type c0;

    /* Declaring variables to hold temporary edges */
    int i;
    Sig_Type fdum;

    /* statements of function body */
    for (i = (25); i <= (31); i++) {
        /* Declaring variables to hold temporary edges */
        int k, e14, e15;
        Sig_Type fdum, lower_limit, upper_limit, count;
        e15 = i + 1;
        FixAssign (limit[at(1, limit3_C, 2)][e15], upper_limit);
        FixAssign (limit[at(1, limit3_C, 2)][i], lower_limit);
        n9_count_C = 0;
        FixAssign (e17, n9_count[at (1, n9_count_C, 2)]);

        for (k = (0); k <= (127); k++) {
            /* Declaring variables to hold temporary edges */
            int idum, e20, e21, e22;
            Sig_Type fdum, e25;
            FixLT (x[at(1, x3_C, 2)][k], 16, 0, upper_limit, 16, 0, e21);
            FixGTE (x[at(1, x3_C, 2)][k], 16, 0, lower_limit, 16, 0, e20);
            e22 = e20 && e21;
            FixPlus (n9_count[at (1, n9_count_C, 2)], 8, 0, e23, 8, 0, e25, 8, 0);
            if (e22) {
                FixAssign (e25, n9_count[n9_count_C]);
            }
            else {
                FixAssign (n9_count[at (1, n9_count_C, 2)], n9_count[n9_count_C]);
            }
        }
    }
}

```

```
    if (--n9_count_C<0) n9_count_C = 1;
}

n9_count_C = (n9_count_C+1)%2;
FixAssign (n9_count[n9_count_C], count);
FixAssign (count, subclass[at(1, subclass3_C, 2)][i]);
if (--n9_count_C<0) n9_count_C = 1;
}

n9_count_C = (n9_count_C+1)%2;
s_unlock(&subclass_31);
}
```

# Appendix **D**

## **McDAS User's Manual**

This appendix contains information on how to use the tools in McDAS. Specifically, manual pages are given for the McDAS compilation manager, the Silage to CDFG translator, the scheduler, and the code generator.



## D.1 McDAS Compilation Manager

### NAME

Mcdas -- McDAS compilation manager

### SYNOPSIS

Mcdas [-aV] [FlowGraph]

### DESCRIPTION

Mcdas is a graphical, X-window based user interface to the McDAS system. It manages the overall scheduling and compilation process, keeps track of database versioning, shows the different options available to the user at any point in time and displays the scheduling and code generation results using textual and graphical feedback.

The Mcdas window consists of different fields, being a data entry area, a scrollbar area, a message display window, an information window and a menu area. The functionality of those different window areas will now be discussed on a one by one basis.

- **Data Entry Window** : Allows for the setting of a number of compilation parameters, such as the design name, the machine, and the architecture topology. A fourth field, called Version, displays the current version of the flowgraph database. This database is created by parsing the silage file. Each scheduling operation creates a new version of the database. The designer can always go back to previous points in the design process by overwriting the version number.
- **Scroll Bar Window** : The Processor Scroll Bar is used to set the number of available processors in the target architecture. The Model Scroll Bar dictates how far to model the inter-processor communication. The scheduler uses this measure to trade-off computational time versus solution quality.
- **Message Window** : Displays tool status and error messages.
- **Information Window** : Textual feedback from tools (results, flowgraphs, etc) is displayed in this window. It must be mentioned that some tools will utilize pop-up windows to display results in a graphical way.
- **Menu Window** : Contains a number of command buttons, which enable the user to interact with McDAS, guide the synthesis process and display results. The commands can be divided in two classes : the generic commands and the compilation specific commands. The currently available commands will now be discussed in detail.

#### Generic Commands

- **Machine** : Select an underlying processor technology. One of three processors can be selected: Sequent or SMART or Ideal - Sequent is the default machine. Using the Sequent machine will result in an implementation for the Sequent machine. That is the computation time costs used by the scheduler will be those that were benchmarked from the Intel 386, the core processor of the Sequent machine. In addition, the C code generated will contain Sequent-specific routines. Similarly, using the SMART machine will use estimates from the AT&T DSP32C, the core processor of the SMART machine. Using the Ideal machine assumes there is no cost in I/O, function calls, and loop test and increment.
- **Topology** : Select an interconnect structure for the architecture. One of three configurations can currently be selected: Shared Bus or Linear Array or Configurable - Shared Bus is the default topology. Using the Shared Bus topology will specify that the core processors are connected by a single shared bus. Using the Linear Array topology assumes that the core processors are connected in a ring architecture. Finally, in the configurable topology, it is assumed that we have a single shared bus which can be configured by the scheduler.

This is the configurable bus which is available in the SMART architecture.

- **View Silage** : displays the original Silage description in the information window.
- **View Flowgraph** : the current version of the flowgraph database is shown in the information window (using the AFL - ascii flowgraph language- format).
- **View History** : displays a history of how each version of the flowgraph is generated, giving the Machine, the Topology, and the processor count.
- **View PartitionStat** : displays the scheduling results of the current version showing the number of available processors, the stagetime, the speedup, the processor assignment, the buffer memory usage, and the communications incurred.
- **View GanttChart** : displays the scheduling results in the form of a Gantt chart.
- **View PartitionLog** : shows a log of the entire scheduling of the flowgraph. Often used for debugging purposes.
- **View OutputResult** : shows the output of the application after execution on the target architecture.

#### Compilation Commands

- **FlowGen**: This is normally the first step in the compilation procedure. The SILAGE input description is read and parsed into a Control/Data Flow Graph (CDFG). The CDFG is automatically displayed on the Information window.
- **Schedule** : schedule the current version of the flowgraph database. The architecture is derived from the Machine, Topology, and Processors menus and scroll bar. The scheduling statistics is automatically displayed on the Information window.
- **CodeGen** : generate C code for the current scheduled version of the flowgraph database. The implementation is targeted for the architecture as described from the Machine and Processors fields. The resultant code is automatically displayed on the Information window. Currently, only the Sequent Machine is supported.
- **Execute** : compile and execute the C code for the current version of the flowgraph database. The resultant output is automatically displayed on the Information window.

#### OPTIONS

- a Use the ascii-database format. Default is the Oct database.
- V Set the current database version. Default is the 0th version.

#### AUTHORS

Phu Hoang  
 University of California, Berkeley  
 hoang@zabriskie.Berkeley.EDU

#### BUGS

Only the Sequent machine is currently supported for code generation and execution.

## D.2 Silage To Flowgraph Translator

### NAME

Sil2Flow -- Silage to OCT Flowgraph Translator

### SYNOPSIS

**Sil2Flow** [-a] [-d] [-f] [-H] [-M] [-m] *filename*

### DESCRIPTION

**Sil2Flow** generates a hierarchical flowgraph from a program written in Silage. The name of the Silage file should be *filename.sil*.

Silage is a signal-flow language developed especially for specifying Digital Signal Processing algorithms. Each signal in Silage is actually a stream of samples. A special operator @ is used to refer to the value of a signal some iteration earlier. Silage supports boolean, integer, fixed point, and floating point data types. Interpolation and decimation of sample rates are supported. Silage follows the *single-assignment* rule, which guarantees that no signal is ever defined twice. This property makes it translatable uniquely into a flowgraph representation.

The flowgraph is described as a composition of nodes, data edges, and control edges. Nodes represent computations, data edges represent the flow of data between computations, and control edges are used to force dependency constraints between nodes that don't communicate data. The graph is *hierarchical* in that a node may also represent an instance of a complete graph. The flowgraph is stored in the OCT database, to which all tools interact.

An ASCII format flowgraph description language (AFL), which has a 1 to 1 correspondence to the OCT policy, serves as an easy readable user-interface into the OCT database. **Sil2Flow** generates an OCT flowgraph as a default, but can generate an AFL flowgraph upon request.

The Silage To Flowgraph translator maps the Silage program to a flowgraph with essentially the same hierarchical structure. A function call in Silage is represented as a *func* node which has a pointer to a subgraph representing the function body. Similarly, an iteration in Silage is represented as an *Iter* node with a pointer to a subgraph representing the loop body.

**Sil2Flow** is currently used by the HYPER high level synthesis system, and the McDAS multiprocessor compiler system. A number of transformations are done after the basic Silage to Flowgraph translation. The -H option performs transformations for HYPER. The -M option performs transformations for McDAS.

### OPTIONS

- a      Generate a textfile of the flowgraph in the AFL format. The name of the file will be *filename.afl*. This flag also disables the generation of the OCT flowgraph.
- d      Debug mode for the frontend of Silage2Flow. Print a textfile to stdout the Silage program with data types of all signals.
- f      Force the creation of subgraphs for primitive nodes, not just hierarchical nodes.
- H      Performs a number of transformations to transform certain operations into more primitive operations for HYPER. Some transformations include expanding multiplication into adds and shifts, add operations to perform tests for exiting loops, etc.
- M      Performs a number of transformations for McDAS such as manifest expression evaluation, common subexpression, dead code elimination.
- m      **Sil2Flow** automatically translates a multirate application into an equivalent single-rate application by clustering operations with the same rate into one process. Each process is then invoked a number of times corresponding to its rate. The -m flag disables this automatic

translation to allow the user to see the most basic translation.

#### **AUTHORS**

Phu Hoang  
University of California, Berkeley  
hoang@zion.Berkeley.EDU

#### **BUGS**

Sil2Flow does not cover the complete scope of the SILAGE language yet. A precise definition of the scope of Sil2Flow is described in Sil2Flow (5).

## D.3 Scheduler for Sequent

### NAME

ParSequent -- McDAS scheduler for the Sequent multiprocessor

### SYNOPSIS

ParSequent [-aLftibec] [FlowGraph]

### DESCRIPTION

ParSequent is a version of the McDAS scheduler targeted at the Sequent machine. The customization is done by compiling the program with a header file appropriate for the Sequent. The remaining text describes the generic scheduler.

The McDAS scheduler takes a flowgraph in the OCT or AFL format, a multiprocessor architecture description, and generates a schedule for implementing the flowgraph on the target machine. The goal of the scheduler is to maximize the throughput of the resultant implementation given a target architecture. This is achieved by simultaneously performing pipelining, retiming, and parallel execution. Communication overhead is considered make the scheduling more accurate.

The flowgraph is described as a composition of nodes, data edges, and control edges. Nodes represent computations, data edges represent the flow of data between computations, and control edges are used to force dependency constraints between nodes that don't communicate data. The graph is *hierarchical* in that a node may also represent an instance of a complete graph.

The architecture description includes the characterization of the computation and communication costs of the architecture (implemented as a C header file), the number of available processors, and the processor interconnection. All three components can be selected independently in the Mcdas compilation manager.

The scheduler tries to minimize the time allocated to a pipeline stage to maximize the throughput. This is done in an iterative manner. At the end of the scheduling, the schedule is annotated back onto the flowgraph, assigning to every node a processor, a pipeline stage, and an execution order. The scheduling statistics is also displayed on the user showing the processor assignments and utilization, the final speedup, the memory usage, and the communication costs. The scheduler also generates a schedule file which can be displayed graphically in the form of a Gantt chart (see McDAS compilation manager).

### OPTIONS

- a Use the ascii-database format. Default is the Oct database.
- L Keeps a log of the scheduling results in all iterations. The information is stored in **Design.log**.
- f Force the creation of subgraphs for primitive nodes, not just hierarchical nodes. Used with the *-a* option.
- p[n] Give the number of processors available. The default number is 8.
- t[n] Give the amount of time available in a pipeline stage. This is used if a desired throughput rate is known, and the goal is to minimize the number of required processors. The available time is the inverse of the desired throughput rate. The default value is 0.
- i[n] Give the number of iterations to attempt scheduling. The default number is 50.
- b[n] Give the bus configuration of the architecture. A value of 0 gives a shared bus architecture. A 1 gives a ring (or linear array) architecture, and a 2 gives a configurable bus architecture.
- e[n] Give the effort to model the interprocessor communication. The range is from 1 to 3, with

3 being the highest effort. The higher the effort, the more detail the data transfer is modelled. At a low effort, many data transfers are grouped into 1 block. At a high effort, each data transfer is individually considered. This allows the user to trade-off the accuracy of the scheduler and its scheduling time.

-c Dumps a flowgraph with the computation costs annotated.

#### **AUTHORS**

Phu Hoang  
University of California, Berkeley  
hoang@zabriskie.Berkeley.EDU

#### **BUGS**

## D.4 Scheduler for SMART

### NAME

ParSmart -- McDAS scheduler for the SMART multiprocessor

### SYNOPSIS

ParSmart [-aLfptibec] [FlowGraph]

### DESCRIPTION

ParSmart is a version of the McDAS scheduler targeted at the SMART machine. The customization is done by compiling the program with a header file appropriate for SMART. The remaining text describes the generic scheduler.

The McDAS scheduler takes a flowgraph in the OCT or AFL format, a multiprocessor architecture description, and generates a schedule for implementing the flowgraph on the target machine. The goal of the scheduler is to maximize the throughput of the resultant implementation given a target architecture. This is achieved by simultaneously performing pipelining, retiming, and parallel execution. Communication overhead is considered make the scheduling more accurate.

The flowgraph is described as a composition of nodes, data edges, and control edges. Nodes represent computations, data edges represent the flow of data between computations, and control edges are used to force dependency constraints between nodes that don't communicate data. The graph is *hierarchical* in that a node may also represent an instance of a complete graph.

The architecture description includes the characterization of the computation and communication costs of the architecture (implemented as a C header file), the number of available processors, and the processor interconnection. All three components can be selected independently in the Mcdas compilation manager.

The scheduler tries to minimize the time allocated to a pipeline stage to maximize the throughput. This is done in an iterative manner. At the end of the scheduling, the schedule is annotated back onto the flowgraph, assigning to every node a processor, a pipeline stage, and an execution order. The scheduling statistics is also displayed on the user showing the processor assignments and utilization, the final speedup, the memory usage, and the communication costs. The scheduler also generates a schedule file which can be displayed graphically in the form of a Gantt chart (see McDAS compilation manager).

### OPTIONS

- a Use the ascii-database format. Default is the Oct database.
- L Keeps a log of the scheduling results in all iterations. The information is stored in **Design.log**.
- f Force the creation of subgraphs for primitive nodes, not just hierarchical nodes. Used with the *-a* option.
- p[n] Give the number of processors available. The default number is 8.
- t[n] Give the amount of time available in a pipeline stage. This is used if a desired throughput rate is known, and the goal is to minimize the number of required processors. The available time is the inverse of the desired throughput rate. The default value is 0.
- i[n] Give the number of iterations to attempt scheduling. The default number is 50.
- b[n] Give the bus configuration of the architecture. A value of 0 gives a shared bus architecture. A 1 gives a ring (or linear array) architecture, and a 2 gives a configurable bus architecture.
- e[n] Give the effort to model the interprocessor communication. The range is from 1 to 3, with

3 being the highest effort. The higher the effort, the more detail the data transfer is modelled. At a low effort, many data transfers are grouped into 1 block. At a high effort, each data transfer is individually considered. This allows the user to trade-off the accuracy of the scheduler and its scheduling time.

-c Dumps a flowgraph with the computation costs annotated.

#### **AUTHORS**

Phu Hoang  
University of California, Berkeley  
hoang@zabriskie.Berkeley.EDU

#### **BUGS**



## D.5 Scheduler for Ideal Multiprocessor

### NAME

ParIdeal -- McDAS scheduler for an ideal multiprocessor

### SYNOPSIS

ParIdeal [-aLfptibec] [FlowGraph]

### DESCRIPTION

**ParIdeal** is a version of the McDAS scheduler targeted at an ideal multiprocessor machine. The customization is done by compiling the program with a header file appropriate for the machine. The machine is ideal in the sense that it assumes zero overhead in function calls, loop test and increment, and I/O. The remaining text describes the generic scheduler.

The McDAS scheduler takes a flowgraph in the OCT or AFL format, a multiprocessor architecture description, and generates a schedule for implementing the flowgraph on the target machine. The goal of the scheduler is to maximize the throughput of the resultant implementation given a target architecture. This is achieved by simultaneously performing pipelining, retiming, and parallel execution. Communication overhead is considered make the scheduling more accurate.

The flowgraph is described as a composition of nodes, data edges, and control edges. Nodes represent computations, data edges represent the flow of data between computations, and control edges are used to force dependency constraints between nodes that don't communicate data. The graph is *hierarchical* in that a node may also represent an instance of a complete graph.

The architecture description includes the characterization of the computation and communication costs of the architecture (implemented as a C header file), the number of available processors, and the processor interconnection. All three components can be selected independently in the Mcdas compilation manager.

The scheduler tries to minimize the time allocated to a pipeline stage to maximize the throughput. This is done in an iterative manner. At the end of the scheduling, the schedule is annotated back onto the flowgraph, assigning to every node a processor, a pipeline stage, and an execution order. The scheduling statistics is also displayed on the user showing the processor assignments and utilization, the final speedup, the memory usage, and the communication costs. The scheduler also generates a schedule file which can be displayed graphically in the form of a Gantt chart (see McDAS compilation manager).

### OPTIONS

- a Use the ascii-database format. Default is the Oct database.
- L Keeps a log of the scheduling results in all iterations. The information is stored in **Design-log**.
- f Force the creation of subgraphs for primitive nodes, not just hierarchical nodes. Used with the -a option.
- p[n] Give the number of processors available. The default number is 8.
- t[n] Give the amount of time available in a pipeline stage. This is used if a desired throughput rate is known, and the goal is to minimize the number of required processors. The available time is the inverse of the desired throughput rate. The default value is 0.
- i[n] Give the number of iterations to attempt scheduling. The default number is 50.
- b[n] Give the bus configuration of the architecture. A value of 0 gives a shared bus architecture. A 1 gives a ring (or linear array) architecture, and a 2 gives a configurable bus architecture.

- e[n] Give the effort to model the interprocessor communication. The range is from 1 to 3, with 3 being the highest effort. The higher the effort, the more detail the data transfer is modelled. At a low effort, many data transfers are grouped into 1 block. At a high effort, each data transfer is individually considered. This allows the user to trade-off the accuracy of the scheduler and its scheduling time.
- c Dumps a flowgraph with the computation costs annotated.

**AUTHORS**

Phu Hoang  
University of California, Berkeley  
hoang@zabriskie.Berkeley.EDU

**BUGS**

## D.6 Code Generator for Sequent

### NAME

Flow2SeqC -- OCT Flowgraph To C for Sequent Machine

### SYNOPSIS

Flow2SeqC [-a] [-v] [-f] [-l] [-d] *input*

### DESCRIPTION

Flow2SeqC takes a scheduled flowgraph in the OCT or AFL format, and generates C code to implement the behavior of the flowgraph. The C code contains library routines which implements multiprocessing on the Sequent multiprocessor machine.

The flowgraph is described as a composition of nodes, data edges, and control edges. Nodes represent computations, data edges represent the flow of data between computations, and control edges are used to force dependency constraints between nodes that don't communicate data. The graph is *hierarchical* in that a node may also represent an instance of a complete graph.

A scheduled flowgraph is one which is annotated with a multiprocessor schedule. Specifically, each node in the top level of hierarchy of the flowgraph is assigned a processor, an execution order, and a pipeline stage. Flow2SeqC partitions the flowgraph into subgraphs based on the processor assignment, and the nodes in the subgraphs are then ordered according to the schedule. C code is generated for each subgraph and assigned to the corresponding processor. A process forking mechanism is used to initiate multiprocessing. From that point on, each processor enters an infinite loop, executing its code once for each data sample. The processors globally synchronize at the beginning of each sample.

Interprocessor communication is achieved through shared memory in the Sequent. A FIFO mechanism is used to support the communication between processors. The size of the FIFO is calculated from the pipeline stage assignment of the source and destination processors, guaranteeing that no data can be corrupted during execution. The FIFOs are automatically generated and managed by the C code.

A command file *input.com* should be present at compiled time to tell the compiler where to find the input signal samples, and how many iterations to simulate. Flow2C generates the C file *input.c* and a makefile to compile the *input.c* program to simulate in either fixed point or floating point arithmetic. This is done by entering: `make high` for floating point, or `make bit` for fixed point. These commands generate executables *inputH* and *inputB* that perform the actual simulation. The files storing the input samples should be present at simulation time.

The results of the simulation are stored in a file called *input.time*. This file uses the xgraph-format (cfr (OCT) xgraph). A number of statistics file are also generated which measure the running time of each processor. These can be used to validate the accuracy of the estimations done in the scheduler.

### OPTIONS

- a Reads the flowgraph in the AFL format.
- v Verbose mode used with -a option. Check and print to stdout the various steps in reading the flowgraph from the AFL file.
- f Force the creation of subgraphs for primitive nodes, not just hierarchical nodes. Used with the -a option.
- l Minimize the number of local variables used in the C program.
- d Prints the generated code to standard out instead of to *input.c*



## D.7 Silage Syntax

### NAME

Silage -- Silage syntax and Flowgraph structure

### DESCRIPTION

Silage is a signal-flow language developed especially for specifying Digital Signal Processing algorithms. Each signal in Silage is actually a stream of samples. A special operator '@' is used to refer to the value of a signal some iteration earlier. Silage supports boolean, integer, fixed point, and floating point data types. Silage follows the *single-assignment* rule, which guarantees that no signal is ever defined twice. This property makes it translatable uniquely into a flowgraph representation.

The flowgraph is described as a composition of nodes, data edges, and control edges. Nodes represent computations, data edges represent the flow of data between computations, and control edges are used to force dependency constraints between nodes that don't communicate data. The graph is *hierarchical* in that a node may also represent an instance of a complete graph. The flowgraph is stored in the OCT database, to which all HYPER tools interact.

The Silage To Flowgraph translator maps the Silage program to a flowgraph with essentially the same hierarchical structure. A function call in Silage is represented as a *func* node which has a pointer to a subgraph representing the function body. Similarly, an iteration in Silage is represented as an *Iter* node with a pointer to a subgraph representing the loop body.

An ASCII format flowgraph description language (AFL), which has a 1 to 1 correspondence to the OCT policy, serves as an easy readable user-interface into the OCT database. Sil2Flow generates an OCT flowgraph as a default, but can generate an AFL flowgraph upon request.

### SYNTAX SUMMARY

The Silage language used in Sil2Flow is based on the *Silage to C (S2C)* compiler written by Chris Scheers [1], extended to include indefinite iterations, loop delays, and improved specification of multirate signals. Only the added features available in Sil2Flow will be described here. The syntax for these constructs are defined, and some examples illustrated. The reader is encouraged to refer to [1] for the basic Silage description and [2] for the basic OCT flowgraph policy.

*Loop Delay:* In Silage, in order to accumulate or find a maximum of a sequence of signals, an iteration construct is used to run across the length of the sequence, as in:

```
sum[0] = 0;
(i: 1 .. N) ::
    sum[i] = sum[i-1] + value[i];
```

Because a statement like "sum = sum + value[i]" does not obey the single assignment rule, the operations above had to be expressed using an array, as "sum[i] = sum[i-1] + value[i]". This notation can be misleading, for an array is not what is wanted here. The problem can be solved by introducing another notation for loop local variables which allows the value of a local variable at a previous iteration to be accessed. We thereby introduce the *loop delay operator* :

```
sum # n
```

which means "the value of sum, n iterations ago". The index *n* must be manifest and smaller than the maximum value of the loop counter. Loop delays can be initialized in the same way timing delays are initialized, but using '##' instead of '@@'. The accumulation above can now be expressed elegantly as:

```
sum##1 = 0;
```

```
(i: 1 .. N) ::
    sum = sum#1 + value[i];
```

Only the final value of sum is available at the end of the loop.

*Infinite Iterations:* The iteration construct is used to repeat some computation a fixed number of times. However, in certain applications, it is desirable to repeat the computation until a certain condition is met, which depends on the data being computed. In [1], the author allowed for procedurally executable Silage to handle this case, using a *while-do* construct. In order to maintain the data flow semantic, Sil2Flow is replacing this with a *do-exit* construct, defined in BNF form as follows:

```
<indefinite iteration> ::= 'do' { <definition> ';' }+
                          <exit clause> ';'
                          'od' ';'

<exit clause> ::= <expr> '=' 'exit' <if_body>
                  'tixe'

<if_body> ::= <expr> '->' <expr>
             | <expr> '->' <expr> '||' <expr>
             | <expr> '->' <expr> '||' <if_body>
```

In a do-exit construct, we treat the computation as being in its own indefinite loop, which ends when the exit condition is met. The signals declared to be exited is then exported back to the main indefinite loop as output signals of the indefinite iteration. An example Silage program using a do-exit construct is :

```
func main (In : fix<12,5>) Out : fix<12,5> =
begin
    do
        i@@1 = 0;
        i = i@1 + 1;
        m = exit i > In -> i || -1;
    od;
end;
```

*Multirate functions:* Sil2Flow will support 6 multirate functions: Upsampling, Downsampling, interpolation, decimation, time-multiplexing, and time-demultiplexing. For release 1.0, only the first 4 are supported. The syntax in BNF form is as follows:

```
<resample_stmt> ::= <name> '=' <resample_func> '('
                  <name> ',' <scale> ',' <phase> ')'
```

```
<resample_func> ::= 'Upsample'
                  | 'Downsample'
                  | 'Interpolate'
                  | 'Decimate'
```

`<scale>::= <integer>`  
`<phase>::= <integer>`

*scale* and *phase* are integral values representing the factor to resample by and which sample to start with, respectively. The phase value is always with respect to the higher rate signal. As an example:

```
x = Downsample(y, 2, 1);
```

means that the signal *y* is to be resampled with a lower sample rate. The rate is decreased by 2, and the resampling starts with sample *y*[1] instead of *y*[0]. The resultant signal is called *x*. For more information on these operations, see[3].

#### REFERENCES

- [1] Scheers, C., "User Manual for the S2C Silage to C Compiler," IMEC Laboratory, Belgium, August, 1988.
- [2] Rabaey, J., and P. Hoang, "OCT Flowgraph Policy," U.C Berkeley Internal Document, September, 1989.
- [3] Jacobs, G., "Multirate Digital Signal Processing," European Development Center, November, 1990.

#### AUTHORS

Phu Hoang  
University of California, Berkeley  
hoang@zion.Berkeley.EDU