COMPILE-TIME SCHEDULING OF DATAFLOW PROGRAM GRAPHS WITH DYNAMIC CONSTRUCTS

by

Soonhoi Ha

ABSTRACT

Scheduling of dataflow graphs onto parallel processors consists of assigning actors to processors, ordering the execution of actors within each processor, and firing the actors at particular times. Many scheduling strategies do at least one of these operations at compile time to reduce run-time cost of scheduling activities. In this thesis, we classify four scheduling strategies, (1) fully dynamic, (2) static-assignment, (3) self-timed, and (4) fully static. These are ordered in decreasing run-time cost. Optimal or near-optimal compile-time decisions require deterministic, data-independent program behavior known to the compiler. Thus, moving from strategy number (1) towards (4) either sacrifices optimality, decreases generality by excluding certain program constructs, or both. This thesis proposes scheduling techniques valid for strategies (2), (3), and (4) for dataflow graphs with dynamic constructs such as if-then-else, for-loop, do-while-loop, and recursion; for such graphs, although it is impossible to deterministically optimize the schedule at compile time, reasonable decisions can be made. For many applications, good compile-time decisions remove the need for dynamic scheduling or load balancing. We assume a known statistical distribution for the dynamic behavior of the constructs, and show how a compile-time decision about assignment and/or ordering as well as timing can be made. The criterion we use is to minimize the expected total idle time due to the construct; in certain cases, this will also minimize the expected makespan of the schedule. We will also show how to determine the number of processors that should be assigned to the construct. The method is illustrated with several programming examples, yielding very promising results.

Edward A. Lee
Thesis Committee Chairman

ACKNOWLEDGEMENTS

Praise God, my Heavenly Father! He led me, specially to the completion of this thesis, leads, and will lead me forever.

Past five years of my graduate work may not be imagined without a chain of helps and contributions of others. To me, the completion of thesis means how much I was indebted to others. It was a great blessing to study under Prof. Edward A. Lee. His encouragement and kindness has kept me pursuing my research with great joy. His academic zeal has opened up my eye to see what a researcher should be. Words can not express my thanks to him. I also give special thanks to Prof. David Messerschmitt and Prof. Jan Rabaey for encouraging my research to fruition.

Several colleagues deserve special appreciation. Shuvra Battacharyya, my best friend from the first graduate year, has continually cheered me up. Joe buck, Alan Kamas, Tom Parks, Phil Lapsley, John Barry, Paul Haskell, Phu Hoang, and many others have not spared their valuable time and efforts to support my research.

I also thank brothers and sisters in Christ for their prayer and love. Sharing life with them always leads me to more fruitfulness. Finally, there are some without whom I would not be what I am: My parents, a brother, a sister, lovely babies Hee-Joo and Sung-Ho, and my beloved wife Insook. I devote my thesis to them.

Brethren, I do not count myself to have apprehended; but one thing
I do, forgetting those things which are behind and reaching
forward to those things which are ahead. ---- Phil. 3:13

TABLE OF CONTENTS

1

INTRODUCTION

And, you shall know the truth, and the truth shall make you free.

--- John 8:32

To follow the insatiable demands for computing power, such as weather prediction, video processing, and much more, all levels of computer design have been consistently improved. The improvement of device technology has been the major driving force for fast computation by reducing the clock period to as low as tens of nanoseconds. At the architecture level, exploiting various forms of concurrence in programs has been an important technique. At the uniprocessor level, pipelining is the basic technique used to realize temporal concurrence. Spatial concurrence (parallelism) is exploited by multiple functional units by issuing multiple instructions at the same time. Since multiple processors offer more parallel computing power, the use of cooperating multiple processors is promising for further speed improvement. Currently the range of multiprocessor architectures is quite diverse, from small collections of supercomputers to thousands of synchronous single-bit processors.

In spite of significant advances made in the hardware aspects of parallel computation, the actual progress is far below the expected level because of the software and language aspects: synchronization, resource management, and programmability. By Amdahl's law, the speedup is limited by the reciprocal of the fraction of computation which must be performed serially [Amd67]. Therefore, the software for parallel computation should minimize required serial execution, which comes from the inherent nature of programs or the unavailability of required resources. Also, the software must be easy to write and debug.

One approach is to use conventional languages coupled with parallelizing compilers. The advantage of this approach is that existing programs only have to be recompiled and the programmers need not be concerned about the underlying hardware. The compiler, however, must be very complicated; it must check against side effects permitted by the language and partition a sequential program for execution on a multiprocessor system. Except for very regular structures such as a for-loop of array processing, this approach has proven unsuccessful since a conventional sequential language itself hides a great deal of potential parallelism.

Another approach is to extend the conventional style of programming with special primitives for parallel programming: spawning, synchronization, and message passing. Examples are Ada, CSP [Hoa78], extended Fortran (e.g., HEP, Sequent), Multilisp, and Occam. For a specific application, the programmer constructs a parallel algorithm which will uncover the hidden parallelism in a sequential program, and expresses it explicitly. However, much potential concurrency still may never be uncovered because of the inherent sequential concepts of the language, and debugging becomes more difficult.

It is widely acknowledged that conventional imperative languages are designed for a single von Neumann computer [Bac78] where the concept of a present state is matched with sequential execution. The principal operations of these languages involve changing the state of the computation. When applied to a parallel processing system, there is a major problem with side effects in which a part of the code imposes on another part of the code unintentionally. The use of global variables and multiple assignments of the same variable are typical sources of side effects.

On the other hand, dataflow languages[1] are based on function applications to available arguments. As a result, they are free of side effects. Parallelism in dataflow programs can be detected without a global analysis of the programs since during execution any two computations not dependent on each other for data are automatically eligible for concurrent execution. Dataflow languages also seem to offer opportunities for writing more modular, reliable, and verifiable programs. Examples of dataflow languages include Val[Mcg82], Id [Nik89] and SILAGE[Hil89].

Dataflow programs can be described by a dataflow graph. The dataflow graph is a directed graph G(T,E) where each node T (also called an actor or task[2]) stands for either an individual program instruction or a group thereof, and the arcs E carry operands (called tokens or data) from one operator to another (figure 1.1). When data is available

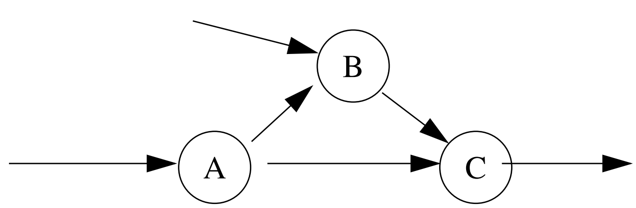


Figure 1.1 A three node dataflow graph with two inputs and one output. The nodes represent functions of arbitrary complexity and the arcs represent paths on which sequences of data flow.

1. The terms *functional* language, *applicative* language, *dataflow* language, and *reduction* language have been used somewhat interchangeably in the literature.
2. The terms *node*, *actor*, and *task* are used interchangeably throughout this dissertation.

on the input arc, actor A can be executed (or fired). Actor B waits for data from actor A before execution even though its other input arc may already have data. Therefore, arcs of a dataflow graph also represent precedence relations among actors. A dataflow graph is usually made hierarchical. In a hierarchical graph, an actor itself may represent another dataflow graph; it is called a *macro* actor.

Whether it serves as an intermediate representation which is translated from a textual language or as a programming language itself, the dataflow graph describes the structure of a program very effectively for multiprocessor systems. Throughout this dissertation, dataflow graphs are used to represent application programs regardless of how they are constructed.

1.1. GRANULARITY OF PARALLELISM

The size of an actor, or the number of primitive instructions in an actor, defines the *granularity* of a parallel program. An actor is the schedulable unit of computation which is executed sequentially. The more finely a program is divided into tasks (actors), the greater the opportunity for parallel execution. However, there is a commensurate increase in the frequency of inter-task communication and associated synchronization demands. For a given machine, there is a fundamental trade-off between the amount of parallelism that is profitable to expose and the overhead of synchronization.

Most textual dataflow languages aim to translate a program instruction into a single actor. A graph in which each node represents a single instruction is called a *fine-grain* dataflow graph. On the other hand, some hybrid graphical/textual languages have been developed to have actors contain textual description of functions [Bab84][Lee87b][Suh90]. Since conventional programming languages are used for textual description, a programmer need not learn a wholly new language. These are called *large-grain* (or *coarse-grain*) dataflow graphs. Large-grain dataflow graphs can also be made by grouping textual dataflow instructions to compose a large actor [Sar87][Ian88]. In this case, the aggregation of instructions must not introduce any cyclic dependency which can not be resolved. A fine-grain dataflow graph is obviously a special case of large-grain dataflow graph.

Which grain size is better? This can not be answered without considering the architecture of the hardware. The next section will reveal various directions of architecture design for the dataflow paradigm.

1.2. ARCHITECTURE FOR DATAFLOW OPERATION

In spite of all the attractive features of dataflow languages, they do not fit conventional von Neumann multiprocessors very well. Sarkar[Sar87] discusses the effect of the actor's total overhead in von Neumann multiprocessors on the actual speedup that can be attained. The overhead includes scheduling overhead and communication overhead for the actor's inputs and outputs. He concludes that the actor size should be large for optimal performance. Proponents of fine-grain dataflow graphs believe that his conclusion applies only to the von Neumann architecture.

There are two fundamental issues for scalable, general purpose parallel computers: latency and synchronization. Latency is the time which elapses between making a request and receiving the associated response, for example memory access time. Since von Neumann instruction sets are traditionally designed with instructions whose execution time is latency dependent, this latency may not be hidden, which results in extra overhead. Synchronization is the time-coordination of activities within a computation. Two popular mechanisms for synchronization are interrupts and semaphores, which respectively require context-switching or busy-waiting overhead. To overcome these difficulties, several architectures have been proposed. Among them, dataflow machines are fundamentally different from conventional von Neumann processors.

1.2.1 Dataflow Machines

The program instruction format for a dataflow machine is essentially an adjacency list representation of the program graph. Since the execution of an instruction is dependent on the arrival of operands, the management of token storage and instruction scheduling are intimately related. While the dataflow model assumes unbounded FIFO queues on the arcs, there are two alternatives for an actual implementation. A *static dataflow* machine provides a fixed amount of storage per arc. On the other hand, a *dynamic dataflow* machine allocates token storage dynamically out of a common pool and assumes that tokens carry tags to indicate their logical position on the arcs. Comprehensive surveys of early dataflow architectures are given by Vegdahl[Veg84], Arvind and Culler [Arv86] and Srini [Sri87].

Static Dataflow Machines

In a static machine, memory locations for tokens can be determined prior to execution. The availability of operands can be monitored by presence flags. Also, detecting enabled nodes can be done by associating a counter with each node. The first generation dataflow machines designed in the seventies fall into this category, including the MIT static dataflow architecture [Den75], the Lau system [Pla76], and TI's data-driven processor (DDP) [Cor79].

The basic instruction execution mechanism of a static dataflow machine is shown in figure 1.2. The activity store contains activity templates of the dataflow program. Each activity template has a unique address which is entered in the instruction queue when the instruction is ready for execution. The fetch unit fetches an executable instruction from the activity store addressed by the head entry of the instruction queue. The operation unit receives the instruction, performs execution and sends the result to the update unit of the same processor or a remote processor through the output unit. The update unit enters the received value into the operand field of specified activity templates. This unit also tests whether the destination instruction is fireable (executable) and, if so, enters the instruction address in the instruction queue.

This architecture illustrates an important characteristic of dataflow machines; communication latency between processors is hidden as long as enough enabled nodes are present in the instruction queue. It also shows that the synchronization primitives are installed in the essential part of the architecture, the update unit.

There is an interesting departure from this basic model. In the argument fetching dataflow architecture [Gao88], the data and signaling roles are separated and an instruction fetches its own argument from a data memory just like in the conventional von Neumann architecture.

Static dataflow machines, however, seem to have a couple of significant shortcomings. First, the amount of parallelism is reduced due to the restriction that an arc has

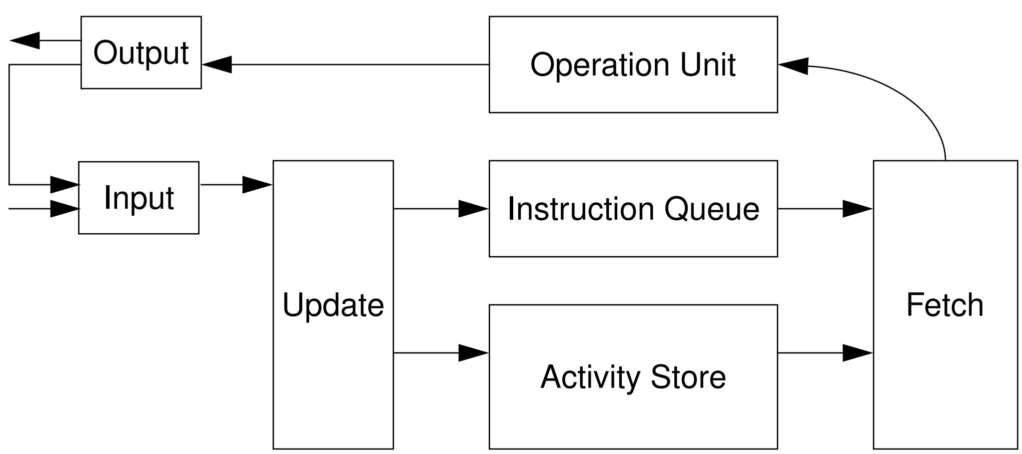


Figure 1.2 Basic instruction execution mechanism of static dataflow machine [Arv86]

a fixed amount of storage. Suppose an output arc of a node A has only one storage. Once the node is executed, the node can not be executed again until the token on the output arc is consumed by the destination node B. Second, a deadlock free dataflow graph may become deadlocked with bounded token storage. Suppose node A in the above example has another output connected to node C. And, node C should consume two tokens from node A before it generates two tokens to node B. After node A is executed once, no node can be executed any more since node A waits until node B consumes its output token and node B can not be executed until node A is executed once more. This is an example of deadlock due to bounded token storage implementation. These shortcomings motivated work on the more general dynamic dataflow approach discussed next.

Dynamic Dataflow Machines

In a dynamic dataflow machine, the number of concurrent invocations of an actor is unlimited. The basic requirement for this feature is that each invocation, called *activity*, of an actor should be uniquely named so that different invocations of the same node are distinguished. The unique name assigned to each activity is called a *tag,* so dynamic dataflow machines are also called *tagged-token dataflow machines*. The U-interpreter [Gos82] shows an example of how to tag activities dynamically and explicitly.

The basic execution model of dynamic dataflow machines is illustrated in figure 1.3. All tokens must carry the tags of their destination. When a token enters the wait-match stage, its tag is compared against the tags of the activities. With an associative store or dynamic hashing, tokens with the same tag are collected to make the associated activity executable. The associative store approach is used in the MIT tagged-token architecture (TTDA) [Arv88a], and the dynamic hashing approach is used in the Manchester architecture [Gur85]. Newly runnable activities from the incoming tokens enter into the instruction fetch unit. The instruction fetch unit delivers the operands and the op-code to the arithmetic unit. The arithmetic unit not only performs the operation but also manipulates the tags. The result is combined with successor instruction addresses specified in the instruction to yield <tag, data> pairs for the wait-match stage. If an instruction has multiple destinations, parallel computations are initiated by generating a token for each address.

When a program needs to deal with large data structures, the pure dataflow semantics is very inefficient since multiple copies of the data structure are required. Structure memory is a way of introducing a limited notion of state into dataflow graphs, without compromising parallelism or determinacy. For example, Arvind's I-structure supports split-phase memory references via a consumer-producer synchronization mechanism.

Performance simulation of dataflow programs on the MIT TTDA architecture is described in [Arv88b]. Since the machine was not built, Arvind et. al. report the number of machine instructions compiled for a uniprocessor. They reported that even before optimization, the number of instructions is comparable to that of von Neumann processors. However, the fundamental problem of dynamic dataflow machines lies in the implemen-

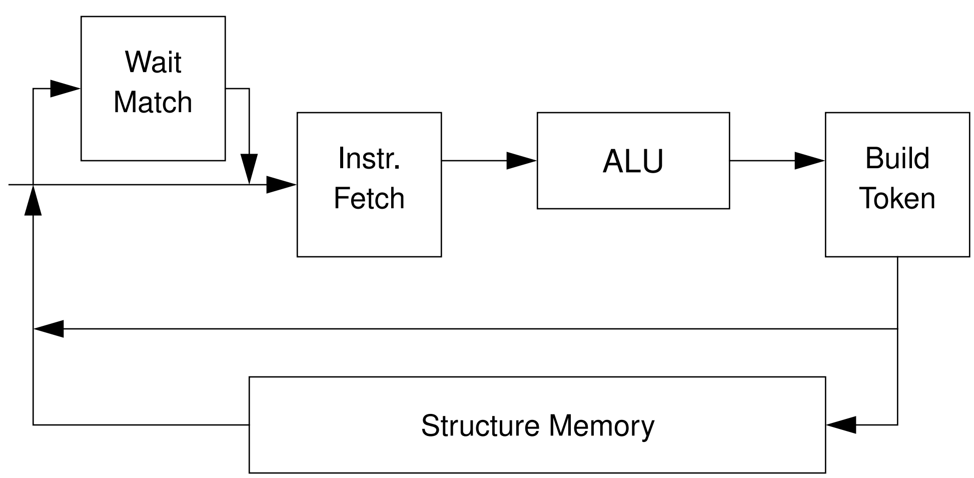


Figure 1.3 Basic instruction execution mechanism of dynamic dataflow machines [Arv88b].

tation of the wait-match unit. Neither associative memories nor hashing schemes provide sufficient bandwidth comparable with the ALU. Recently, Papadopoulos at MIT [Pap88] conceived the idea of *explicit token store*, in which a tag contains the encoded address of a unique global location. This is just like the von Neumann storage model except that each location is supplemented with a small number of presence bits.

In principle, dynamic dataflow machines can execute fine-grain dataflow graphs without any noticeable latency and synchronization overhead. Nonetheless, they are still far from commercial viability as general purpose multiprocessors. One problem to be solved is to design a resource management policy that can govern the tremendous amount of parallelism in case the hardware can not handle it effectively.

1.2.2 Multi-threaded Architecture

The multi-threaded architecture supports the concurrent execution of multiple *threads of control* in a processor to tolerate the memory latency and task switching overhead of von Neumann processors. The Denelcor HEP [Smi85] was the first commercially available multi-threaded computer. In the HEP, an active thread is put to sleep when it encounters a long latency operation by a very fast context switching mechanism to another thread. Even though the HEP mitigates the latency overhead by interleaving multiple threads, it still suffers the basic problem of latency sensitive operations.

Many researchers have conceived that large-grain dataflow graphs fit the multi-threaded architecture by viewing each actor as a thread. By the functional nature of actors, there is no data sharing among threads - a thread communicates with other threads only via the data produced at its completion. Also, the threads are not interruptible. Merging the dataflow paradigm into the multi-threaded architecture leads to dataflow / von Neumann hybrid architectures [Ian88][Hum91]. In both references, they start with a fine-grain dataflow graph and partition it to make a large-grain dataflow graph. By a split

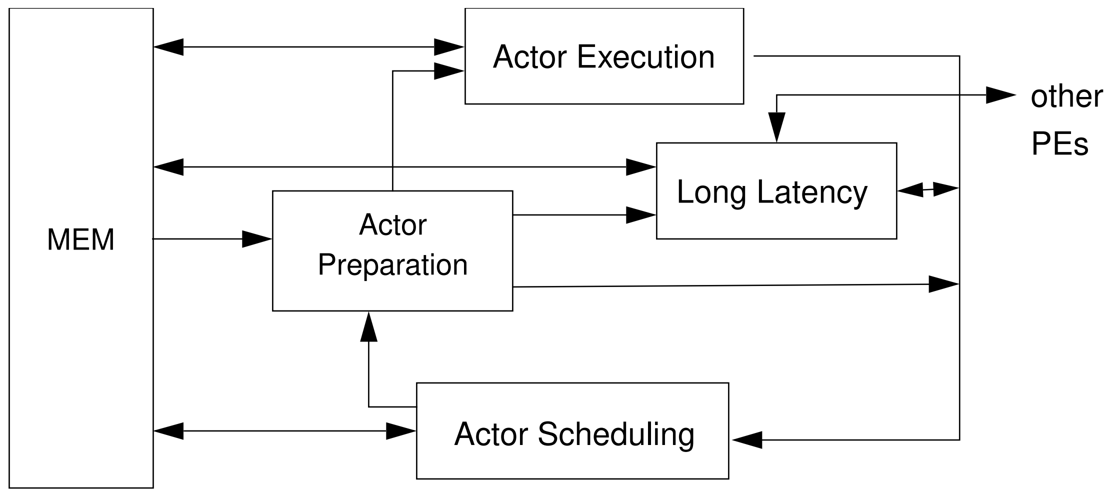


Figure 1.4 A Processing Element (PE) of a dataflow / von Neumann hybrid architecture [Hum91].

transaction strategy, they avoid latency-sensitive operations, which are split into two parts that separately initiate and then synchronize. And these two parts are partitioned into different large actors.

Architectural support for synchronization and scheduling of large actors depends on a number of issues, but the most basic is that of *strictness*. It is likely that the total input requirements for the actor will exceed that of the first instruction. The architecture may provide *strict* scheduling where all actor inputs must be present prior to invoking the actor [Hum91], or *non-strict* scheduling where invocation is based solely on the requirements of the instruction to be executed [Ian88]. In the latter scheme, the possibility of deadlock by partitioning is absent, while the hardware support to suspend actors is necessary.

An example of a processing element in dataflow / von Neumann hybrid architectures is shown in figure 1.4. The Actor Execution Unit (AEU) executes a large actor via a sequential pipeline of von Neumann style. Upon completion of an active actor, the execution unit sends a done signal to the Actor Scheduling Unit (ASU) indicating that the actor has been executed. The ASU, in turn, processes the signals and sends enabled actors to the Actor Preparation Unit (APU). There the enabled actors are enqueued for entry to either the execution unit, or the Long-latency actor Execution Unit (LEU). The LEU is responsible for fetching the long-latency instructions and necessary operands from the memory and processing the instructions.

The hybrid architectures proposed up to now have not been built even though their expected behavior has been simulated extensively. Also, it is expected that compilers for hybrid architectures would be more complicated than for pure dataflow machines. Until an actual machine of this architecture is built, there is no proof of its viability as a general purpose multiprocessor system.

1.2.3 Conventional von Neumann Architecture

In spite of the aforementioned shortcomings of von Neumann machines, most of the multiprocessors, existing or being built, are based on von Neumann processors. The appeal of von Neumann processors is that they are widely available and familiar. The significant overhead for synchronization and latency-sensitive operations in von Neumann processors precludes the use of fine-grain dataflow. Sarkar [Sak87], therefore, partitioned fine-grain dataflow graphs, based on SISAL [McG83], into a large-grain dataflow graphs whose optimal grain size is dependent on the overhead factor. The advantage of automatic partitioning is that the program is portable to any degree of parallel hardware. On the other hand, Suhler et. al. [Suh90] explore large-grain actors whose functions are textually described in conventional languages. They target the Sequent Balance, Intel iPSC-1, and a network of IBM PC RTs. Babb [Bab84] claims that even sequential programs are often easier to design and implement reliably when based on a large-grain dataflow model of computation. This approach, however, forces the programmer to explicitly decompose the program into tasks, and so degrades the performance if the programmer does not reveal enough parallelism in the program.

As general purpose multiprocessor systems, the idea of using large-grain dataflow graphs for von Neumann multiprocessors has not gained much attention. Most systems, instead, use parallel programming languages which are extended from conventional languages with special primitives for parallel tasks. However, signal processing applications offer a good opportunity to make this approach quite popular, as will be discussed later.

1.2.4 Interconnection Networks

Architectural models for a multiprocessor can be classified as being *tightly coupled* or *loosely coupled* [Hwa84]. Tightly coupled multiprocessors communicate through a shared main memory. Hence the rate at which data can be communicated from one processor to the other is on the order of the bandwidth of the memory. A small local memory or high speed buffer (cache) may exist in each processor (figure 1.5). Loosely coupled multiprocessors (also called multicomputers) communicate by passing messages. Again the performance and scalability of the multiprocessor is primarily determined by the interconnection network. The general structure for loosely coupled multiprocessors is

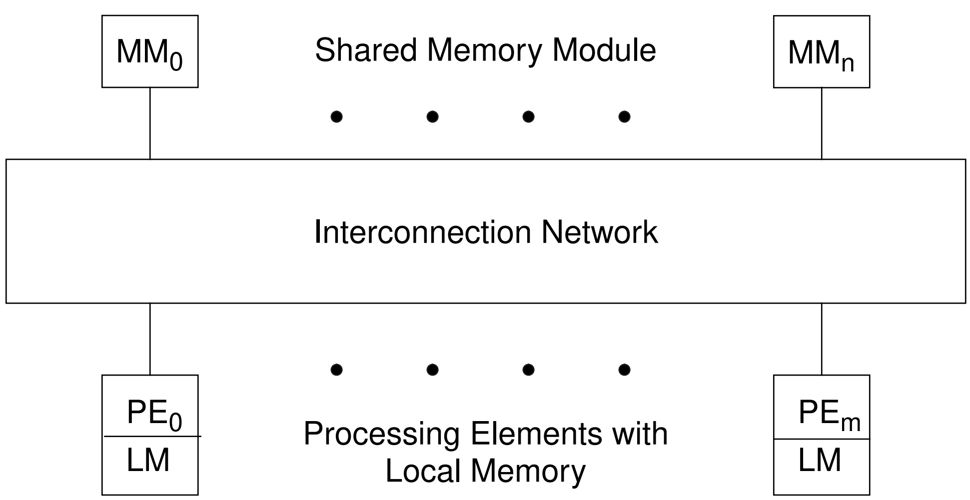


Figure 1.5 The general structure of tightly coupled multiprocessors.

shown in figure 1.6.

Interconnection networks have been reviewed in many surveys [Fen81][Ree87] [Wit81]. The network topologies tend to be regular and can be grouped into two categories: *static* and *dynamic* [Fen81]. In a static topology, links between two processors are passive and can not be reconfigured. On the other hand, links in the dynamic topology can be reconfigured by setting the networks's active switching elements.

Though different authors use different methods to characterize the performance of the static topologies, some common measures are widely accepted [Bhu84][Ree87][Wit81]: they are:

1. average message traffic delay (mean internode distance),
2. average message traffic density per link,
3. number of communication ports per node (degree of a node),
4. number of redundant paths (fault tolerancy),
5. and ease of routing (ease of distinct representation of each node).

A multitude of different topologies compromise these measures, offering a wide range of cost/performance choices. Typical topologies are shown in figure 1.7.The completely connected network with N processors has N-1 connections per node, so it is not suitable for even a moderate number of multiprocessors despite being optimal by all other

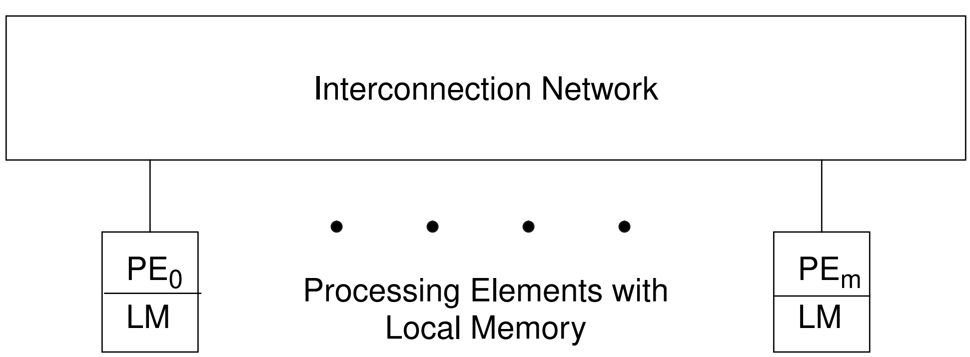


Figure 1.6 The general structure of loosely coupled multiprocessors

criteria. The simplest topology is the single shared-bus topology (e.g. Sequent Balance). Although the single bus topology is quite reliable and relatively inexpensive, it is not tolerant of a malfunction in any of the bus interface circuitry. Moreover, system expansion, by adding more processors or memory, increases the bus contention, which degrades system throughput. To provide more communication bandwidth than by a single bus, multiple bus architectures (e.g. Pluribus) and hierarchical structures with clusters connected by a inter-cluster bus (e.g. Cm*) have been developed. These architecture pay for the increase bandwidth with more complicated bus arbitration logic. The bus bottleneck problem limits the size of multiprocessors with bus topology (typically < 20).

General agreement on the order of magnitude of the average message traffic delay and the average message traffic density is the order of the number of the processors

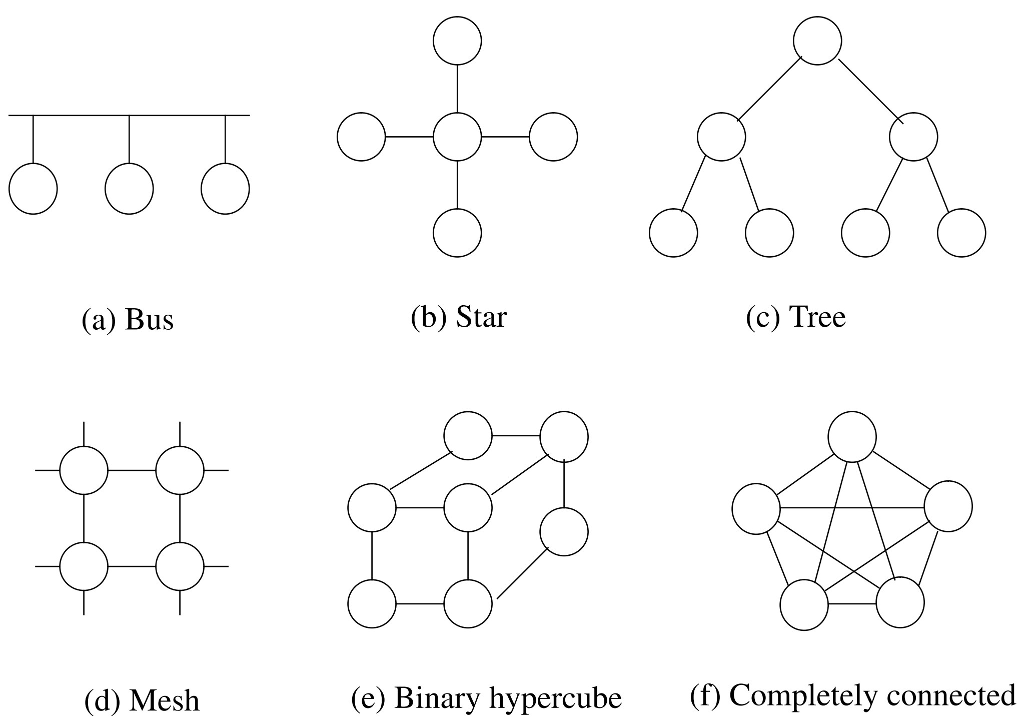


Figure 1.7 Typical static topologies of interconnection networks.

(*O(logN)*). There are some known network topologies satisfying this order: cube-connected-cycle [Pre81], lens [Fin81], dual-bus-hypercube[Wit81], generalized hypercube [Bhu84], and generalized nearest neighbor mesh [Wit81]. Among them, the binary hypercube, which is a special case of the generalized hypercube and also the generalized nearest neighbor mesh, is the most popular for moderate numbers of processors ranging from a few dozen to a few thousand processing elements (e.g. Intel iPSC, Ncube/10). The primary disadvantage of the binary hypercube is that it requires a logarithmic increase in degree of a node as the total number of nodes increases. Other common topologies include star, mesh, tree and their variants.

Dynamic topologies are mainly used for tightly coupled multiprocessors to connect processors to the shared memory modules. There are three topological classes in the dynamic category: single-stage, multistage, and crossbar. Examples of dynamic network topologies are shown in figure 1.8.The shuffle-exchange network [Sto71] is the representative single-stage network based on a perfect-shuffle connection cascaded to a stage of switching elements (e.g. NYU Ultracomputer). A multistage network consists of more than one stage of switching elements and is usually capable of connecting an arbitrary input terminal to an arbitrary output terminal. Depending on whether simultaneous connections of more than one terminal pair may result in conflicts in the use of the network communication links, multistage networks are further divided into three classes: blocking, rearrangeable, and nonblocking [Fen81]. Various kinds of multistage networks have been implemented: omega (e.g. BBN Butterfly, IBM RP3), banyan, delta, Benes, and so on.

In a crossbar switch, every input port can be connected to a free output port without blocking. While this scheme yields a high processor/memory bandwidth, it incurs scaling problems due to a switching cost which increases as N^2.

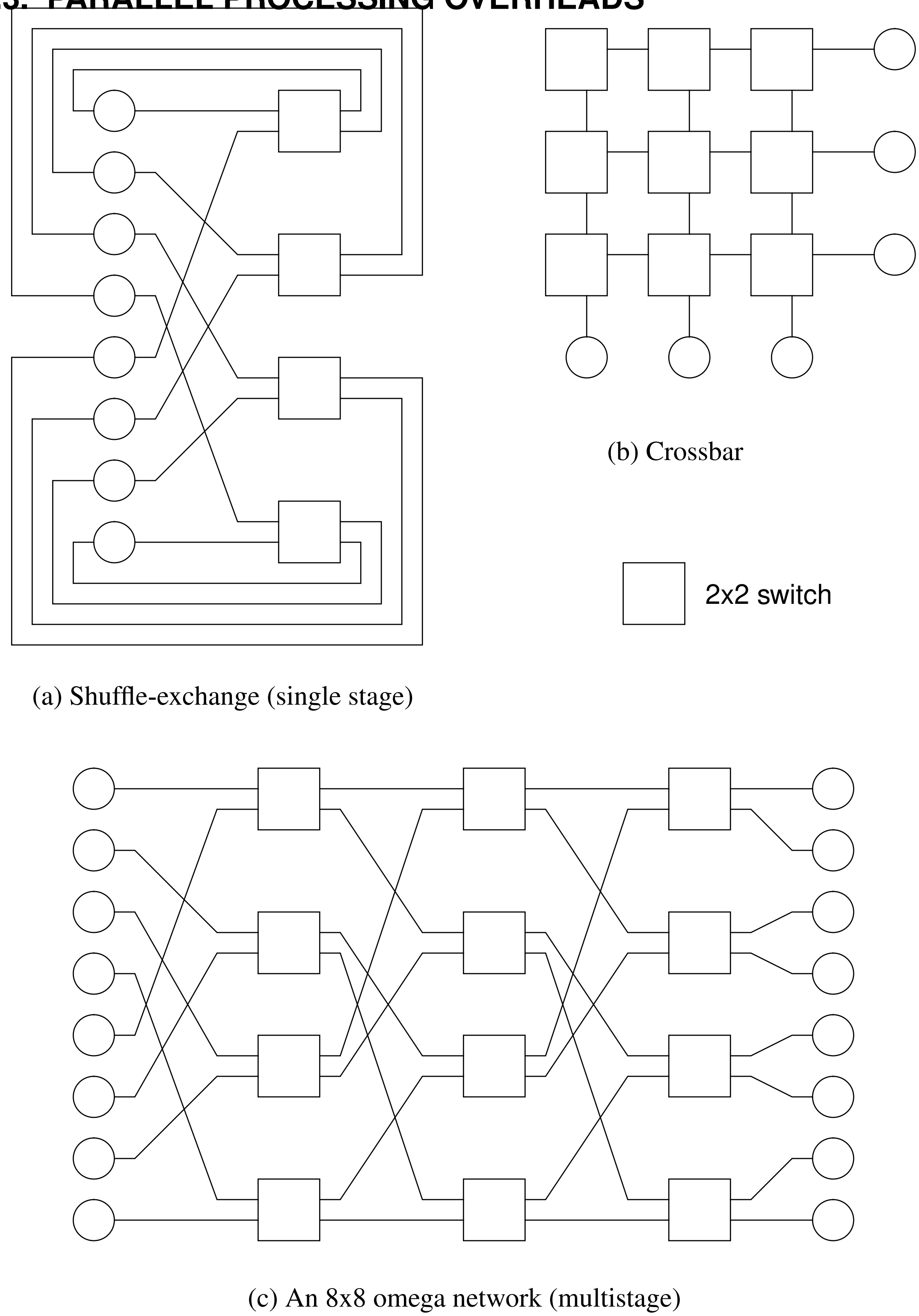


Figure 1.8 Examples of dynamic network topologies

Regardless of what hardware architecture or software programming paradigm is used, there are many factors which limit the attainable speedup in a parallel processing environment.

1.3.1 Load Balancing

Load balancing is the primary concern for parallel processing. A proper load balance distributes the computation load evenly across all of the processors to maximize efficiency, or to keep the processors busy as much as possible. A simple strategy to partition the actors with the same amount of total work usually results in a bad balance since some processors may be idled in case that the assigned actors are not executable. Therefore, the precedence relationship among actors should be taken into account. The finer the granularity of actors is, the less variance of the distributed workloads is expected. However, the load balancing scheme becomes more complicated due to the quadratic increase of precedence relations. Some dataflow machines were proposed to balance the loads at runtime with fine-grain dataflow graphs, which incur prohibitive runtime overhead of shipping codes across the interconnection network.

Another crucial issue, but often neglected in the literature, on load balancing is the *dynamic* (data-dependent) behavior of programs. For a program with dynamic behavior, there is no fixed partitioning which is best for all of the different runtime behaviors of the program.

1.3.2 Interprocessor Communication

Interprocessor communication is also a significant source of the extra overhead. It involves not only the communication delay required to transfer data between the source and destination processors but also involves the arbitration delay to acquire this access privilege to the interconnection network. The common technique to compensate for this overhead is to use a split transaction scheme with dedicated hardware for network access. The overhead can be hidden effectively if the processors are kept busy with other runnable actors during the transactions. This is the main advantage of fine-grain dataflow machines, because they usually have a sufficient number of runnable actors through full exploitation of parallelism.

The arbitration delay is non-deterministic and depends on the network congestion and the arbitration strategy. The primary objective of an arbitration strategy is to reduce the possibility of an extraordinarily long latency. One interesting approach is to randomize the memory access pattern for multistage dynamic networks. Excessive contention for a particular memory module in a dynamic network has been found to cause so-called “hot-spots” in the network, which are analogous to traffic jams. Since regular access patterns are more likely to cause these hot-spots, randomization destroys the regularity of access patterns, and so reduces the possibility of excessive contention according to probability theory.

Communication delay depends on the amount of data transmitted. Therefore, a pair of actors with large communication requirements should be assigned to the same processor. Thus, there is a conflict between reducing communication requirements and load balancing. Communication delay also depends on the length of the path. For static networks, data must typically be routed through intermediate nodes before reaching its final destination. In this case, the physical network topology as well as communication requirements affects the efficiency of the partitioning, which makes the problem of finding the optimal partitioning even more intractable.

There are two possible partitioning strategies based on heuristic rationales. The *unified* strategy incorporates the effect of physical processor connectivity on communication delay when partitioning [Sih91]. The other, called *two-phase* strategy, divides the scheduling problem into two phases.

1. Partition the program ignoring the specific communication network topology of the target machine.
2. Assign the partitioned actors to the physical processors.

Once the first phase is done, the communication requirements for each pair of partitions are determined. The objective of the second phase is to minimize the total communication delays, or message traffic. A message traffic on a link is defined as the volume of data exchanged through the link. The total message traffic is the sum of message traffics on all links. Define v_{ij} and d_{ij} as follows.

v_{ij} - the volume of data exchanged between processor i and j.

d_{ij} - the number of links on the shortest path between processor i and j.

Then the total message traffic becomes

$$\sum_{i,\,j} v_{ij} d_{ij} \qquad (1\text{-}1)$$

The problem of minimizing the total message traffic is well known as the quadratic assignment problem [Han72]. M. Hanan et. al. reviewed three techniques for the placement of logic packages: constructive-initial-placement, iterative-improvement, and branch-and-bound. While they did not consider network congestion, S. Lee and J. K. Aggarwal [Lee87] formulated a set of new object functions quantifying the effect of congestion based on deterministic information of when each communication occurs and with how much volume. The problem of minimizing network congestion with a given assignment may be attacked separately as the traffic scheduling problem [Bia87].

We examine the expected performance improvement we can achieve from the optimal assignment compared with a random assignment in the appendix. The analysis shows that the average performance improvement is about *20%* to *30%* ignoring the effects of the network congestion. Sih reported that the unified strategy can give higher performance improvement since it can reduce the quantities $\{d_{ij}\}$ [Sih91].

1.3.3 Other Factors

The local memory of a processor is limited in size so the number of active invocations of actors should be limited accordingly. In dataflow machines, the number of activities may grow indefinitely unless the degree of parallelism is restricted. It may create a deadlock condition when the local memory is filled with the contexts of the current activities. Therefore, it is a challenging task to manage limited resources without sacrificing too much of the parallelism for dataflow machines.

Various forms of synchronization are necessary to allow cooperation between processors, creating additional overhead. The amount of overhead depends on which scheduling scheme is used, which will be discussed in the next chapter.

1.4. TARGET APPLICATION: DIGITAL SIGNAL PROCESSING

So far, we have reviewed program representations and multiprocessor architectures for general purpose applications. Program representations and multiprocessor architectures are closely related. For conventional multiprocessors, fine-grain dataflow graphs has been proven very inefficient. Therefore, parallel languages of von Neumann type are preferred in spite of their shortcomings, such as difficulty in programming and debugging. The merits of fine-grain dataflow graphs come into life only with dataflow machines. Dataflow machines, however, are still immature and possess some difficulties, such as resource management and compatibility with conventional languages, to be overcome before they are commercially viable. Between these two extremes, there is large-grain dataflow (LGDF) which has gained little attention for general purpose applications.

In this dissertation, we focus on digital signal processing (DSP) applications.

They differ significantly from general purpose computations in that they are consistently numerically intensive, and many are *real-time* meaning that the full set of input data is not available before output data must be computed. In particular, for most such applications, algorithms are repetitively applied to an essentially infinite stream of input data. Also, runtime behavior is mostly deterministic. These characteristics of DSP applications are efficiently exploited by a combination of LGDF representation and conventional multiprocessors.

Digital signal processing algorithms are usually described in the literature by block diagrams consisting of functional blocks connected by dataflow paths. The blocks represent signal processing subsystems such as digital filters, FFT units, or adaptive equalizers. Each block may itself represent another block diagram, so the specification is hierarchical. This is consistent with the general practice in signal processing where, for example, a phase locked loop may be treated as a block in a large system, and may be itself a network of simpler blocks. In this case, block diagrams are large grain dataflow graphs.

Many block diagram system specifications have been developed to permit users to implement signal processing algorithms more naturally [Lee87b]. They differ in some description details but they use a common data-driven paradigm. Among them, we concentrate on the *synchronous dataflow* (SDF) graph and its extension, the *dynamic dataflow* (DDF) graph.

1.4.1 Synchronous Dataflow Graph

Synchronous Dataflow (SDF) is a special case of data flow (either fine-grain or large-grain) in which the number of data samples produced and consumed by each node on each invocation is specified a priori [Lee87b]. An example of an SDF graph is shown below in figure 1.9. The numbers at the tail and head of each arc indicate the number of data samples consumed and produced by the respective nodes. For example, node B consumes two data samples from node A and one data sample from node D, and produces one data sample to node C. The inscription 1D on the arc between node C and D indicates the presence of a *delay* in the signal processing sense, corresponding to a sample offset between the input and the output. In other words, the *n*th sample consumed by node D will be the *(n-1)*th sample produced by node C. This implies that the first sample D consumes is not produced by C at all, but is part of the initial state of the arc's first-in first-out (FIFO) buffer. This initial data sample is necessary to avoid the deadlock condition, where each node waits for data from its predecessor.

Systems where all sample rates are rational multiples of all other sample rates are called *synchronous* in the signal processing literature. Synchronous DSP systems are easily described using the SDF paradigm, hence the name for the paradigm. For example, a 2:1 decimator node would have one input and one output, but would consume two tokens for every token produced. Thus, relative sample rates are represented by the numbers attached to the input or output arcs of nodes. One requirement of SDF graphs is that sample rates should be consistent. Inconsistent sample rates can lead to deadlock, or unbounded memory requirements. *Consistency* is defined to mean that the same number of tokens are consumed as produced on any arc, in the long run. A systematic method for consistency checking was developed for SDF graphs [Lee87b], and generalized further

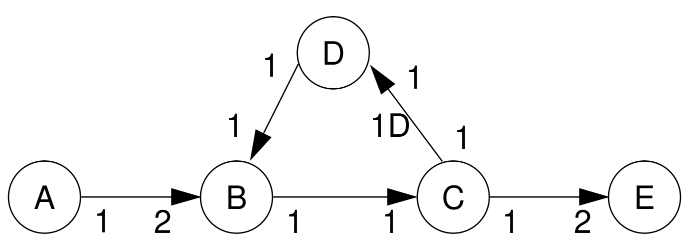


Figure 1.9 A synchronous dataflow graph

for dataflow languages [Lee91b].

The most salient feature of the SDF paradigm is that the execution order of nodes can be determined at compile-time. Moreover, if the execution time of each node is known and fixed beforehand, nodes can be partitioned optimally at compile-time. Deadlock avoidance and bounded memory requirements can also be guaranteed at compile-time. As a result, the runtime supervisory overhead can be replaced with the corresponding compiler task. Recall that dataflow architectures are required to preserve the merits of dataflow graphs for general purpose applications. However, they seem to be unnecessary for SDF graphs, because what they do in hardware can be done at least as well by a compiler at much lower cost. Many researchers follow this line: translate block diagram descriptions of signal processing algorithms into run-time code for multiple programmable DSP processors of von Neumann type [Lee89a][Zis87][Tha90].

SDF graphs consist only of synchronous actors where the number of tokens produced and consumed must be independent of the data. Most nodes for signal processing applications are synchronous and this synchrony is taken advantage of through compile-time analysis. However, the SDF paradigm is too restrictive to express general signal processing applications. Dynamic dataflow graphs overcome this difficulty by allowing asynchronous actors in the specification.

1.4.2 Dynamic Dataflow Graph

Some asynchronous nodes are shown in figure 1.10. The SWITCH node consumes one data input from the input arc and one boolean input from the control arc, **C**. Depending on the Boolean value received from the control arc, it produces one output either to the true arc, **T**, or to the false arc, **F**. The number of data samples produced on the two outputs, therefore, is data-dependent. On the contrary, the SELECT node consumes one input either from the true arc or from the false arc, depending on the boolean value received from the control arc. The figure also illustrates two data-dependent up-sample nodes and one data-dependent down-sample node. The number of data samples produced on the output arc of the REPEATER is determined by the control value, hence it is data-dependent. The DownCounter node generates down-counted integer samples starting from the value of the input sample. The LastOfN node consumes x input samples and produces one output sample, where x is determined by the control value received from the control arc. Dynamic dataflow (DDF) graphs consist of both asynchronous nodes and synchronous nodes [Lee88][Buc91a]. By using asynchronous nodes, the DDF paradigm can represent the well-known dynamic constructs such as if-then-else, for-loop, and do-while-loop; thus it overcomes the main modeling limitation of the SDF paradigm (figure 1.11). In figure 1.11 (b), the diamond containing D on the arc connected to the control input of the SELECT node represents a logical delay, or an initial data sample. The initial Boolean value should be "false" so that the first token selected comes from the outside, rather than from the feedback loop.

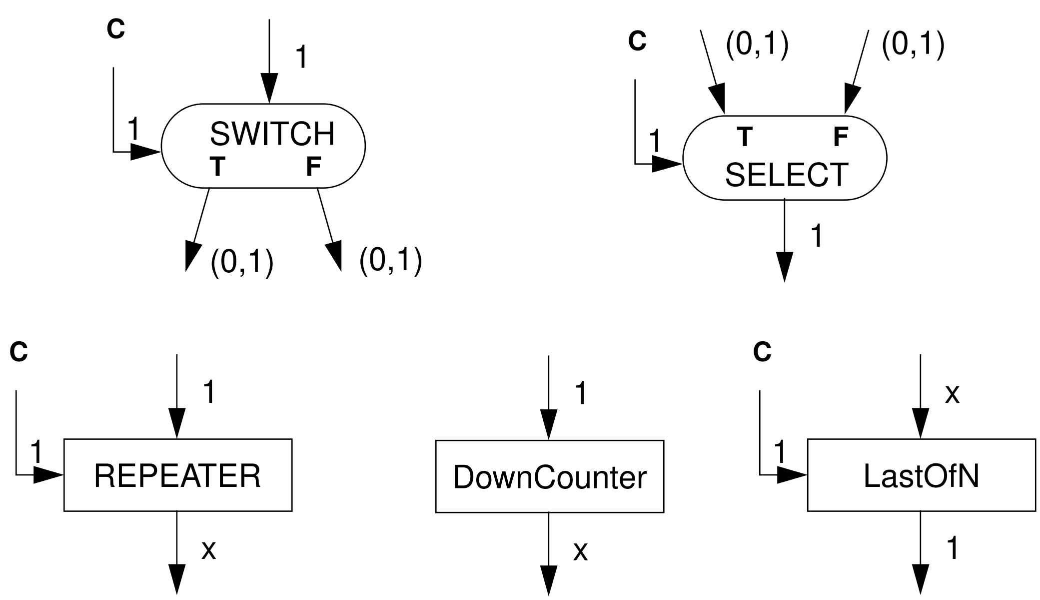


Figure 1.10 Some examples of asynchronous nodes.

Dynamic dataflow graphs sacrifice the run-time efficiency of SDF graphs for the enhanced modeling power. The execution order of the nodes of a DDF graph can not be fully specified at compile-time. For example, in the *if-then-else* construct in figure 1.11 (a), either the f(x) node or the g(x) node is fired after the SWITCH node, depending on the control value, c, which is known at run-time only. The requirement of runtime decision making turns some researchers' attention to dataflow digital signal processors like dataflow machines for general applications: the Hughes Dataflow Machine (HDFM)

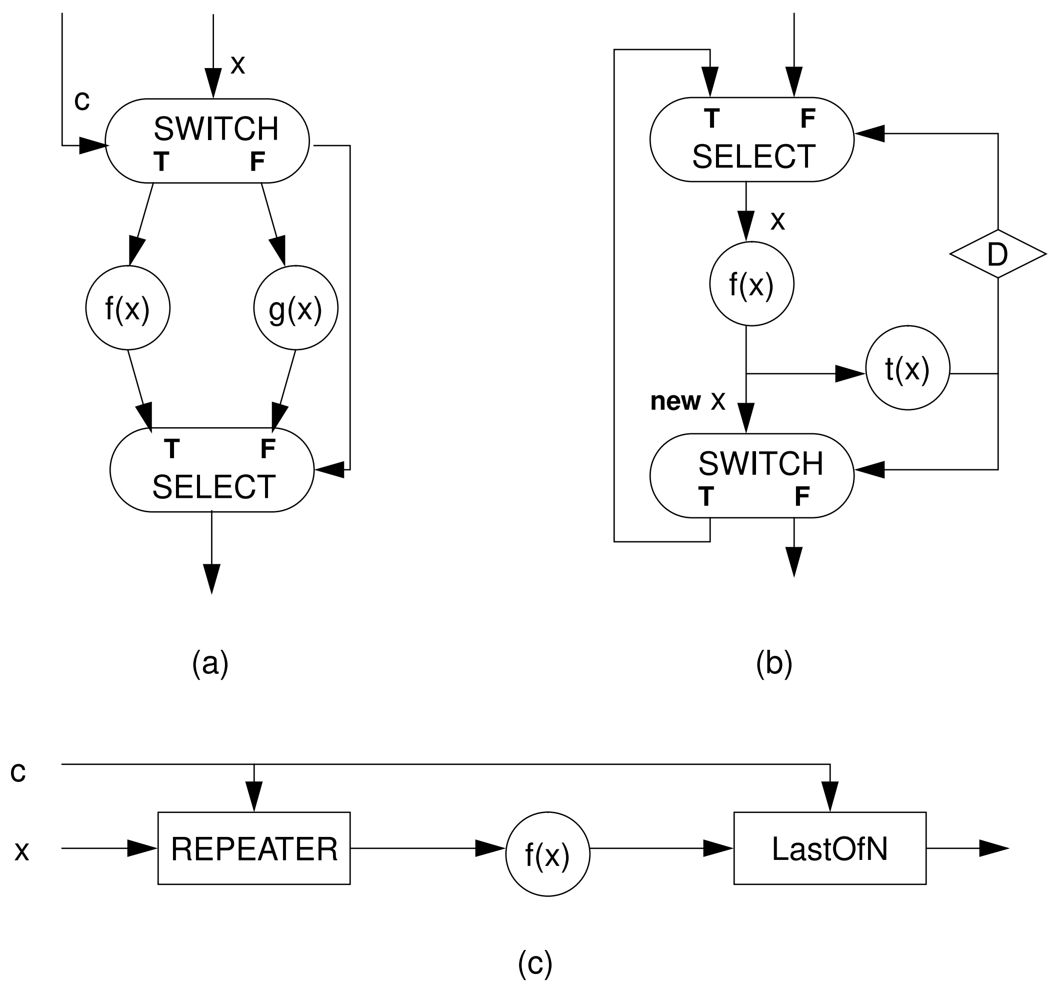


Figure 1.11 Dynamic dataflow graphs that model some familiar dynamic constructs: (a) if c then f(x) else g(x), (b) do x = f(x) while t(x), and (c) for (i = c; i > 0; i--) f(x).

[Gau85], the NEC uPD7281. [Cha84] and so on. Dataflow DSPs, however, are operated by fine-grain dataflow graphs, which again possess the same difficulties as the general purpose dataflow machines discussed earlier. J. Gaudiot [Gau87] concludes that dataflow DSPs find their applications in problems which involve large amounts of heuristics and decision making.

Digital signal processing algorithms provide a unique set of opportunities; they often have predictable, or mostly predictable control flow. The overhead of dataflow DSPs seems too high a price to pay to manage a small amount of run-time decision making. A challenging research area is to incorporate decision making capability and at the same time preserve the efficiency of dataflow execution in multiple conventional DSPs of von Neumann style.

1.5. CONCLUSION

At least for signal processing applications, data-driven principles of execution are a necessity in the design of multiprocessor systems, be they incorporated at compile or runtime. The granularity of parallelism presents a trade-off between managing dynamic behavior and utilizing highly efficient sequential, control-flow execution. The optimal granularity, therefore, is application-specific. In particular, many digital signal processing applications match well with large-grain dataflow graphs. Regardless of what hardware architecture or software programming paradigm is used, there are fundamental difficulties that arise when using multiple processors: load balancing and communication overhead. The efficient coordination of processors requires both a program partitioning and a scheduling strategy. In this dissertation, we focus on applications that have at most a small amount of data-dependent behavior, which covers most signal processing applications and computation-intensive applications. Since we make no restriction here about the granularity of the dataflow graph, the proposed techniques are valid both for fine-grain and large-grain even though we assume large-grain dataflow graphs throughout the dissertation.

2

SCHEDULING

For you yourselves know how you ought to follow us, for we were not disorderly among you; ...... Not because we do not have authority, but to make ourselves an example of how you should follow us.

--- II Thessalonians 3:17,19

The processor scheduling problem is to map a set of precedence-constrained tasks $\{T_i\}$ $i = 1 \ldots$ n, onto a set of processors $\{P_k\}$, $k = 1 \ldots$ p in order to satisfy a specified objective. An acyclic precedence graph is commonly used to describe the interrelationships among tasks where an arc A_{ij} directed from task T_i to T_j indicates that T_i must precede T_j in execution. While processor scheduling has a very rich and distinguished history [Cof76] [Gon77], most efforts have been focused on *deterministic* models, where the execution time of a task T_i on a processor P_k is fixed and there can be no conditional (data-dependent) nodes in the program graph.

There are a number of factors that can be gauged to classify the scheduling techniques: task interruptibility, deadlines, processor homogeneity, and so on [Gon77]. If the interruption (and subsequent resumption) of a task before its completion is permitted, we speak of *preemptive* scheduling. In *nonpreemptive* scheduling, any task which has started execution on a processor must be completed. Preemptive scheduling should consider the context-switching overhead to assess the gain of preemption. Ignoring that overhead, preemptive scheduling generates schedules that are better than those generated by nonpreemptive scheduling. In this thesis, we restrict our attention to nonpreemptive scheduling.

In real-time applications, *deadlines* or *scheduled completion times* may be established for individual tasks T_i. If the deadlines are enforced for each execution, we speak of a *hard real-time* schedule. In a *soft real-time* schedule, the deadline requirement is based on a statistical distribution of terminations. We exclude hard real-time disciplines throughout the thesis. We also assume that the processors are all identical.

Deterministic schedules are usually displayed with timing diagrams known as *Gantt charts* as shown in figure 2.1. In this schedule, three processors are involved. The tasks assigned to each processor and their order of execution and execution time requirements are represented by the horizontal lines and task identifications. The dark area represents the idle time of the associated processor.

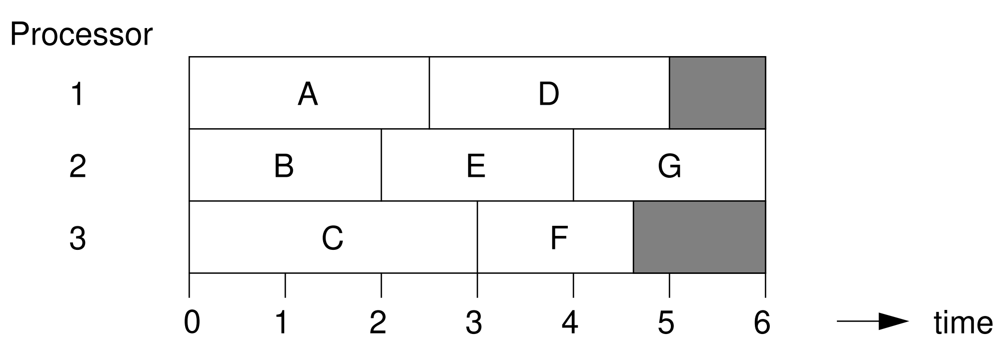


Figure 2.1 A task schedule in Gantt chart form.

2.1. SCHEDULING OBJECTIVES

A program graph may be executed only once, or repeated at irregular intervals over a long period of time. The scheduling objective, in this case, is either (1) to minimize the finishing time, also called *makespan*, of the program with a given number of processors, or (2) to minimize the number of processors required to process the program in the smallest possible time [Ram72]. A program graph is first converted to a homogenous dataflow graph, and finally to an acyclic precedence graph. The second conversion is accomplished by splitting ideal delays into a pair of input and output nodes. The output node is connected to the source node of the removed arc, and the input node is connected to the destination node of the arc. Then, the smallest possible time is nothing but the longest path, called *critical path*, in terms of computational delay between inputs and outputs in a given acyclic precedence graph. In multiprocessor scheduling, the number of processors is fixed, so we focus on minimizing the finishing time of a given program.

In most DSP applications, however, the program is executed once for every sample of an input stream. For such iterative executions, the objective is to maximize the throughput or to minimize the iteration period. A tightest lower bound on the iteration period, referred as the *iteration period bound*, can be obtained from a dataflow graph [Ren81]. The iteration period bound (T_o) is

$$T_o = \max_{l \in loops} \left\{ \frac{D_l}{n_l} \right\}, \qquad D_l = \sum_{j \in l} d_j \tag{2-1}$$

where n_l is the total delays in loop l, d_j is the execution time of the j th node on the l th loop, and D_l represents the total execution time of the l th loop. The quantity inside the brackets $\{D_l\ /\ n_l\}$ is called the *loop bound*, T_l. The loops which have the largest loop bound are called the *critical loops*. The critical loops determine the iteration period bound of a dataflow graph. A schedule that achieves the iteration period bound is said to be *rate optimal*.

If the number of processors is not fixed, another possible objective is to minimize the required number of processors to implement a rate optimal schedule, which is lower bounded by the *processor bound*, P_o:

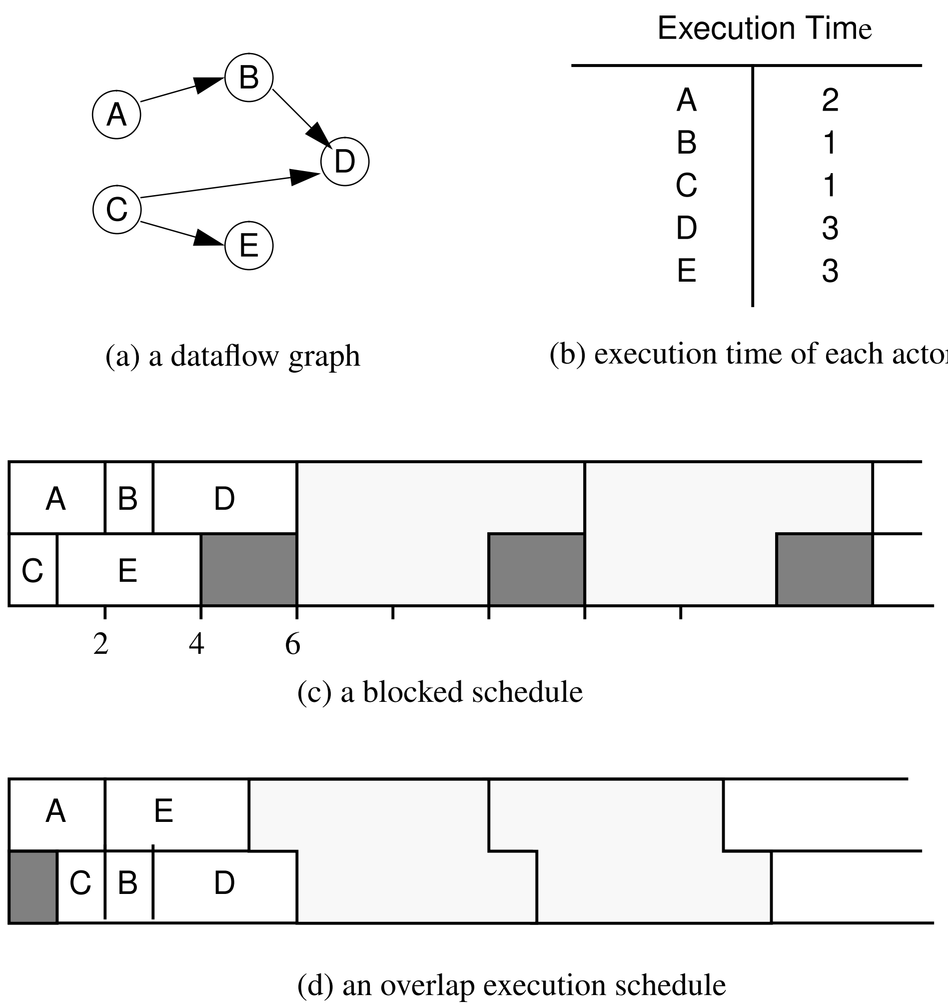
$$P_o = \left\lceil \frac{D}{T_o} \right\rceil, \qquad D = \sum_j d_j \tag{2-2}$$

where D is the total execution time of the program executed sequentially, and T_o is the iteration period bound. The processor bound may not be attainable since the above formula ignores precedence constraints although it enforces load balancing. A *processor optimal* schedule uses the minimum number of processors possible.

In addition to rate and processor optimality criteria, delay optimality can also be defined. A *delay optimal* schedule minimizes the time delay between a pair of input and output nodes. Another indirect measure of performance is the processor efficiency which is defined as the ratio of the average busy time of the processors to the iteration period. For a rate optimal schedule, the difference between 100% and the true processor efficiency is called the *inherent processor inefficiency.*

2.1.1 Blocked Schedule

Although we are interested in iterative execution cases, we aim to minimize the makespan as the scheduling objective assuming that the whole system is committed to one execution at a time. This falls under the category of *non-overlap execution scheduling* according to P. Hoang [Hoa90]. We construct a *blocked schedule* where each iteration cycle terminates before the next cycle begins. The iteration period becomes the length of



(a) a dataflow graph

(b) execution time of each actor

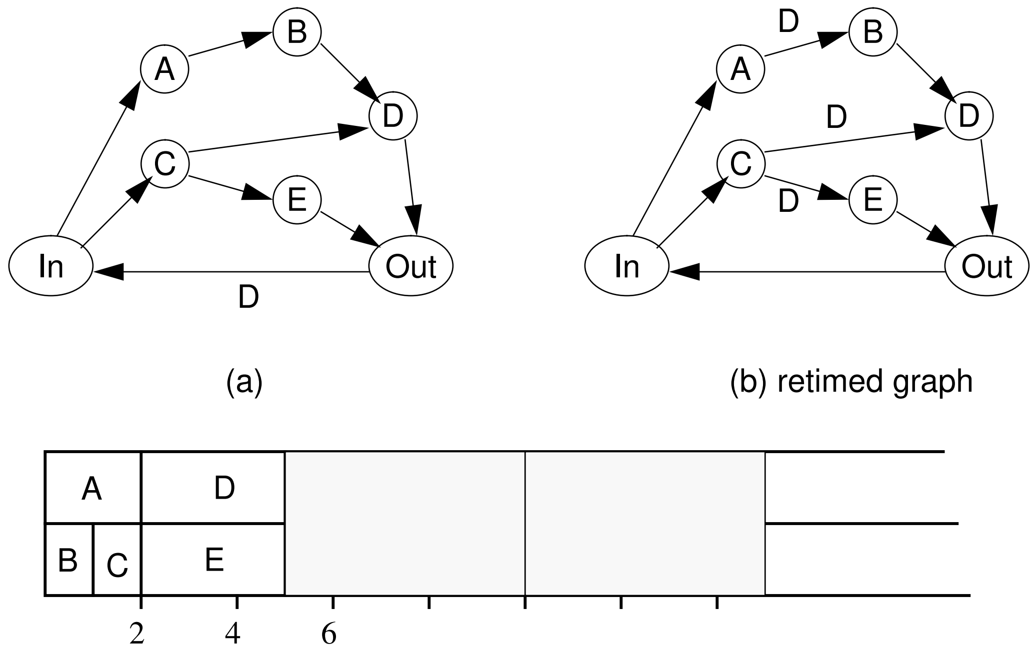
(c) a blocked schedule

(d) an overlap execution schedule

Figure 2.2 A comparison of a blocked schedule and an overlap execution schedule. (a) A dataflow graph has two input actors and two output actors whose execution times are shown in (b). A blocked schedule (c) is made so that the iteration period is 6 time units, and each period has 2 idle time units on the second processor. An overlap execution schedule (d) shows the optimal throughput, 5 time units.

one cycle of a blocked schedule, which is also the reciprocal of the throughput. Certainly, the non-overlap execution schedule does not guarantee the rate-optimal realization because it places artificial boundaries between iterations of the graph. On the other hand, *overlap execution scheduling* allows overlapped execution of successive iterations. A comparison of these two categories is illustrated in figure 2.2. When a blocked schedule is constructed, all processors are synchronized after each cycle of iteration so that the *pattern of processor availability* is flat before and after each cycle (meaning that all processors become available for the next cycle at the same time). As a result, the second processor is padded with no-ops, two time units of idle time. On the other hand, the optimal throughput is achieved by an overlap execution schedule. At the sixth time unit, the first processor begins execution of the next iteration with actor A, while the second processor is still processing actor D of the current iteration. The main motivation of using blocked scheduling is to reduce the computational complexity of scheduling.

However, it is possible to improve the throughput performance of a nonoverlap execution schedule. One technique we may use is *loop winding* [Gir87], which basically pipelines the program graph. In spite of mapping pipeline stages into a pipeline structure of hardware, it shares processors between stages to achieve a functional pipeline. The *retiming* [Lei83] technique can be used for optimal pipelining [Pot91]. Retiming was originally developed to alter the clock period of a synchronous circuit by relocating registers. The retiming transformation is performed on the dataflow graph before it is converted to the acyclic precedence graph. If a graph contains cycles, the iteration period is limited by the iteration period bound. Retiming provides a systematic method for transforming a graph so that the iteration period approaches this bound. In iterative execution cases, the whole graph can be thought of as the body of a cycle. By adding dummy input and output nodes and connecting them with some delays, the graph becomes ready for the retiming transformation. An example of the retiming transformation and the resulting schedule are shown in figure 2.3. Note that after the retiming transformation, the original dataflow graph is functionally pipelined. In the dataflow context, the delays represent initial data samples, which should be provided beforehand. For instance, the first iteration cycle could be processed to produce the initial samples before processing the retimed graph. Or, we can make a schedule preamble of actors A, C in figure 2.3 for example.



(c) a blocked schedule

Figure 2.3 (a) A dummy input node and a dummy output node connected with a unit delay are added to a graph. (b) The graph of (a) is retimed. (c) The blocked schedule of the retimed graph shows the optimal throughput in this example.

After scheduling actors A, C as a preamble, we schedule actors B, D, E of the current execution cycle and actors A, C of the next execution cycle using blocked scheduling. Instead of applying the retiming transformation, in practice, the programmer may insert delays on a certain cutset of the graph to realize a functional pipeline to expose more parallelism between iterations.

It may be possible to expose the hidden parallelism between iterations by increasing the blocking factor. The *blocking factor* corresponds to how many iterations are expressed in a program graph. If we assume that successive invocations of the same actor can not overlap in time, the new dataflow graph with blocking factor 2 and the corresponding blocked schedule becomes as displayed in figure 2.4. The figure illustrates that by increasing the blocking factor the throughput can be increased in many cases. In this particular example, the corresponding blocked schedule produces the optimal schedule. The costs incurred by an increasing blocking factor are two-fold; more memory is required in each processor to store the longer schedule, and scheduling will take longer because more nodes are present in the acyclic precedence graph. To our knowledge, the problem of finding the optimal blocking factor is still open except a special case, in which the execution time of each actor is the same, or unity [Cha92].

Sometimes, the program graph can be modified by changing the algorithm; digital filters for instance to reduce the iteration period bound [Par89a][Par89b].

2.2. A SCHEDULING TAXONOMY

The Scheduling of parallel computations consists of assigning actors to proces-

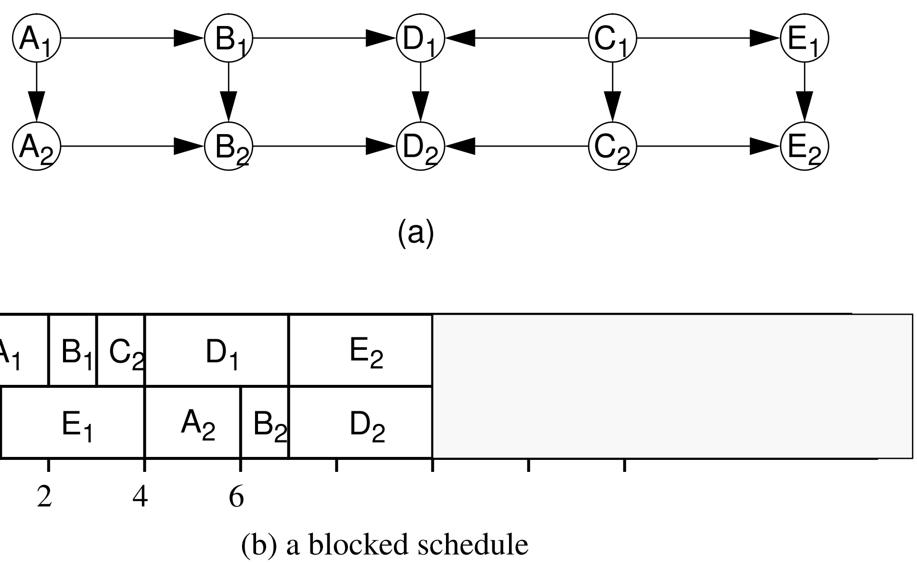


Figure 2.4 (a) A dataflow program graph derived from figure 2.2 (a) by increasing the blocking factor to 2. (b) The optimal blocked scheduling optimizes the throughput in this case. It shows that increasing the blocking factor is a way to increase the throughput of a nonoverlap execution schedule.

	assignment	ordering	timing
fully dynamic	RUN	RUN	RUN
static-assignment	COM-	RUN	RUN
self-timed	COM-	COM-	RUN
fully static	COM-	COM-	COM-

Figure 2.5 The time which the scheduling activities “assignment”, “ordering”, and “timing” are performed is shown for four classes of schedulers. The scheduling activities are listed on top and the strategies on the left [Lee89b].

sors, specifying the order in which actors fire on each processor, and specifying the time at which they fire. These tasks can be done either at compile time or at run time. Depending on which operations are done when, we define four classes of scheduling, depicted in figure 2.5.

The first type is *fully dynamic,* where actors are scheduled at run time only. When all input operands for a given actor are available, the actor is assigned to an idle processor and fired. The second type is *static assignment*, where an actor is assigned to a processor at compile time and a local run-time scheduler invokes actors assigned to the processor based on data availability. In the third type of scheduling, the compiler determines the order in which actors fire as well as assigning them to the processors. At run-time, the processor waits for data to be available for the next actor in its ordered list, and then fires that actor. We call this *self-timed* scheduling because of its similarity to self-timed circuits. The fourth type of scheduling is *fully static*; here the compiler determines the exact firing time of actors, as well as their assignment and ordering. This is analogous to synchronous circuits. As with any taxonomy, the boundary between these categories is not rigid. Self-timed scheduling and fully static scheduling are both called *static* scheduling.

We can give familiar examples for each of the four strategies applied in practice. Fully dynamic scheduling has been applied in the MIT static dataflow architecture [Den80], the LAU system, from the Department of Computer Science, ONERA/CERT, France [Pla76], and the DDM1 [Dav78]. It has also been applied in a digital signal processing context for coding vector processors, where the parallelism is of a fundamentally different nature than that in dataflow machines [Kun87]. A machine that has a mixture of fully dynamic and static-assignment scheduling is the Manchester dataflow machine [Wat82]. Here, 15 processing elements are collected in a ring. Actors are assigned to a ring at compile time, but to a PE within the ring at run time. Thus, assignment is dynamic within rings, but static across rings.

Examples of static-assignment scheduling include many dataflow machines [Sri86]. Dataflow machines evaluate dataflow graphs at run time, but a commonly adopted practical compromise is to allocate the actors to processors at compile time. Many implementations are based on the tagged-token concept [Arv82]; for example TI's data-driven processor (DDP) executes Fortran programs that are translated into dataflow graphs by a compiler [Cor79] using static-assignment. Another example (targeted at digital signal processing) is the NEC uPD7281 [Cha84]. The cost of implementing tagged-token architectures has recently been dramatically reduced using an "explicit token store" [Pap88]. Another example of an architecture that assumes static-assignment is the proposed "argument-fetching dataflow architecture" [Gao88], which is based on the argument-fetching data-driven principle of Dennis and Gao [Den88].

When there is no hardware support for scheduling (except synchronization primitives), then self-timed scheduling is usually used. Hence, most applications of today's general purpose multiprocessor systems use some form of self-timed scheduling, using for example Communicating Sequential Processes (CSP) principles [Hoa78] for synchronization. In these cases, it is often up to the programmer, with meager help from a compiler, to perform the scheduling. A more automated class of self-timed schedulers targets wavefront arrays [Kun88]. Another automated example is a dataflow programming system for digital signal processing called Gabriel that targets multiprocessor systems made with programmable DSPs [Lee89a]. Taking a broad view of the meaning of parallel computation, asynchronous digital circuits can also be said to use self-timed scheduling.

Systolic arrays, SIMD (single instruction, multiple data), and VLIW (very large instruction word) computations [Fis84] are fully statically scheduled. Again taking a broad view of the meaning of parallel computation, synchronous digital circuits can also be said to be fully statically scheduled.

As we move from fully dynamic to fully static, the compiler requires increasing information about the actors in order to construct good schedules. However, assuming that the information is available, the ability to construct deterministically optimal schedules increases. The *domain* of a scheduling strategy can be loosely defined as the set of algorithms for which the scheduling strategy does well. The *range* is the set of architectures that the strategy can target well. Most practical scheduling strategies have a limited domain or range.

2.2.1 Fully Static Scheduling

Of the classes we have defined, fully static scheduling has the narrowest domain. The subclass of dataflow graphs for which fully static scheduling works best is synchronous data flow [Lee87a] with all actors having known execution times. Unfortunately, even in this restricted domain, algorithms that accomplish such optimal scheduling have combinatorial complexity, except in certain trivial cases [Cof76][Cap84]. Therefore, good heuristic methods have been developed over the years. The typical approach is based on *list scheduling*, in which actors are assigned priorities and placed in a list and executed in order of decreasing priority [Ada74][Cof76][Sih90b]. Other approaches such as clustering [Kim88][Sar87], declustering [Sih91] and 0-1 integer programming [Kon90] also have been proposed. These heuristics all aim to minimize the schedule length (*makespan* of the program). This approach is adequate for latency-sensitive applications or in situations where barrier synchronization between iterations of the schedule is required.

In most DSP applications, however, the program is executed once for every sample of an input stream. As a result, by overlapping executions of successive iterations, the computational throughput can be improved. Typical approaches are based on cyclo-static scheduling [Sch85][Gel91], which is rate, processor and delay optimal. However, the exponential worst-case running time of the scheduling algorithm, the lack of consideration for resource constraints and the extensive communication requirements preclude a practical implementation, possibly except for simple structures such as digital filters. *Chain partitioning* [Bok88] represents another technique based on pipelining serial tasks on a linear array of processors to maximize throughput; Its domain is again quite limited. Recently, a heuristic that simultaneously considers pipelining, retiming, parallelism and hierarchical node decomposition has been proposed as part of a software environment for partitioning DSP programs onto a configurable multiprocessor system [Hoa90].

The *range* depends on the sophistication of these methods, although most straightforward implementations target homogeneous tightly coupled multiprocessors with full interconnectivity. The requirement for full interconnectivity limits the range to machines with modest parallelism.

A subclass of fully static scheduling is the set of techniques based on projecting dependence graphs for *regular iterative algorithms* onto systolic arrays [Kun88] [Rao85]. These techniques have a very limited domain (RIA's) and range (systolic arrays), but do extremely well within these constraints.

Fully static scheduling has the lowest run-time cost: no hardware and no software.

Since behavior is completely known at compile time, there is no need to check at run time to see when actors can be fired. The compiler can figure it out, so actors simply fire at the designated time, and are assured that their data is available.

The domain of static scheduling specifically does not include dataflow graphs with actors that have data-dependent execution times or actors that may or may not fire, depending on the value of some data somewhere in the graph. These restrictions are severe, since they exclude both conditionals and data-dependent iteration within an actor or involving several actors. The restrictions can be relaxed, however, at the expense of optimality in the resulting schedule. For example, an actor with a data-dependent execution time can be padded so that it always executes in worst-case time. This is not so bad when the application has hard real-time constraints, but otherwise may be very costly. As another example, to implement the synchronous dataflow equivalent of if-then-else, both branches of the conditional may be computed, and the desired result may be selected. There are again applications where this option is acceptable, for example when one of the two conditional branches is trivial, but most of the time the cost will be high. The concept of static scheduling has been extended to solve some of these problems, using a technique called *quasi-static scheduling* [Lee88]. In quasi-static scheduling, some firing decisions are made at run-time, but only where absolutely necessary.

2.2.2 Self-Time Scheduling

Self-timed scheduling has a slightly broader application domain than fully static scheduling. Although the order of execution of actors is fixed for each processor at compile time, the exact firing times are not. Consequently, the schedule can automatically compensate for certain fluctuations in execution times. For example, if one actor finishes earlier than expected, the following actor can fire immediately, as long as its data is available. This means that the second actor can finish later than expected without any loss in overall speed. Compared to using worst-case execution times, self-timed scheduling will always do at least as well, as long as the overhead for synchronization is negligible. Self-timed scheduling is also more robust. Minor fluctuations in execution times will not affect the correctness of the execution, and will have little effect on its performance. For example, interrupts are often useful for handling I/O operations, but their introduction introduces uncertainty in the execution of any actor that may be interrupted. The Gabriel system [Lee89a] uses self-timed scheduling successfully for modestly parallel target architectures.

It is not quite certain just how restricted the domain of self-timed scheduling is, even though it does well with synchronous dataflow when the variability of the execution times of the actors is small enough. In the presence of limited amounts of data-dependent firing of actors, it will also do well. Loosely, it appears that the domain can be stated simply as the set of algorithms that are "reasonably" static. It is not clear just how much dynamic behavior must be present before enough time is spent idling, waiting for data tokens to arrive, that a more dynamic scheduling technique would have been more effective. Nonetheless, this domain seems like a good match to signal processing, for which practical implementations of most well-known algorithms occasionally require data dependencies.

In self-timed scheduling, there is no need to determine at run time which actor to invoke next. The ordering is specified at compile time. The only run-time scheduling function is to determine whether the actor that goes next is ready to be fired. If it is not, then the machine is idled until it is. It is up to the compile-time scheduler to find the ordering that minimizes the idling time. Determining whether the actor is ready to fire is a simple matter of handshaking. One-bit semaphores, using a full/empty discipline, are sufficient. This type of mechanism is very commonly used on multiprocessor machines that are not dataflow machines.

2.2.3 Static Assignment Scheduling

Static assignment scheduling, in principle, has a still broader domain, because it can adjust the ordering of the execution of actors. An example is shown in figure 2.6. In that example, one of six actors, D, has a data-dependent execution time. Depending on the outcome, it may be better to schedule the actors using the ordering in (b) or in (c). While it is possible to reduce the total execution time by rearranging the order of execution, it is not always easy to determine at run time which actor should be fired next when there is more than one possibility. For example, in (c), after A completes on the second processor, either B or E can be fired. For the implementation to be cost effective, the decision would have to be made on the basis of local or static (compile-time) information.

Static-assignment scheduling is a compromise that admits data dependencies, although all hope of optimality must be abandoned in most cases. Although static-assignment scheduling is commonly used, compiler strategies for accomplishing the assignment are not satisfactory. The main techniques include clustering [Efe82][Chu87], 0-1 integer programming [Chu80], or simulated annealing [Zis87] techniques. But none of

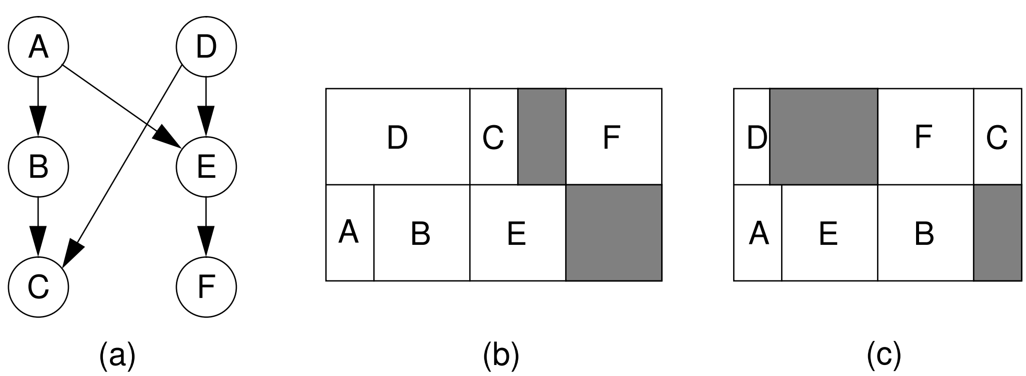


Figure 2.6 Two static-assignment schedules for two processors are shown for the precedence graph in (a). The execution time of actor D is data-dependent and is longer for the schedule in (b) than for the schedule in (c). Note that the ordering of the firing of actors is determined at run time and is different in (b) and (c).

these consider precedence relations between actors. To compensate for ignoring the precedence relations and to regain optimality, some researchers propose a dynamic load balancing scheme at run-time [Kel84][Bur81][Iqb86]. Unfortunately, the cost can be nearly as high as fully dynamic scheduling. Others have attempted, with limited success, to incorporate precedence information in heuristic scheduling strategies. For instance, Chu and Lan use very simple stochastic computation models to derive some principles that can guide heuristic assignment for more general computations [Chu87].

Static-assignment schedulers have an easier time at run-time because there is no need to determine how to assign actors to processors. Control becomes localized, because each processor only has to worry about the actors that have been assigned to it. A simple “greedy” scheduling algorithm simply fires an actor immediately when the processor becomes free, assuming there is an actor ready to be fired. This is not optimal because it is sometimes better to leave the processor idle until another actor is ready to fire. However, this compromise is common, even in fully static schedulers. When more than one actor is ready to be fired, the scheduler must determine which one to fire. A typical approach that is far from optimal is to randomly order the list of actors and apply a “fairness” principle, in which no actor will be tried twice before all other actors have been tried [Gao83]. Another alternative would be to use compile-time analysis of the dataflow graph to assign priorities to the actors. It seems to us that this idea should work much better in static-assignment scheduling than in fully dynamic scheduling as investigated by Granski, et. al. [Gra87], because the run-time implementation cost would be trivial.

2.2.4 Fully Dynamic Scheduling

Fully dynamic scheduling has the broadest application domain, since it can in principle subsume all functions in the previous models, and perform them at run time. Furthermore, it has the flexibility to redirect the computational load in response to changing conditions in the algorithm. However it requires too much hardware and/or software run-time overhead. For instance, the MIT static dataflow machine [Den80] proposes an expensive broadband packet switch for instruction delivery and scheduling. Furthermore, it is not usually practical to make globally optimal scheduling decisions at run-time. One attempt to do this by using static (compile-time) information to assign priorities to actors to assist a dynamic scheduler was rejected by Granski et. al., who concluded that there is not enough performance improvement to justify the cost of the technique [Gra87].

2.2.5 Summary

In order to reduce implementation costs and make it possible to reliably meet real-time constraints, the more scheduling that is done at compile time the better. Unfortunately, in order to automatically do more at compile time, it appears to be necessary to restrict the system to a narrower range of applications. Hence, static-assignment and self-timed scheduling strategies look like the most promising compromises between hardware cost, performance, and flexibility. The choice should depend on the amount of data-dependent behavior in the expected applications. Both strategies require compile-time decisions; they require that tasks be assigned to processors at compile time, and in addition, self-timed scheduling requires that the order of the execution of the tasks be specified. If there is no data-dependency in the application, then these decisions can be made optimally (or nearly so, to avoid complexity problems). When there is data-dependency, however, optimal or near optimal compile-time strategies become intractable. Most previously proposed solutions include random choices, clustering (to minimize communication overhead), and load balancing. These solutions either ignore precedence relationships in the dataflow graph, or use heuristics based on oversimplified stochastic models. This is justifiable if there is so much data-dependency that the precedence relationships are constantly changing. However, there is a large class of applications, including scientific computations and digital signal processing, where this is not true. Consequently, it appears that self-timed is more attractive for scientific computation and digital signal processing, while static-assignment is more attractive where there is more data dependency.

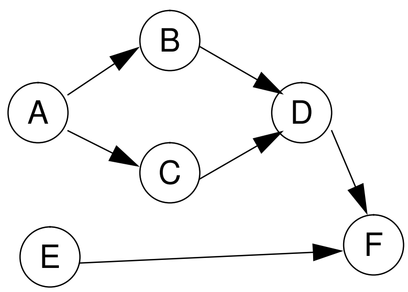
2.3. LIST SCHEDULING

List scheduling is the most common technique for non-overlapped scheduling. Even though our scheduling idea is not restricted to a specific technique, it is implemented via list scheduling. In list scheduling, actors are assigned priorities. During the scheduling process, the runnable actors are placed in a list sorted by their priorities. The actor of highest priority is assigned to the first available processor. The performance of a list schedule is determined by how the priorities are assigned to the actors. The most commonly used priority scheme is Hu's level scheduling [Hu61] or variants of Hu's technique [Ada74] [Cof76][Koh76]. Any actors lacking successors are attached to a common dummy actor and the priority of each actor is set equal to its level, defined as the largest sum of execution times on any directed path from the actor to the dummy actor.

The class of list schedules may not contain an optimal solution as is shown in figure 2.7. The list scheduling method does not allow a processor to remain idle as long as there are runnable actors, but the figure illustrates that it is sometimes necessary to idle processors to achieve an optimal schedule. An optimal list schedule has been proven, however, to be near optimal in terms of makespan, and at most twice as long as an optimal schedule [Koh76]. In some special cases, the optimal list schedule yields the optimal schedule.

2.4. DYNAMIC LEVEL SCHEDULING

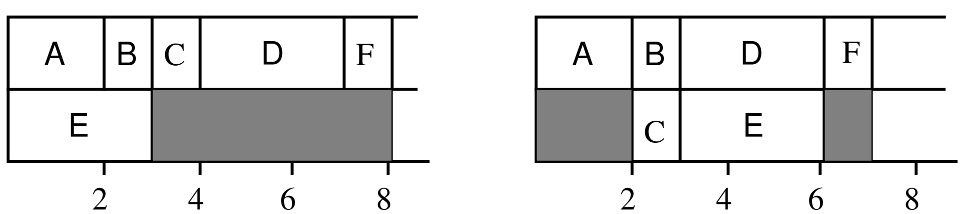
Classical list scheduling algorithms ignore the interprocessor communication (IPC) cost when assigning actors onto processors. Runnable actors are assigned to available processors to exploit as much of the parallelism of a program graph as possible. After the nodes are scheduled, traffic scheduling is performed to minimize the communication costs.



(a) a program graph

Node	Exec. time	Level
A	2	6
B	1	5
C	1	5
D	3	4
E	3	4
F	1	1

(b) execution times and levels of actors



(c) an optimal list schedule

(d) an optimal schedule

Figure 2.7 An example shows that an optimal list schedule is not necessarily an optimal schedule. (a) A program graph has two input nodes and one output node. (b) The execution times and levels of actors are displayed. (c) An optimal list schedule schedules node E at time zero on the second processor to result in a non-optimal makespan of 8 time units. (d) The optimal schedule has makespan of 7 time units and idles the second processor intentionally during the first two time units.

In reality, however, the IPC is not so small that it may be neglected at the scheduling phase. The effect of the IPC cost is demonstrated in figure 2.8. In the figure, after actor A is executed, actors B and C are both runnable. The list scheduling algorithm assigns them on two available processors to exploit the parallelism as shown in figure 2.8 (b). At run-time, however, communication overhead is involved between actors A and C, and actors C and D. If the IPC cost per unit data sample is small, the performance of the list schedule is not degraded (figure 2.8 (c)). If the IPC cost is not negligible as is illustrated in figure 2.8 (d), exploiting the parallelism by assigning actors to different processors is detrimental.

The IPC cost consists of the data-transmission time and the communication set-up

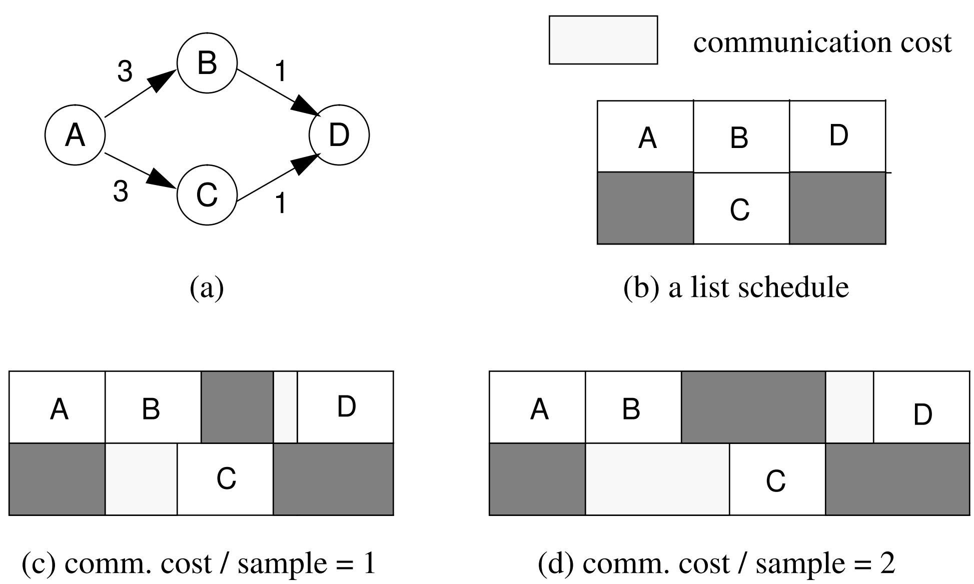


Figure 2.8 The effect of the interprocessor communication cost on a list scheduling algorithm based on Hu's level. (a) In a simple program graph, the number on each arc represents the number of data samples consumed or produced at the nodes attached to the arc. For simplicity, the execution time of the actors is 4 identically. (b) An optimal list schedule based on Hu's level. (c) When the communication overhead per each data sample is 1, the list schedule is optimal. (d) When the communication overhead per each data sample is 2, the list schedule is worse than assigning all actors on a single processor.

overhead. The latter gets costly as the sharing of communication resources increases. For example, if a system is a shared bus architecture, the IPC cost easily grows to a few tens of instructions: requesting the bus, grabbing the bus, sending the data, and releasing the bus. A new shared bus architecture that significantly reduces the IPC overhead has recently been proposed [Lee90]. In this approach, the bus access pattern is determined at compile-time. The bus controller arbitrates bus congestion according to the predetermined order, removing most of the IPC overhead from the processors.

Recently, G. Sih [Sih90b][Sih91] proposed a modified list scheduling algorithm, called the *dynamic level scheduling* (DLS) algorithm that takes into account resource limitation and communication overhead in a given network topology. He defines a dynamic level which incorporates resource constraints and network configuration, while Hu's level is static based on computation time only, without regard of the communication overhead. The priority of an actor is the sum of the static level ($SL(i)$) and its communication requirements with its ancestors as follows:

$$Level(i) = SL(i) + C_{adj}\left(\max_{k}\{D_{ki}\}\right) , \qquad (2\text{-}3)$$

where D_{ki} represents the number of data units passed from node k to node i, and $C_{adj}(D)$ denotes the time needed to communicate D data units between adjacent processors. The static level is the priority in classical list scheduling algorithms. The level in (2-3) is dynamic in the sense that it changes as the schedule is constructed.

The runnable actors are maintained in a list sorted by priority. The actor of the highest priority is fetched from the list and scheduled to an optimal processor. To choose the optimal processor, the actor is assumed to be scheduled on each processor one by one. The processor on which the actor would be scheduled earliest is the optimal processor. The scheduled time of the actor on processor *j* is expressed as:

$$T(j) = max\left\{ avail(j), \max_{k}(finishTime(k) + C'_{p(k)j}) \right\} \quad , \tag{2-4}$$

where *k* is an ancestor of the actor and *p(k)* is the processor assigned to the ancestor *k*. *avail(j)* is the earliest time when processor j is available, and *finishTime(k)* is the time when actor *k* is completed. After actor *k* is completed, data samples are passed from processor *p(k)* to processor *j*, taking $C'_{p(k)j}$ time units. In summary, the actor can be scheduled on processor *j* after all data samples are collected from the ancestors and when the processor is available. To compute $C'_{p(k)j}$, resource contention should be accounted for. We use a simple FIFO heuristic in which the processor that requests earlier gets service earlier. Thus, the communication resources are scheduled at the same time actors are assigned. For example, let's apply this technique to figure 2.8 (d). After assigning actors A and B, we should assign actor C. If actor C is assigned to processor *0*, *T(0)* in equation (2-4) becomes 8. On the other hand, if actor C is assigned to processor *1*, *T(1)* becomes 10. Therefore, the optimal processor for actor C is processor *0*, not processor *1*.

The DLS algorithm can be extended to heterogeneous multiprocessor systems. In this thesis, we assume a homogeneous multiprocessor system. The DLS algorithm assumes that communication can be overlapped with computation in a processor.

We modify the DLS algorithm to allow an actor to be scheduled on more than one processor. A dynamic construct is regarded as an atomic actor in the modified DLS algorithm, but it usually takes more than one processor on its execution. The number of the assigned processors to the dynamic construct is determined during the scheduling procedure. To account for the communication cost of the dynamic construct, we may define the *synchronization* processor at the beginning and at the end of the local schedule of the dynamic construct. All IPC requirements to the dynamic construct are via a synchronization processor.

2.5. SUMMARY

Most scheduling efforts have concentrated on deterministic scheduling because the scheduling problem is well defined and manageable at compile-time. However, most parallel applications contain some sort of data dependent behavior such as conditionals, data-dependent iterations, and recursions. For applications where data dependent behavior is not dominant, for example digital signal processing applications, resorting to runtime scheduling is not economical. In this thesis, we develop a scheduling technique to deal with such data-dependent behavior in the context of deterministic scheduling. This scheduling technique will remove the overhead of fully-dynamic scheduling while utilizing the mature techniques of deterministic scheduling. It will be applied to both the self-timed scheduling strategy and to the static assignment strategy that cover most current dataflow parallel processing systems.

3

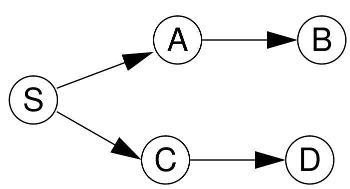
QUASI-STATIC SCHEDULING

Not that I speak in regard to need, for I have learned in whatever state I am to be content; I know how to be abased, and I know how to abound. Everywhere and in all things I have learned both to be full and to be hungry, both to abound and to suffer need.

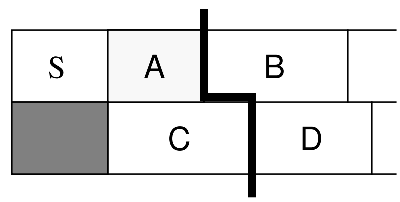
--- Philippians 4:11,12

Static scheduling is adequate for a rather restricted class of algorithms that can be described using the SDF model. However, certain essential constructs cannot be described using the SDF model, particularly conditional computation. Since such constructs are important even for DSP applications, the conventional solution is to reject static scheduling and incur the (substantial) cost of dynamic scheduling. But, dynamic scheduling is not required for most algorithms; thus a much simpler approach based on *quasi-static* scheduling is proposed. In quasi-static scheduling, most of the scheduling decisions are made at compile-time. Some scheduling decisions are made at runtime, but only when absolutely necessary. The quasi-static scheduling idea was proposed by E. Lee [Lee88].

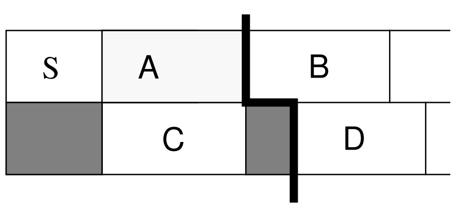
The idea of quasi-static scheduling is shown in figure 3.1. Suppose there is a non-deterministic actor A. This actor need not be atomic. At compile-time, we assume a fixed execution time for actor A. Based on that time, a deterministic schedule can be constructed for the whole graph as in (b). At run-time, the schedule is followed before the execution of actor A. When actor A is executed, it may take longer or shorter than the assumed execution time of compile-time. If actor A takes longer, some idle time is inserted on the other processor so that the pattern of processor availability after actor A is same as the compiled one. Otherwise, the processor that executed A is idled during the time difference between the assumed execution time and the actual execution time. By



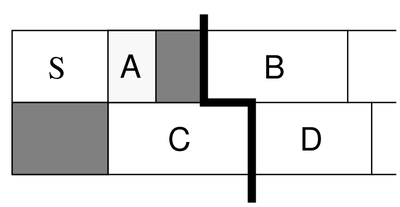
(a) a dataflow graph with non-deterministic actor A



(b) compile-time schedule



(c) runtime schedule when actor A takes longer than assumed.



(d) runtime schedule when actor A takes shorter than assumed.

Figure 3.1 (a) A dataflow graph consists of five actors among which actor A is a non-deterministic actor such as a conditional or a data-dependent iteration. (b) Gantt chart for compile-time scheduling assuming a certain execution time for actor A. (c) At runtime, if actor A takes longer, the second processor is padded with no-ops so that the pattern of processor availability after actor A is same as the compiled one. (d) If actor A takes less than the assumed time, the first processor is idled. The pattern of processor availability is shown as a dark line on the Gantt charts.

keeping the pattern of processor availability consistent with the compile-time schedule the remaining deterministic schedule after actor A can be followed.

The run-time behavior of actor A and the amount of idle time varies at run-time. This enforced idle time will be tolerable against the overhead of dynamic scheduling if the execution times of actors do not vary greatly. Signal processing algorithms generally support this model [Lee87a].

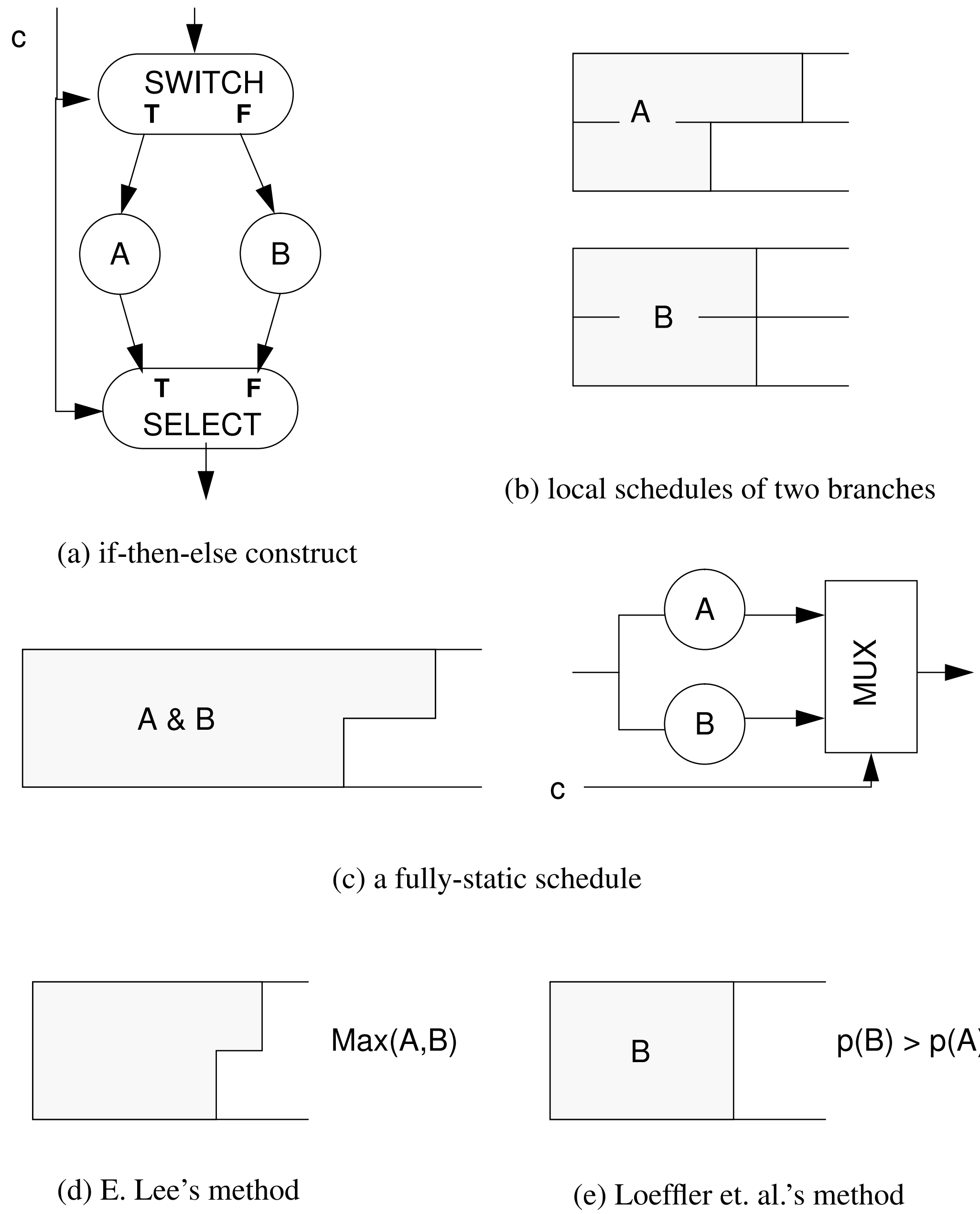
The most challenging problem of quasi-static scheduling is to determine the optimal compile-time profile of a non-deterministic actor. The *compile-time profile*, in short *profile*, of an actor is defined as the static information about the actor necessary for a given scheduling technique. For example, if we use the original Hu's level algorithm for list scheduling, the profile of an actor is simply the computation time of the actor. If we want to use G. Sih's dynamic level scheduling algorithm, we need information about the communication requirements of an actor as well as its computation time in order to comprise the profile of the actor. In case a non-deterministic actor is not an atomic actor but a subgraph that expresses a dynamic construct such as a conditional, a data-dependent iteration, or a recursion, we may need more than one processor to execute the non-deterministic actor. Then, the number of assigned processors and the local schedule of the actor on the assigned processors will be the profile of the actor. In this thesis, we concentrate on these dynamic constructs as the source of non-deterministic actors.

In this chapter, we first review the previous work related to the quasi-static scheduling technique on how to define the profiles of non-deterministic actors. In the next section, we introduce our solutions for profile decision and discuss their effectiveness. Later, we relate the quasi-static scheduling technique with the static-assignment and the self-timed scheduling strategies. Finally, we summarize the proposed scheduling technique.

3.1. PREVIOUS WORK

All the deterministic scheduling heuristics assume that static information about the actors is known. But none have addressed how to define the static information of non-deterministic actors. The pioneering work on this issue was done by Martin and Estrin [Mar69]. They calculated the mean path length for each actor based on the statistical distribution of dynamic behavior of non-deterministic actors for list scheduling. They define the mean path length between an actor and a dummy terminal actor of the acyclic precedence graph as the level of the actor. Since this is very expensive to compute, the mean execution times instead are usually used as the static information of non-deterministic actors [Gra87]. Even though the mean execution time seems a reasonable choice, it is by no means optimal. In addition, both approaches have the common drawback that a non-deterministic actor is manipulated as a schedulable unit, thus assigned to a single processor, which is a severe limitation for a multiprocessor system.

Two groups of researchers have proposed quasi-static scheduling techniques independently: E. Lee [Lee88] and Loeffler et. al. [Loe88]. They developed methods to schedule conditional constructs and data-dependent iteration constructs respectively. Both approaches allow more than one processor to be assigned to dynamic constructs. Figure 3.2 shows a conditional and compares three scheduling methods: Lee's, Loeffler's, and the replacement of the conditional with a multiplexer. In figure 3.2 (b), the local schedules of the two branches are shown with two processors in this example. In E. Lee's method, as shown in (d), we overlap the local schedules of both branches and choose the maximum termination for each processor. If both branches are about the same size, this scheme is very efficient. But, it is inefficient if either one branch is more likely to be taken and the size of the likely branch is much smaller. On the other hand, Loeffler et. al. first schedule the higher probability branch and the pattern of termination is observed. Then,



(a) if-then-else construct

(b) local schedules of two branches

(c) a fully-static schedule

(d) E. Lee's method

(e) Loeffler et. al.'s method

Figure 3.2 Three different schedules of a conditional construct, one for fully-static scheduling and two for quasi-static scheduling. (a) An example of a conditional construct that forms a non-deterministic actor as a whole. (b) Local deterministic schedules of the two branches. (c) A fully-static schedule executes both branches, which is equivalent to the modified graph at the right. (d) E. Lee overlaps the local schedules of both branches and chooses the maximum for each processor [Lee88]. (e) Loeffler et. al. take the local schedule of the branch which is more likely to be executed [Loe88] the lower probability branch is scheduled to overlap in time and padded with no-ops so that the pattern of termination is the same (figure 3.2 (e)). Finally, a conditional *evaluation* can be replaced by a conditional *assignment* to make the construct static which modifies the graph as illustrated in figure (c). In this scheme, both true and false branches are scheduled and the result from one branch is selected depending on the control boolean. An immediate drawback is inefficiency which becomes severe when a branch is not a small actor. Another severe problem occurs when an unselected branch generates an exemption condition such as divide-by-zero error. All these methods on conditionals are ad-hoc and not appropriate as a general solution.

Quasi-static scheduling is very effective for data-dependent iteration constructs if the construct can make effective use of all processors in each cycle of the iteration [Lee88] [Loe88]. In data-dependent iteration, the number of iteration cycles is determined at run time and cannot be known at compile time. In quasi-static scheduling technique, we schedule one iteration and pad with no-ops to make the pattern of processor availability at the termination the same as the pattern at the start (figure 3.3). The pattern of processor availability after the iteration construct is independent of the number of iteration cycles, which is consistent with the quasi-static scheduling requirement. This scheme breaks down if a construct can not utilize all of the processors effectively.

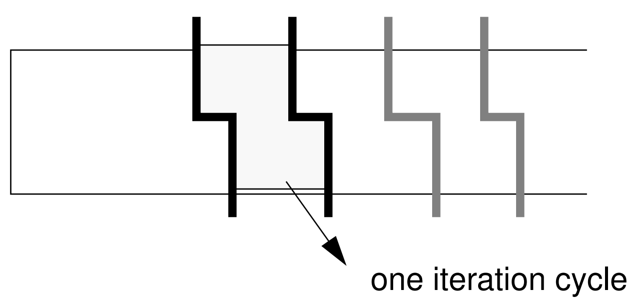


Figure 3.3 A quasi-static scheduling of a data-dependent iteration construct. The pattern of processor availability is independent of the number of iteration cycles.

The recursion construct has not yet been treated successfully in any statically scheduled data flow paradigm. Recently, a proper representation of the recursion construct has been proposed [Suh90]. But it does not explain how to schedule the recursion construct onto multiprocessors. With finite resources, careless exploitation of the parallelism of the recursion construct may cause the system to deadlock. Resource management in this case has been an open problem in the area of data flow computation.

In summary, dynamic constructs such as conditionals, data-dependent iterations, and recursions, have not been treated properly in past scheduling efforts, either for static scheduling or dynamic scheduling. Some ad-hoc methods have been introduced but proved unsuitable for a general solution. This research is motivated by this observation.

3.2. PROPOSED SCHEME FOR PROFILE DECISION

Dynamic constructs result in variations on the makespan of a program. If we assume that all dynamic constructs are decoupled, we may isolate the effect of each dynamic construct on the makespan separately. Suppose we have a dynamic construct in a system. The profile of the dynamic construct is assumed at compile-time. If a quasi-static scheduling strategy is applied to the system, the average makespan of the system depends on the assumed profile of the dynamic construct. Our proposed scheme is to decide the profile of a construct so that the average makespan is minimized assuming that all actors except the dynamic construct are deterministic. This objective is not suitable for a hard real-time system as it does not bound the worst case behavior.

The *run-time cost* of an actor i (C_i) is the sum of the total computation time devoted to the actor and the idle time due to the quasi-static scheduling strategy over all processors. In figure 3.1, the run-time cost of a non-deterministic actor A is the sum of the lightly (computation time) and darkly shaded areas after actor A or C (immediate idle time after the dynamic construct). The makespan for a certain iteration can be written as

$$makespan = \frac{1}{T}(C_i + R) \tag{3-1}$$

where T is the total number of processors in the system, and R is the rest of the computation including all idle time that may result both within the schedule and at the end. R is determined at compile-time and fixed at run-time. Therefore, we can minimize the expected makespan by minimizing the expected cost of the non-deterministic actor A if we assume that R is independent of the decision of the profile of an actor. This assumption is unreasonable when precedence constraints make R heavily dependent on our choice of profile. If there are many runnable actors at any time compared to the number of processors and the execution times of all actors are small relative to the makespan, the assumption is valid. Realistic situations are likely to fall between these extremes.

3.2.1 Assumptions

The proposed scheme selects the compile-time profile of each dynamic construct to minimize the expected run-time cost. The computation of the run-time cost of a dynamic construct is based on the following assumptions.

1. Quasi-static scheduling is used. We have to insert idle times on the processors after the dynamic construct if it behaves differently at run-time from the assumed profile. These idle times are included in the run-time cost of the dynamic construct.
2. The statistical distribution of the run-time behavior of the dynamic construct is known at compile-time. The validity of this assumption varies to large extent depending on the application. In signal processing applications where a given program is repeatedly executed with an input data stream, simulation may give the necessary information. In general, we prefer to use some well-known distribu-

tions, such as uniform or geometric, which have nice mathematical properties.

3. If an application contains more than one dynamic construct, their dynamic behaviors are independent of each other.
4. The program graph is highly parallelizable. The previous assumption and this one are necessary to make the proposed scheme effective.
5. When we construct a local schedule, we assume that all assigned processors are available at the same time, that is, that the pattern of processor availability is flat.

The last three assumptions are necessary to make the analysis tractable or the proposed scheme effective. The last assumption is needed when the body of a dynamic construct contains another dynamic construct (nested dynamic constructs case).

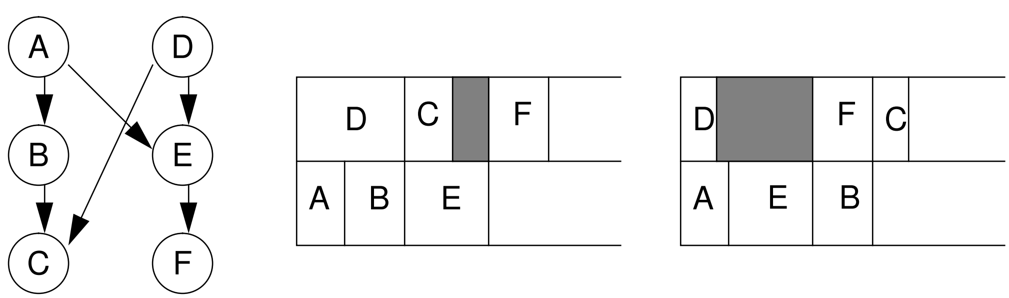
3.3. STATIC ASSIGNMENT AND SELF-TIMED SCHEDULING

Once the profiles of the dynamic constructs of a program have been decided, we can construct a static schedule accordingly. Quasi-static scheduling means global synchronization that makes the pattern of processor availability after a dynamic construct consistent with the scheduled one. This requires hardware for global synchronization, which may be less expensive than the handshaking required for self-timed execution (a simple wired-or circuit would suffice). However, inserting idle time may be unnecessary in reality. Furthermore, if handshaking is omitted, then the system is intolerant of runtime fluctuations, due for example to interrupts or I/O operations. Hence the quasi-static scheduling strategy is regarded as impractical. Nonetheless, it suggests a good strategy for static-assignment or self-timed scheduling. First we construct a quasi-static schedule. To get self-timed execution, we insert handshaking at runtime, and ignore the firing times dictated by the quasi-static schedule. To get static-assignment execution, we discard all information from the quasi-static schedule except the assignment of actors to processors.

3.3.1 Static Assignment Scheduling

In static-assignment scheduling, actors are assigned to processors without defining the execution order. Unlike dynamic load balancing or techniques that compromise between interprocessor communication cost and load balance, the quasi-static scheduling strategy considers arbitrary precedence relations at compile time. If the actual computation times are similar to those assumed by the compile-time scheduler, then it can get close to the minimal makespan.

An example of static-assignment scheduling is shown in figure 3.4. A dataflow program consists of six actors with precedence relationships shown in (a). Actor D represents a dynamic construct, or a non-deterministic actor. Suppose that the program is statically scheduled using a certain compile-time profile of actor D, and the resulting assignment puts actors D, C, and F onto the first processor, and the rest onto the second processor. The ordering and timing information is discarded. Assuming D has a data-dependent execution time, the run-time schedule depends on its outcome. Two possible schedules are shown in figure 3.4 (b) and (c). By inspection, we can see in the figure that



(a) precedence relation (b) (c)

Figure 3.4 An example of static-assignment scheduling. The precedence relations are shown in (a), and two possible schedules, which depend on the execution time of actor D, are shown in (b) and (c).

the schedules shown are optimal in the sense of minimizing makespan, given assignment of the actors. However, designing a run-time scheduler that reliably produces these schedules is not easy. Assume that when a processor becomes free, if there is an actor ready to be fired, then the run-time scheduler will fire it. This is not necessarily optimal, but in deterministic processor scheduling it can be shown to be reasonable. Then the only decision to be made by the scheduler occurs when there is more than one actor ready to fire. In figure 3.4 (b), the run-time scheduler never faces this decision, so a very simple strategy will yield the schedule shown. In (c), however, after the completion of actor A, the second processor must decide between firing B or E. E is the better choice, but it is not clear at all how the scheduler might know this. An immediate idea is to use some of the static information that was discarded: specifically the ordering information. However, this does not guarantee the right choice, because the static information is based on an assumption about the data-dependent execution time, and the outcome may be far from this. The alternative of stochastic modeling of the program is not very promising either, because only the most grossly oversimplified stochastic models yield to optimization.

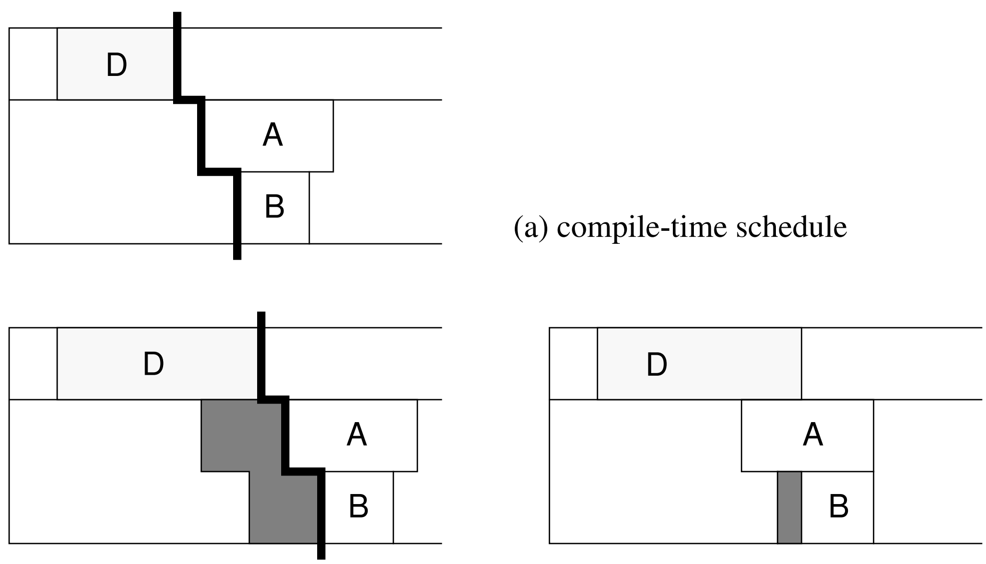
The above observations lead to an interesting conclusion. In static-assignment scheduling, the run-time scheduler on each processor faces an ambiguous decision only if more than one of the actors assigned to it are ready to fire when the last actor completes. If this situation arises rarely, then a naive scheduler will work well. However, under the same conditions, a self-timed strategy would work just as well, and the cost would be lower. On the other hand, if the situation arises frequently, then we do not know how to make the decision. Practical proposals are to make the decision arbitrarily, subject to a "fairness" principle, in which no actor will be tried twice before all other actors have been tried [Gao83]. It may be profitable to augment this strategy by using information discarded from the static schedule, but as argued before, this is not guaranteed to lead to an optimal schedule.

A comparison with the Granski, et. al. proposal [Gra87] is in order. In fully dynamic scheduling, assignment is easy, assuming the target architecture is homogeneous. It does not matter which free processor gets an actor, once the decision has been made to fire that actor. So the decisions to be made by the scheduler are simply which of the actors that are ready to be fired should be fired. If the number of actors that are ready to be fired is smaller than the number of available processors, then there is no decision to be made, and the scheduler will not be helped by static information. It is only if the number of ready actors is large that static information can help. In [Gra87] the authors report that the improvement due to using static information in a dynamic scheduler degrades to no improvement for large numbers of processors. We just stated the reason for this.

3.3.2 Self-Timed Scheduling

In self-timed scheduling, we define the execution order of actors at compile time, thus avoiding the difficulty of designing the local controller. In the example of figure 3.4, suppose that actors are constrained to execute in the order given by figure 3.4 (b). In this case, we sacrifice some freedom to optimize at execution time. However, if the variability in execution time is small enough, then there is little justification for paying the run time cost of static-assignment scheduling. Of course, if the explicit token store mechanism of Papadopoulos [Pap88] proves to be truly low cost, then the additional adaptability of static-assignment scheduling makes it more attractive. As pointed out earlier, however, tractable static-assignment scheduling is *not* guaranteed to outperform self-timed. It is easy to construct demonstration examples where, for example, an iteration finishes well before expected, causing an order change that results in a *larger* makespan than if there were no order change.

The difference between quasi-static and self-timed scheduling is shown in figure 3.5. In quasi-static scheduling, actors A, B are executed after actor D even if actor A is independent of the dynamic construct D, assuming the scheduler places A after the assumed end of the dynamic construct. We also have to synchronize the processors of actors by inserting idle time compulsorily. However, in self-timed scheduling, actor A is executed independently of the completion of actor D when its data are available. Idle time may be automatically inserted after A while the next actor waits for data. Similarly, actor B is executed as soon as it is runnable; that is, as soon as all input data are available and the assigned processor is available. Since all actors are executed before or at the same time as in the quasi-static scheduling case, the self-timed scheduling strategy always gives a result better than or equal to the quasi-static scheduling strategy, assuming that the overhead for synchronization is comparable. In addition, self-timed scheduling does not need a global synchronization, but only local handshaking. As a result, we believe that



(a) compile-time schedule

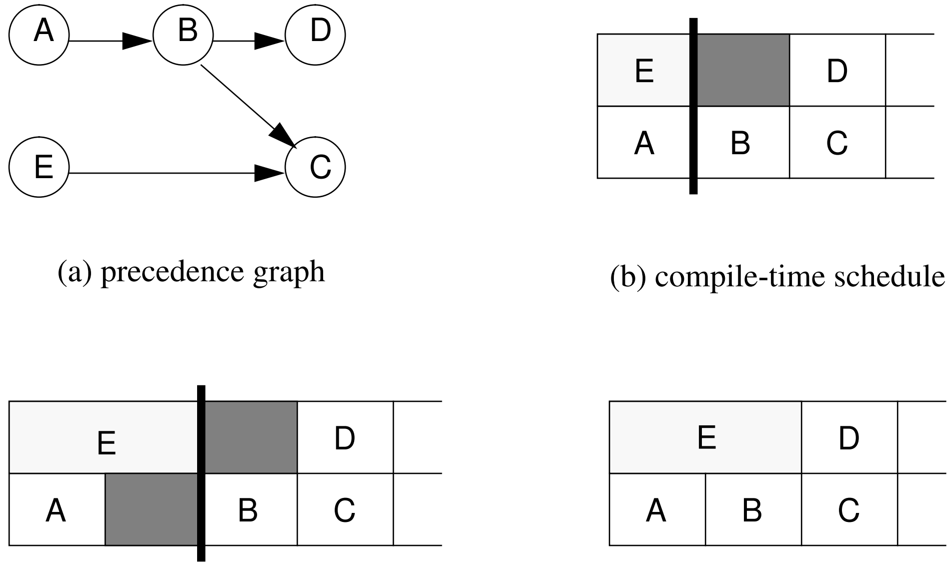
(b) quasi-static scheduling

(c) self-timed scheduling

Figure 3.5 Comparison between quasi-static scheduling and self-timed scheduling. In quasi-static scheduling, the pattern of processor availability after the dynamic construct is enforced by global synchronization. In self-timed scheduling, the pattern is only enforced if the precedences require it. Here we have assumed that actor B is dependent on the dynamic construct D, but actor A is not.

self-timed scheduling is more attractive.

Self-timed scheduling overcomes a difficulty of quasi-static scheduling illustrated in figure 3.6. In the precedence graph shown in figure 3.6 (a), assume the non-deterministic actor E is equally likely to run for 0, 1, or 2 time units and one of two processors is to be devoted to the dynamic construct. Then our proposed strategy yields the quasi-static schedule in (b). However, suppose the actual execution time exceeds the assumed value. A strict quasi-static schedule, in which global synchronization enforces the pattern of processor availability after the iteration, would execute as shown in figure 3.6 (c), while a self-timed schedule would execute as shown in (d). In this case, our proposed schedule is no more optimal than that in (d), because we consider only the idle time before the com-



(a) precedence graph (b) compile-time schedule

(c) quasi-static schedule (d) self-timed schedule

Figure 3.6 An example showing that a difficulty in quasi-static scheduling is overcome in self-timed scheduling. Actor E is a non-deterministic actor. According to the precedence graph in (a), a quasi-static schedule is shown in (b). Assume now that the actual execution time of actor E is two. Quasi-static execution of the schedule results in the schedule shown in (c), while self-timed execution results in the schedule shown in (d).

pletion of the non-deterministic actor when assessing the run-time cost of actor E. In other words, our choice of the profile of actor E is only locally optimal. In this example, the idle time after the dynamic construct varies at run-time. Self-timed execution can sometimes compensate for this deficiency in the quasi-static scheduling strategy. Idle time immediately after the completion of the dynamic construct has no effect on the performance since there is no compulsory idle time. In other words, for self-timed execution, the schedules in (b) and (d) are equivalent.

This does not lead us to the conclusion that the quasi-static scheduling strategy we propose is optimal under self-timed execution. Consider the two schedules in figure 3.7, which assume the same precedence graph from figure 3.6. Under self-timed scheduling, the schedule in figure 3.7 (a) is clearly preferable to that in (b), because even if the dynamic construct runs twice as long as the assumed execution time, the makespan will not be affected. Our scheduling strategy thus far imposes no constraints that would prefer the schedule in figure 3.7 (a). Intuitively, care should be taken to schedule actors after the non-deterministic actor in static-assignment or self-timed scheduling. For examples of this type, the problem can be largely avoided by the following heuristic; all else being equal, actors independent of the non-deterministic actor should not be assigned to the processors executing the dynamic construct. This heuristic may be easily incorporated in

(a) optimal schedule

(b) bad schedule

Figure 3.7 For the same precedence graph as in figure 3.6, two static schedules with the same makespan are shown. However, if the actual number of iterations turns out to be two, the schedule in (a) is better than that in (b).

the original static scheduling without significant cost.

3.4. PROPOSED TECHNIQUE

Summing up the discussions of the previous sections, the proposed scheduling technique consists of three steps.

1. Determine the profile of each dynamic construct. The profile of a dynamic construct consists of the number of processors assigned to the construct and the assumed execution times of the construct on the assigned processors (local schedules).
2. Compute a schedule using a deterministic scheduling technique. Even if we use a list scheduling scheme based on the original Hu's level or the dynamic level scheduling algorithm by Gil Sih, other heuristics can be applied.
3. Apply the obtained schedule to the self-timed scheduling or the static assignment scheduling strategy. To achieve self-timed execution, discard the timing information from the static schedule, but retain both the assignment and the ordering of actors. For static-assignment scheduling, retain only the assignment of actors to processors.

In the rest of this thesis, we focus on the first step, deciding the optimal profiles of well-known dynamic constructs: conditionals, data-dependent iterations, and recursions. Our scheduling strategy in the second step is different from most of the existing deterministic scheduling techniques in that certain actors (non-deterministic actors) may need to be assigned to more than one processor. Pioneering work in this direction has been done by J. Blazewics et. al. [Bla86]. They aim to minimize the schedule length as a non-overlap execution schedule, allowing certain tasks to take more than one processor at a time for their processing. They examined both non-preemptive and preemptive scheduling techniques, but ignored the precedence relations of actors as well as communication overhead. The requirement of assigning more than one processor to an actor complicates the scheduling problem further.

4

PROFILE DECISIONS

Well done, good and faithful servant; you have been faithful over a few things, I will make you ruler over many things

--- Matthew 25:21

The compile-time profile of a nondeterministic actor, or a dynamic construct, consists of the number of assigned processors and the local schedule on the assigned processors. The profile is chosen to minimize the runtime cost of the actor, assuming quasi-static scheduling. In this chapter, we illustrate how the proposed technique determines the optimal profiles of three well-known types for dynamic constructs: data-dependent iterations, conditionals, and recursions.

4.1. DATA-DEPENDENT ITERATION

In a data-dependent iteration, the number of iteration cycles is determined at runtime and cannot be known at compile-time. Two possible dataflow representations for data-dependent iteration are shown in figure 4.1. In figure 4.1 (a), since the upsample actor produces *x* tokens each time it fires, and the iteration body consumes only one token when it fires, the iteration body must fire *x* times for each firing of the upsample actor. In figure 4.1 (b), the number of iterations need not be known prior to the commencement of the iteration. Here, a token coming in from above is routed through a SELECT actor into the iteration body. The "D" on the arc connected to the control input of the SELECT actor indicates an initial token on that arc with value "false". This ensures that the data coming into the "F" input will be consumed the first time the SELECT actor fires. After this first input token is consumed, the control input to the SELECT actor will have value "true" until the function t(x) indicates that the iteration is finished by producing a token with value "false". During the iteration, the output of the iteration function f(x) will be

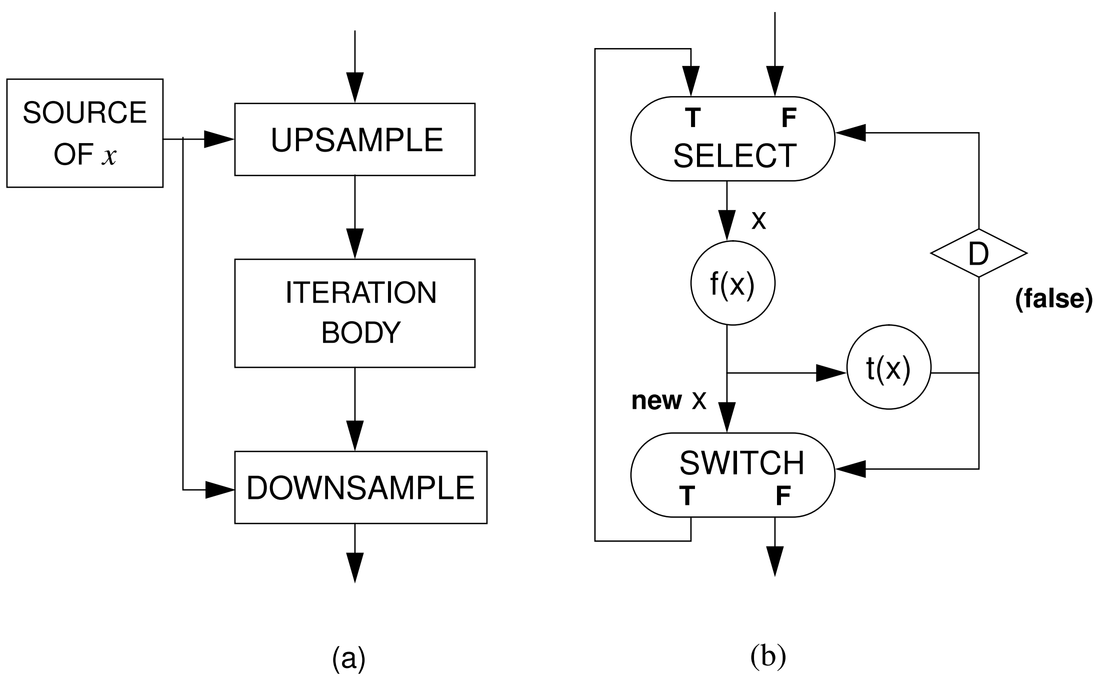


Figure 4.1 Data-dependent iteration can be represented using the either of the data-flow graphs shown. The graph in (a) is used when the number of iterations is known prior to the commencement of the iteration, and (b) is used otherwise.

routed around by the SWITCH actor, again until the test function t(x) produces a token with value "false". There are many variations on these two basic models for data-dependent iteration.

For simplicity, we will group the body of a data-dependent iteration into one node, and call it a data-dependent iteration actor. In other words, we assume a hierarchical dataflow graph. In figure 4.1 (a), the "iteration body" actor consists of the upsample, data-dependent iteration, and downsample actors. The data-dependent iteration actor may consist of a sub-graph of arbitrary complexity, and may itself contain data-dependent iterations. In figure 4.1 (b), everything between the SELECT and the SWITCH, inclusive, is the data dependent iteration actor. In both cases, the data-dependent iteration actor can be viewed as an actor with a stochastic runtime, but unlike atomic actors, it can be scheduled onto several processors. Although our proposed strategy can handle multiple and nested iteration, for simplicity all our examples will have only one iteration actor in the dataflow graph.

The method given in this thesis can be applied to both kinds of iteration in figure 4.1 identically. There is, however, an important difference between them. In figure 4.1 (b), each cycle of the iteration depends on the previous cycle. There is a recurrence that prevents simultaneous execution of successive cycles. In figure 4.1 (a), there is no such restriction, unless the iteration body itself contains a recurrence.

The proposed scheme has two components. First, the compiler must determine which processors to allocate to the data-dependent iteration actor. These will be called the *iteration* processors, and the rest will be called *non-iteration* processors. Second, the data-dependent iteration actor is optimally assigned an *assumed execution time* to be used by the scheduler. In other words, although its runtime will actually be random, the scheduler will assume a carefully chosen deterministic runtime and construct the schedule accordingly. The assumed runtime is chosen so that the expected total idle time due to the difference between the assumed and actual runtimes is minimal; the expected runtime cost of the actor is thus minimized. Locally minimizing idle time is well known to fail to minimize the expected makespan, except in certain special cases. We will discuss these special cases and argue that the strategy is nonetheless promising, particularly when combined with other heuristics.

The assumed execution time and the number of processors devoted to the iteration together give the scheduler the information it needs to schedule all actors around the data-dependent iteration. It does not address, however, how to schedule the data-dependent iteration itself. We will not concentrate on this issue because it is the standard problem of statically scheduling a periodic dataflow graph onto a set of processors [Lee87a]. Nonetheless, it is worth mentioning techniques that can be used. To reduce the computational complexity of scheduling and to allow any number of nested iterations without difficulty, blocked scheduling can be used. In these techniques, several cycles of an iteration are executed in parallel to increase the overall throughput. The proposal applies regardless of which method is used, but in all our illustrations we assume blocked scheduling. We similarly avoid specifics about how the scheduling of the overall dataflow graph is performed. Our method is consistent with simple heuristic scheduling algorithms, such as Hu-level scheduling [Hu61], as well as more elaborate methods that attempt, for example, to reduce interprocessor communication costs. Broadly, our method can be used to extend any deterministic scheduling algorithm (based on execution times of actors) to include data-dependent iteration.

4.1.1 Expected Runtime Cost

We assume that the probability distribution of the number of cycles of the iteration actor is known or can be approximated at compile time. Let the number of iteration cycles be a random variable I with known probability mass function $p(i)$. Denote the minimum possible value of I by MIN and the maximum by MAX. MAX need not be finite. Suppose that we allocate N processors to the data-dependent iteration actor. If the total number of the processors is T, the number of non-iteration processors is T-N.

The local schedule of one iteration cycle of the data-dependent iteration actor is depicted in figure 4.2. The processors are synchronized at the beginning and at the end of the schedule for blocked scheduling. The makespan of one iteration cycle is τ_N. The next iteration cycle is assumed invocable after t_N. Given the local schedule of one iteration cycle, we decide the assumed number of iteration cycles, x_N, and the number of overlapped cycles, k_N. For the time being we restrict x_N to integers. Once the two parameters, x_N and k_N, are chosen, the profile of the data-dependent iteration actor is determined as shown in figure 4.3 (a). The subscript N of τ_N, t_N, x_N, and k_N represents that they are functions of N, the number of iteration processors. For brevity, we will omit the subscript N for the variables throughout this chapter.

At runtime, for each invocation of the iteration actor, there are three possible outcomes: the actual number i of cycles of the iteration is (1) equal to, (2) greater than, or (3) less than x. These cases are displayed in figure 4.3. Note that the pattern of processor availability after the iteration is strictly enforced according to the quasi-static scheduling

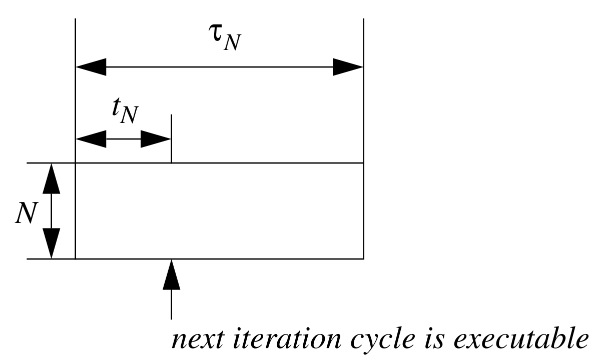


Figure 4.2 A blocked schedule of one iteration cycle of a data-dependent iteration actor.

discipline. Consider the case where the assumed number x is exactly correct. Then no idle time exists on any processor (figure 4.3 (a)). Otherwise, some of the iteration processors will be idled if the iteration takes fewer than x cycles (figure 4.3 (b)), or else the non-iteration processors as well will be idled (figure 4.3 (c)). The runtime cost of the iteration actor for each iteration cycle is the sum of the execution times (dotted area in figure 4.3) on the iteration processors and the idle times (dark area in the figure) due to the iteration. For $i \le x$, the runtime cost becomes $N\tau x$. For $i > x$, it becomes $N\tau x + T\tau\left\lceil \frac{i-x}{k} \right\rceil$. Therefore, the expected cost of the iteration actor *C(N,k,x)*, which is a weighted sum of the

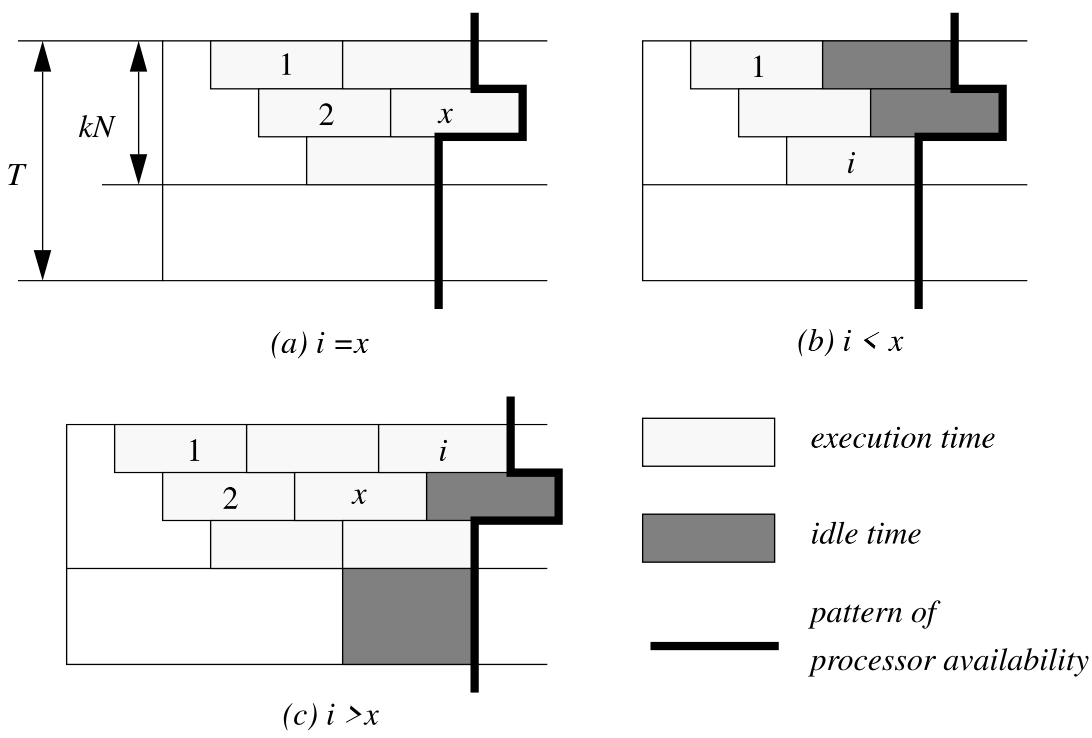


Figure 4.3 A quasi-static schedule is constructed using a fixed assumed number x of cycles in the iteration. The runtime cost of the actor is the sum of the dotted area (execution time) and the dark area (idle time due to the iteration). The idle time due to the difference between x and the actual number of cycles i is shown for 3 cases: i is equal to, less than, or greater than the assumed number, x.

runtime cost by the probability mass of the number of iteration cycles, becomes

$$C(N, k, x) = \sum_{i = MIN}^{x} p(i)N\tau x + \sum_{i = x+1}^{MAX} p(i)\left(N\tau x + T\tau\left\lceil \frac{i-x}{k} \right\rceil\right) \tag{4-1}$$

By combining the first term with the first element of the second term, we reduce it to

$$C(N, k, x) = N\tau x + T\tau \sum_{i = x+1}^{MAX} p(i)\left\lceil \frac{i-x}{k} \right\rceil \quad . \tag{4-2}$$

4.1.2 Assumed Execution Time

Our method is to choose three parameters, *N, x*, and *k*, in order to minimize the expected cost in equation (4-2). First, we assume that *N* is fixed. How to determine the optimal value for *N* will be explained in the following section. Since *C(N,k,x)* is a non-increasing function of *k* with fixed *N*, we can determine the parameter *k*. The parameter k is bounded by two ratios: $\frac{T}{N}$, and $\frac{\tau}{t}$. The latter constraint is necessary to avoid any idle time between iteration cycles on a processor. Interdependency between cycles is already accounted for in *t*. As a result, *k* is set to be

$$k = min\left(\left\lfloor \frac{T}{N} \right\rfloor, \left\lfloor \frac{\tau}{t} \right\rfloor\right) \quad . \tag{4-3}$$

The next step is to determine the optimal *x*. If a value *x* is optimal, the expected cost *C(N,k,x)* is not decreased if we vary *x* by *1* or *-1*. Therefore, we obtain the following inequalities,

$$C(N,k,x) - C(N,k,x+1) = -N\tau + T\tau \sum_{i=0}^{\infty} p(x+1+ik) \le 0$$
$$C(N,k,x) - C(N,k,x-1) = N\tau - T\tau \sum_{i=0}^{\infty} p(x+ik) \le 0 \quad . \tag{4-4}$$

In these inequalities, we assume that *MAX* is infinite, which is equivalent to defining that *p(i)* is zero for *i* greater than *MAX*. Since τ is positive, from inequality (4-4),

$$\sum_{i=0}^{\infty} p(x+1+ik) \le \frac{N}{T} \le \sum_{i=0}^{\infty} p(x+ik) \quad . \tag{4-5}$$

Suppose that *k* is equal to *1*. The inequality (4-5) becomes

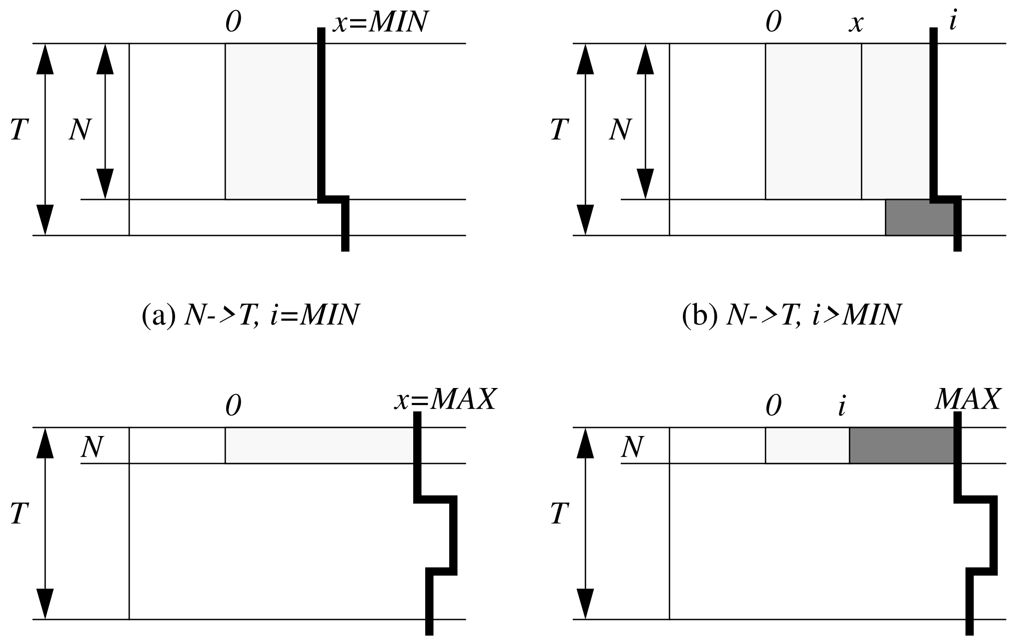
$$\sum_{i=x+1}^{\infty} p(i) \le \frac{N}{T} \le \sum_{i=x}^{\infty} p(i) \quad . \tag{4-6}$$

All quantities in this inequality are between *0* and *1*. The left and right sides are non-increasing functions of *x*. Furthermore, for all possible *x*, the intervals

$$\left[\sum_{i=x+1}^{\infty} p(i), \sum_{i=x}^{\infty} p(i) \right] \tag{4-7}$$

are non-overlapping and cover the interval [*0,1*]. Hence, either there is exactly one integer *x* for which *N/T* falls in the interval, or *N/T* falls on the boundary between two intervals. Consequently, inequality (4-6) uniquely defines the one optimal value for *x*, or two adjacent optimal values.

This choice of *x* is intuitive. As the number of iteration processors approaches the total number, *T*, of processors, *N/T* goes to *1* and *x* tends towards *MIN*. Thus even if an iteration finishes unexpectedly early, the iteration processors will not be idled. Instead the non-iteration processors (if there are any) will be idled (figure 4.4 (a) and (b)). On the other hand, *x* will be close to *MAX* if *N* is small. In this case, unless the iteration runs



(a) *N->T, i=MIN* (b) *N->T, i>MIN*

(c) *N=1, i=MAX* (d) *N=1, i<MAX*

Figure 4.4 When the number of the iteration processors *N* approaches the total number, *T*, *x* approaches *MIN* and the iteration processors will not be idled for any actual number of iterations (a and b). On the other hand, when *N* is small, *x* tends toward *MAX* so that the non-iteration processors will not be idled (c and d).

through nearly *MAX* cycles, the iteration processors, of which there are few, will be idled while the non-iteration processors need not be idled (figure 4.4 (c) and (d)). In both cases, the processors that are more likely to be idled at runtime are the lesser of the iteration or non-iteration processors.

Consider the special case that $N/T = 1/2$. Then from equation (4-6),

$$\sum_{i=x+1}^{\infty} p(i) = 1 - \sum_{i=MIN}^{x} p(i) \le \frac{1}{2} \tag{4-8}$$

which implies that

$$\sum_{i=MIN}^{x} p(i) \ge \frac{1}{2} \quad . \tag{4-9}$$

Furthermore,

$$\sum_{i=x}^{\infty} p(i) \geq \frac{1}{2} \quad . \tag{4-10}$$

Taken together, equation (4-9) and equation (4-10) imply that the best choice for x is the *median* of the random variable I (not the mean, as one might expect). In retrospect, this result is obvious because for any random variable I, the value of x that minimizes $E|I-x|$ is the median. Note that for a discrete-valued random variable, the median is not always uniquely defined, in that there can be two equally good candidate values. This is precisely the situation where x falls on the boundary between two intervals (4-7).

For k greater than *1*, inequality (4-4) is not sufficient in general for deciding the optimal x because the intervals, $[\sum_{i=0}^{\infty} p(x+1+ik), \sum_{i=0}^{\infty} p(x+ik)]$, involved in inequality (4-5) are overlapping if the summation $\sum_{i=0}^{\infty} p(x+ik)$ is not monotonic or constant over x. Therefore, we split the search space of x into k subspaces. Each subspace is mapped to an element of a modulo k group. In other words, x belongs to the i-th subspace if x modulo k is equal to i. We compute the optimal x_i in the i-th subspace; the expected cost $C(N,k,x_i)$ is not decreased if x_i is varied by $+k$ or $-k$.

$$\begin{aligned} C(N,k,x_i) - C(N,k,x_i+k) &= -N\tau k + T\tau \sum_{j=0}^{\infty} p(x_i+1+j) \leq 0 \\ C(N,k,x_i) - C(N,k,x_i-k) &= N\tau k - T\tau \sum_{j=0}^{\infty} p(x_i-k+1+j) \leq 0 \end{aligned} \quad . \tag{4-11}$$

From inequality (4-11),

$$\sum_{j=0}^{\infty} p(x_i + 1 + j) \le \frac{Nk}{T} \le \sum_{j=0}^{\infty} p(x_i - k + 1 + j) \tag{4-12}$$

All quantities in this inequality are between *0* and *1*. The left and right sides are non-increasing functions of x_i. Furthermore, for all possible x_i in the *i*-th subspace, the intervals

$$\left[\sum_{j=0}^{\infty} p(x_i + 1 + j), \sum_{j=0}^{\infty} p(x_i - k + 1 + j)\right] \tag{4-13}$$

are non-overlapping. Hence, either there is exactly one integer x_i for which *Nk/T* falls in the interval, or *Nk/T* falls on the boundary between two intervals. Consequently, inequality (4-12) uniquely defines the one optimal value for x_i, or two adjacent optimal values. This discussion is parallel to that for the case when $k = 1$.

After computing x_i's from all subspaces, we calculate the optimal x by comparing all the expected costs $C(N,k,x_i)$'s. Or, we select a subset of $\{x_i\}$ by applying inequality (4-5) and compare the expected costs for the subset. Note that $\{x_i\}$ are consecutive numbers, as can be seen from inequality (4-12). When k is equal to *1*, we have only one x_i, which is the optimal x. The inequality (4-12) is reduced to inequality (4-6).

Up to now, we have implicitly assumed that the optimal x is an integer, corresponding to an integer number of cycles of the iteration. We state it as a theorem.

Theorem 4.1: *The optimal value of the assumed number of iteration cycles for a data-dependent iteration construct is an integer.*

Proof:

For non-integer x, the total expected cost is restated as

$$C(N, k, x) = \sum_{i = MIN}^{\lfloor x \rfloor} p(i)N\tau x + \sum_{i = \lfloor x \rfloor + 1}^{MAX} p(i)\left(N\tau x + T\tau\left\lceil \frac{i-x}{k} \right\rceil\right) \quad . \tag{4-14}$$

Defining $\delta_x = x - \lfloor x \rfloor$, so $0 \le \delta_x < 1$. Then equation (4-14) becomes

$$\begin{aligned} C(x) &= \sum_{MIN}^{\lfloor x \rfloor} p(i)N\tau(\lfloor x \rfloor + \delta_x) + \sum_{\lfloor x \rfloor + 1}^{MAX} p(i)\left(N\tau(\lfloor x \rfloor + \delta_x) + T\tau\left\lceil \frac{i - \lfloor x \rfloor}{k} \right\rceil\right) \\ &= C(\lfloor x \rfloor) + \sum_{i = MIN}^{\infty} p(i)N\tau\delta_x \end{aligned} \tag{4-15}$$

This tells us that between $\lfloor x \rfloor$ and $\lfloor x \rfloor + 1$, *C(x)* is an affine function of δ_x , so it must have its minimum at $\delta_x = 0$ because the slope is positive. This proves that the optimal value of *x* is an integer. Q.E.D.

The exact form of the probability distribution is usually not available. Instead, we approximate it by a certain well-known distribution, such as a uniform distribution or a geometric distribution, that has some useful analytical properties; for example, the inequality (4-5) is a sufficient condition for the optimal *x* because each summation term is monotonic in *x*. As a result, we can obtain a closed formula for the optimal values of *x*.

Uniform Distribution

Suppose that *p(i)* is a uniform distribution over the range *MIN* and *MAX*. In other words,

$$p(i) = \begin{cases} \dfrac{1}{MAX - MIN + 1} & MIN \le i \le MAX \\ 0 & otherwise \end{cases} \quad \cdot \tag{4-16}$$

The second summation term in inequality (4-5) becomes

$$\sum_{i=0}^{MAX} p(x+ik) = \frac{1}{MAX - MIN + 1}\left\lfloor \frac{MAX - x}{k} + 1 \right\rfloor \quad . \tag{4-17}$$

Therefore, from inequality (4-5),

$$\begin{gathered} \left\lfloor \frac{MAX - x - 1}{k} + 1 \right\rfloor \le \frac{N}{T}(MAX - MIN + 1) \le \left\lfloor \frac{MAX - x}{k} + 1 \right\rfloor \\ \left\lfloor \frac{MAX - x - 1}{k} \right\rfloor \le \frac{N}{T}(MAX - MIN + 1) - 1 \le \left\lfloor \frac{MAX - x}{k} \right\rfloor \end{gathered} \quad . \tag{4-18}$$

After a few manipulations,

$$x = MAX - \left\lceil \frac{N}{T}(MAX - MIN + 1) - 1 \right\rceil k \quad . \tag{4-19}$$

By replacing x in inequality (4-18), we can verify that equation (4-19) gives an optimal value of x. There will be another optimal value of x that is one smaller than that in equation (4-19) if $\frac{N}{T}(MAX - MIN + 1)$ is an integer. In the special case that exactly half of the processors are devoted to the iteration and k is unity, x becomes the expected number of cycles of the iteration, which for this distribution is the same as the median. Also, as N gets smaller, x tends toward MAX and as N approaches T, x tends towards MIN, just as expected. Note that no special treatment is required for the case that $k > 1$ because (4-17) is non-increasing in x.

Geometric Distribution

A uniform probability mass function $p(i)$ is not a good model for many types of iteration. In situations involving convergence, a geometric probability mass function may be a better approximation. At each cycle of the iteration, we proceed to the next cycle with probability q and stop with probability 1-q.

For generality, we still allow an arbitrary minimum number MIN of cycles of iteration. The maximum number, *MAX*, is infinite. Let $j=i\text{-}MIN$, where i is the number of cycles of the iteration. Then, the geometric probability mass function means that for any non-negative integer r,

$$P[j \geq r] = q^r \ , \tag{4-20}$$

and

$$P[j = r] = p(r) = q^r(1-q) \quad . \tag{4-21}$$

To use inequality (4-5), we find

$$\sum_{i=0}^{\infty} p(x+ik) = (1-q)\frac{q^{x-MIN}}{1-q^k} \tag{4-22}$$

Since (4-22) is non-increasing in x, no special treatment is required for the case that $k > 1$. Therefore, the optimal value of x satisfies

$$(1-q)\frac{q^{x+1-MIN}}{1-q^k} \leq \frac{N}{T} \leq (1-q)\frac{q^{x-MIN}}{1-q^k} \tag{4-23}$$

So,

$$x = \left\lfloor \log_q \frac{N(1-q^k)}{T(1-q)} \right\rfloor + MIN \tag{4-24}$$

To gain intuition about this expression, consider the special case where $k = 1$ and $q = 0.5$ meaning that after each cycle of the iteration we are equally likely to proceed as to stop. Further specializing, when exactly half of the processors are devoted to the iteration, x becomes $MIN+1$, which is the expected number of iteration cycles, as well as the median. Note that practical applications are likely to have a larger value for q, in which the median will be smaller than the mean.

The expressions for x, the assumed number of iteration cycles, are simple enough to be of practical use in a parallelizing compiler that assumes a geometric or uniform probability mass function. However, there remains the question of determining how many processors to devote to an iteration.

4.1.3 Processor Partitioning

In the previous discussion, we assumed that we can somehow allocate the optimal number N of processors to the data-dependent iteration. Now we give a strategy for determining this number. Unfortunately, in practical situations, the detailed structure of the dataflow graph affects the optimal choice of N. To keep the scheduler simple, our preference is to adopt a suboptimal policy that is optimal for a subset of graphs and reasonable for the rest: Select N to minimize the run-time cost of the iteration actor. The suboptimality of this policy is explained in section 3.2.

First, let us again consider a geometric distribution on the number of cycles of the iteration. Since

$$\begin{aligned}\sum_{i=x+1}^{\infty} p(i)\left\lceil\frac{i-x}{k}\right\rceil &= \sum_{i=x+1}^{\infty} q^{i-MIN}(1-q)\left\lceil\frac{i-x}{k}\right\rceil \\ &= q^{x-MIN}(1-q)\sum_{j=1}^{\infty} q^{j}\left\lceil\frac{j}{k}\right\rceil = \frac{q^{x-MIN+1}}{1-q^{k}}\end{aligned}, \tag{4-25}$$

we get

$$C(N,k,x) = N\tau x + T\tau\frac{q^{x-MIN+1}}{1-q^{k}} \quad . \tag{4-26}$$

Since both x and τ are functions of N, dependency of the run-time cost on N can not be clearly defined. If we replace x using equation (4-24), we get

$$C(N,k,x) = N\tau\left(MIN+\left\lfloor \log_q\frac{N(1-q^k)}{T(1-q)}\right\rfloor\right)+T\tau\frac{q}{1-q}q^{\left\lfloor \log_q\frac{N(1-q^k)}{T(1-q)}\right\rfloor} \quad , \tag{4-27}$$

which is a complicated transcendental that looks as if it has to be minimized numerically. Fortunately, we can draw some intuitive conclusions for certain interesting special cases.

Consider a case where linear speedup of the iteration actor is possible. In other words, $\tau N = K$, where K is the total amount of computation in one cycle of the iteration. Equation (4-27) simplifies slightly to

$$C(N, k, x) = K(MIN) + K\left\lfloor \log_q\left(\frac{N}{T}\right)\right\rfloor + T\frac{K}{N}\frac{q}{1-q}q^{\left\lfloor \log_q\left(\frac{N}{T}\right)\right\rfloor} \quad , \tag{4-28}$$

where k becomes *1*. The first term is constant in N and the second term is decreasing in N. We will now show that the third term is approximately constant in N, suggesting that $C(N,k,x)$ is minimized by selecting the largest possible value, $N=T$. This is intuitively appealing, since with linear speedup applying more processors to the problem would seem to make sense. To show that the third term is approximately constant, note that

$$\frac{N}{qT} = q^{\left(\log_q\left(\frac{N}{T}\right)-1\right)} > q^{\left\lfloor \log_q\left(\frac{N}{T}\right)\right\rfloor} \geq \frac{N}{T} \quad . \tag{4-29}$$

Consequently, the third term is bounded as follows,

$$\frac{K}{1-q} > T\frac{K}{N}\frac{q}{1-q}q^{\left\lfloor \log_q\left(\frac{N}{T}\right)\right\rfloor} \geq \frac{Kq}{1-q} \quad . \tag{4-30}$$

These bounds do not depend on N. Note, however, that when $N=T$, this third term is at its minimum, $Kq/(1-q)$. It may also be at this minimum for other values of N, but since the middle term in equation (4-28) decreases as N increases, the conclusion is that N should be made as large as possible, namely $N=T$.

Consider another extreme situation, when no speedup of the iteration is possible and $k = 1$. In this case, $\tau = K$, independent of N. For the third term in equation (4-27), we use similar bounding arguments and find that both the upper and the lower bounds on the third term increase linearly in N. The first and the second term also increase in N.

Hence, the conclusion is that if no speedup is possible, we should use as few processors as possible, or $N = 1$. This is a reassuring conclusion.

For general speedup characteristics, we can not draw general conclusions. This suggests that a compiler implementing this technique may need to solve equation (4-27) for the optimal *N*. If the total number of processors *T* is modest, then this task should not be too onerous, although we certainly prefer to not have to do it. The task can be somewhat simplified, perhaps, by the observation that we can shrink the range of *N* to be examined by looking into the range of *C(N,k,x)*. Using inequality (4-23) in equation (4-26), we get

$$\tau N\left(x + \frac{q}{1-q}\right) \le C(N, k, x) \le \tau N\left(x + \frac{1}{1-q}\right) \quad . \tag{4-31}$$

For some values of *N*, the upper bound is smaller than the lower bound for some other value of *N*, so we can ignore the latter *N*'s.

Now, consider the case where *p(i)* is a uniform distribution. Since

$$\sum_{i=x+1}^{MAX} \left\lceil \frac{i-x}{k} \right\rceil = \sum_{i=1}^{l} ik + (l+1)(MAX - x - lk) \qquad where \qquad l = \left\lfloor \frac{MAX - x}{k} \right\rfloor \tag{4-32}$$

the runtime cost becomes

$$C(N) = N\tau x + \frac{T\tau}{MAX - MIN + 1}\left(\frac{kl(l+1)}{2} + (l+1)(MAX - x - lk)\right) \tag{4-33}$$

We can replace *x* with the value given by equation (4-19). Observe that if we define *R=MAX-MIN+1*, then

$$C(N, k, x) = N\tau x + \frac{T\tau}{R}\frac{k}{2}\left\lceil \frac{NR}{T} - 1 \right\rceil \left\lceil \frac{NR}{T} \right\rceil \quad . \tag{4-34}$$

When *RN/T* is large,

$$\left\lceil \frac{NR}{T} \right\rceil \approx \frac{NR}{T} \quad . \tag{4-35}$$

This crude approximation simplifies the analysis compared to a bounding argument like that above, which can be carried out and leads to the same conclusion. If in addition we assume linear speedup, so that $\tau N = K$, then equation (4-34) simplifies to

$$C(N, k, x) = K(MAX) + \frac{K}{2}k\left(1 - \frac{NR}{T}\right) \quad . \tag{4-36}$$

We see that this function is decreasing linearly in N, suggesting again that we should select the maximum $N=T$.

To summarize, we have derived a general cost function that depends on the speedup attainable for the iteration as more processors are devoted to it. The cost function was given for the special cases when the probability mass function for the number of cycles of the iteration is geometric or uniform. Furthermore, simple special situations lead to intuitive results. Namely, if linear speedup is attainable, then we should devote all the processors to the iteration. If no speedup is possible, then we should devote no more than one processor to the iteration. For more general situations, finding the optimal number of processors requires numerically solving a complicated transcendental.

4.2. CONDITIONALS

Decision making capability is an indispensable requirement of a programming language for general purpose applications, and even for signal processing applications. A dataflow representation for an if-then-else is shown in figure 4.5 (a). A data token x is routed by the SWITCH actor to one of the two functions depending on the value of the boolean token c. The appropriate function is executed, and its result is selected by the SELECT actor depending on the same boolean token. The functions represent subgraphs of arbitrary complexity. The local schedules of both functions are displayed in figure 4.5 (b) and (c) assuming that N processors are assigned to the if-then-else construct.

4.2.1 Expected Runtime Cost

We assume that the probability p_1 with which the "TRUE" branch (branch 1) is selected is known. The "FALSE" branch (branch 2) is selected with the probability $p_2 = 1 - p_1$. Let τ_{ij} be the finish time of the local schedule of the i-th branch on the j-th processor. And, let $\hat{\tau}_j$ be the finish time on the j-th processor in the optimal profile of the conditional construct. We compute the optimal values $\hat{\tau}_j$ to minimize the runtime cost of the construct. When the i-th branch is selected at runtime, the runtime cost becomes

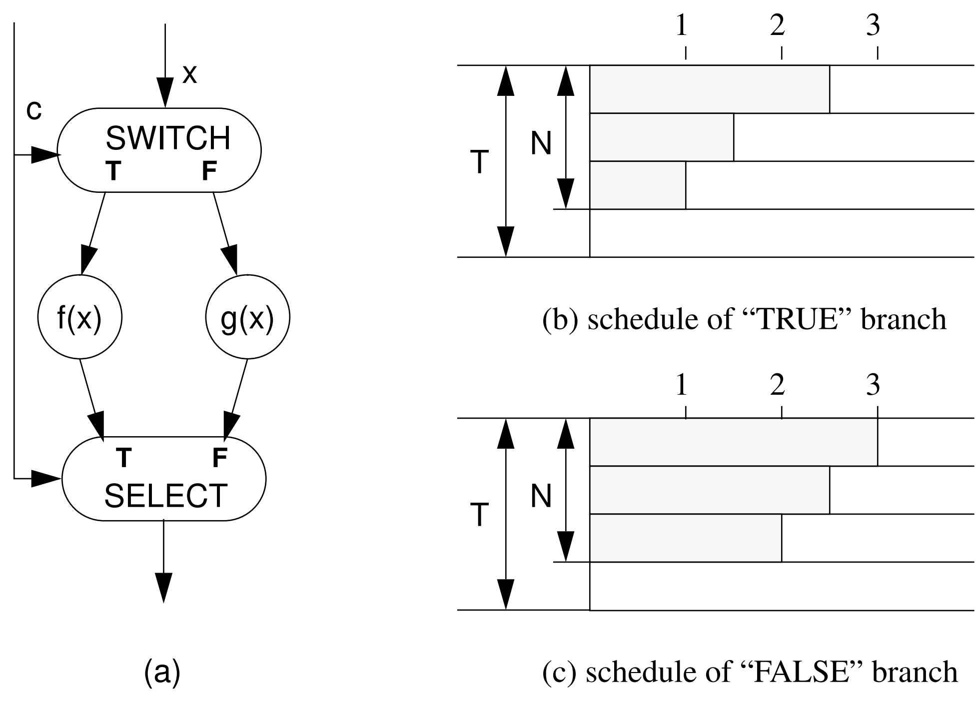


Figure 4.5 (a) A dataflow graph for the expression: y = if(c) then f(x) else g(x). We assume that f(x) and g(x) represent subgraphs of arbitrary complexity. Gantt charts show examples of the local schedule of "TRUE" branch in (b), and that of "FALSE" branch in (c). N processors out of total T processors are assigned to the if-then-else construct.

$$\sum_{k=1}^{N} \hat{\tau}_k + T max\left[0, \max_{j \in [1, N]} (\tau_{ij} - \hat{\tau}_j)\right] \tag{4-37}$$

Therefore, the expected runtime cost C is

$$C = \sum_{k=1}^{N} \hat{\tau}_k + T \sum_{i=1}^{2} p_i max\left[0, \max_{j \in [1, N]} (\tau_{ij} - \hat{\tau}_j)\right] \tag{4-38}$$

It is not feasible to obtain the closed form solutions for $\hat{\tau}_j$ that minimize the expected cost because the *max* function is non-linear and discontinuous. Instead, we developed a numerical technique:

1. Initially, take the maximum finish time of both branch schedules for each processor according to E. Lee's method [Lee88]: that is

$$\hat{\tau}_j = \max_{i \in [1, 2]} \{\tau_{ij}\} \quad . \tag{4-39}$$

   The initial expected cost becomes

$$C = \sum_{j=1}^{N} \max_{i \in [1, 2]} \{\tau_{ij}\} \quad . \tag{4-40}$$

2. Define $\alpha_i = max\left[\max_{j \in [1, N]} (\tau_{ij} - \hat{\tau}_j), 0\right]$. Initially all $\alpha_i = 0$. The variable α_i represents the cost per processor when the profile $\{\hat{\tau}_j\}$ is exceeded at run time because of branch i being selected, as illustrated in equation (4-38). We define *bottle-neck* processors of branch i as the processors $\{j\}$ that satisfy the relation $\tau_{ij} - \hat{\tau}_j = \alpha_i$.

3. Repeat for each $i \in [1, 2]$ until no more changes are made on $\hat{\tau}_j$'s:

   (a) Select an index set Θ_J :

$$\Theta_J = \{ j | \tau_{ij} - \hat{\tau}_j = \alpha_i, \ \tau_{kj} - \hat{\tau}_j < \alpha_k, \quad for \quad all \quad i \neq k \}$$

In words, Θ_J is the set of processors that are bottle-neck processors of branch *i*, but not of other branches.

(b) If we vary $\hat{\tau}_j$ by a small amount, replacing it with $\hat{\tau}_j - \delta$, for all $j \in \Theta_J$, the variation of the expected cost becomes

$$\Delta C = (-|\Theta_J| + Tp_i)\delta \ . \tag{4-41}$$

Increase δ as long as the quantity in equation (4-41) is negative, implying that the expected cost is decreasing by varying $\hat{\tau}_j$'s. At a certain δ, there will be an index *n*, $n \in \Theta_J$, such that $\tau_{kn} - \hat{\tau}_n$ becomes α_k for any $k \neq i$.

At this point, we update the index set Θ_J, which gets smaller as we increase δ, by extracting index *n* from Θ_J. Note that increasing δ means increasing α_i.

If we cannot decrease the expected cost by increasing δ any more, we go back to step 3.

Our algorithm is a greedy algorithm, but also an optimal algorithm. The optimality of the algorithm is proven in the next subsection.

4.2.2 Optimality Of The Proposed Algorithm

Suppose we have a set of optimal values, $\{\tilde{\tau}_j\}$, for the optimal profile of the construct. The optimal $\{\tilde{\tau}_j\}$ possesses some nice properties that are described in the following two theorems.

Theorem 4.2: *When N < T,* $\prod_{i \in [1, 2]} \alpha_i = 0$ *. In other words, at least one* α_i *should be zero for optimal* $\{\tilde{\tau}_j\}$.

Proof:

We prove it by contradiction. Let's assume that all α_i's are positive.

Let α_{min} be the smallest α_i. We can choose δ, a positive number, which is smaller than α_{min}. If we increase all $\tilde{\tau}_j$'s by δ, the expected cost varies:

$$\frac{\Delta C}{\delta} = N - T\sum_i p_i = N - T < 0 \qquad (4\text{-}42)$$

Since the expected cost decreases, the $\{\tilde{\tau}_j\}$ is not optimal, which is a contradiction. Q.E.D.

In case *N* = *T*, the decrease of the expected cost in equation (4-42) becomes zero. Therefore, we may increase the $\tilde{\tau}_j$'s until at least one α_i becomes zero. Without loss of generality, we assume that $\alpha_1 = 0$ throughout this subsection. Now we proceed to another theorem concerning the range of $\tilde{\tau}_j$'s.

Theorem 4.3: $\tau_{min,\,j} \le \tilde{\tau}_j \le \tau_{max,\,j}$ *where* $\tau_{min,\,j} = \min_{i \in [1, 2]} \{\tau_{ij}\}$ *and* $\tau_{max,\,j} = \max_{i \in [1, 2]} \{\tau_{ij}\}$ *, for all j.*

Proof:

Let's assume that there is an index *n* such that $\tilde{\tau}_n < \tau_{min,\,n}$. That means

$$\alpha_1 \geq \tau_{1n} - \tilde{\tau}_n > 0, \quad and \quad \alpha_2 \geq \tau_{2n} - \tilde{\tau}_n > 0 \quad , \tag{4-43}$$

which is contradictory to Theorem 4.2. Therefore, such an index does not exist.

On the other hand, assume that there is an index *m* such that $\tilde{\tau}_m > \tau_{max,\, m}$. When we decrease $\tilde{\tau}_m$ by δ, where $\delta = min\{\tilde{\tau}_m - \tau_{1m}, \tilde{\tau}_m - \tau_{2m}\}$, the change of the expected cost becomes $\frac{\Delta C}{\delta} = -1$ because α_i 's do not vary in this case. This reveals that $\tilde{\tau}_m$ is not optimal. Q.E.D.

The next two theorems describe the relation between the $\tilde{\tau}_j$'s and α_i's.

Theorem 4.4: *For each* $\tilde{\tau}_j$ *, there exists a branch index k such that* $\tau_{kj} - \tilde{\tau}_j = \alpha_k$.

Proof:

Assume that there is no such index *k*. Then, we can select a positive value δ such that

$$\delta = \min_{i \in [1, 2]} \{\alpha_i - (\tau_{ij} - \tilde{\tau}_j)\} \quad . \tag{4-44}$$

If we decrease $\tilde{\tau}_j$ by δ, the expected cost changes: $\frac{\Delta C}{\delta} = -1$ because the α_i's do not vary. This contradicts the assumption that $\tilde{\tau}_j$ is optimal. Therefore, there must be a branch index *k* as stated in the theorem. Q.E.D.

Theorem 4.5: *There is an optimal profile in which there is at least one processor index n such that*

$$\tau_{2n} - \tilde{\tau}_n = \alpha_2, \quad \textit{and} \quad \tilde{\tau}_n = \tau_{1n} \quad (\textit{recall that we assume } \alpha_1 = 0). \tag{4-45}$$

Proof:

Assume that there is no such a processor index *n*. Let $\Lambda_J = \{j | \tau_{2j} - \tilde{\tau}_j = \alpha_2\}$. If we increase $\tilde{\tau}_j$'s for $j \in \Lambda_J$ by a small amount, the change of the expected cost becomes

$$\frac{\Delta C}{\delta} = |\Lambda_J| - Tp_2 \geq 0 . \tag{4-46}$$

On the other hand, if we decrease $\tilde{\tau}_j$'s for $j \in \Lambda_J$ by a small amount, we get

$$\frac{\Delta C}{\delta} = -|\Lambda_J| + Tp_2 \geq 0 \tag{4-47}$$

From equation (4-46) and equation (4-47), $p_2 = \frac{|\Lambda_J|}{T}$. For other values of p_2, we reach a contradiction. For that specific value of p_2, we can decrease $\tilde{\tau}_j$'s for $j \in \Lambda_J$ until there appears an index satisfying equation (4-45). Q.E.D.

Now, we are ready to prove the optimality of our proposed algorithm. First of all, note that the $\hat{\tau}_j$'s obtained from the proposed algorithm satisfy all the relations between $\hat{\tau}_j$'s and α_i's described in theorems 4.2 to 4.5 since the algorithm preserves them throughout the intermediate procedure.

Theorem 4.6: *The $\hat{\tau}_j$'s obtained from the proposed algorithm are optimal.*

Proof:

We will prove this main result by contradiction. Assume that there is an optimal profile $\{\tilde{\tau}_j\}$ that is different from $\{\hat{\tau}_j\}$ obtained from the proposed algorithm.

Let's select the indices of processors, Φ_J, such that

$$\Phi_J = \left\{ j | \tilde{\tau}_j < \hat{\tau}_j, \hat{\tau}_j - \tilde{\tau}_j = \max_{j \in [1, N]} \{\hat{\tau}_j - \tilde{\tau}_j\} \right\} \quad . \tag{4-48}$$

We get that $\hat{\tau}_j > \tau_{1j} \quad for \quad j \in \Phi_J$, from theorem 4.2 and the assumption $\alpha_1 = 0$. Such indices j are, therefore, contained in Θ_J, the index set in step 2-(b) in our algorithm, at the moment that the expected cost does not decrease by further increasing δ. In other words,

$$\frac{\Delta C}{\delta} = -|\Theta_J| + Tp_2 > 0. \tag{4-49}$$

We just showed that $\Phi_J \subseteq \Theta_J$. Therefore, if we increase $\tilde{\tau}_j$'s for $j \in \Phi_J$ by δ, then from inequality (4-49),

$$\frac{\Delta C}{\delta} = |\Phi_J| - Tp_2 < 0 \ . \tag{4-50}$$

This means that we can decrease the expected cost by varying the $\tilde{\tau}_j$'s, which is contradictory to the assumption that the $\tilde{\tau}_j$'s are optimal. If the index set Φ_J is empty, we can find out another index set Ξ_J such that $\Xi_J = \{j | \tilde{\tau}_j > \hat{\tau}_j\}$. From theorem 4.5, the index set Ξ_J contains an index n satisfying equation (4-45). Hence, the index set Ξ_J is a superset of Θ_J, In the proposed algorithm, a superset of Θ_J is shrunk down to Θ_J eventually. In other words, we can reduce the expected cost by decreasing $\tilde{\tau}_j$'s for $j \in \Xi_J$ further. Q.E.D.

We can show that the previous algorithms by E. Lee [Lee88] and Loeffler et. al. [Loe88] are optimal only in some special cases. If the quantity in equation (4-41) is always positive for all indices i, the optimal profile coincides with E. Lee's solution. On the other hand, if $'p_i < 1 \quad for \quad p_i < p_k$, and $N < T$, Loeffler's solution becomes the optimal profile. Their solutions, however, are not optimal in general.

Now we consider the example shown in figure 4.5. Suppose $p_1 = 0.3$ and $p_2 = 0.7$. The initial profile in our algorithm, which is same as E. Lee's profile, is shown in figure 4.6 (a). In this specific example, it is same as Loeffler et. al.'s profile. Let's trace down the proposed algorithm. For $i = 1$ in the step 3, set Θ_J is empty; go to the next index $i = 2$. In step 3-(a), $\Theta_J = \{1, 2, 3\}$. The quantity in equation (4-41) becomes

$$\frac{\Delta C}{\delta} = -3 + 4 \times 0.7 = -0.2 \ . \tag{4-51}$$

This quantity is not changed until we increase δ to 0.5. Hence, we decrease all $\hat{\tau}_j$'s by 0.5. For $\delta > 0.5$, the index set is decreased to $\Theta_J = \{2, 3\}$ and the cost change in equation (4-41) becomes

$$\frac{\Delta C}{\delta} = -2 + 4 \times 0.7 = 0.8 . \tag{4-52}$$

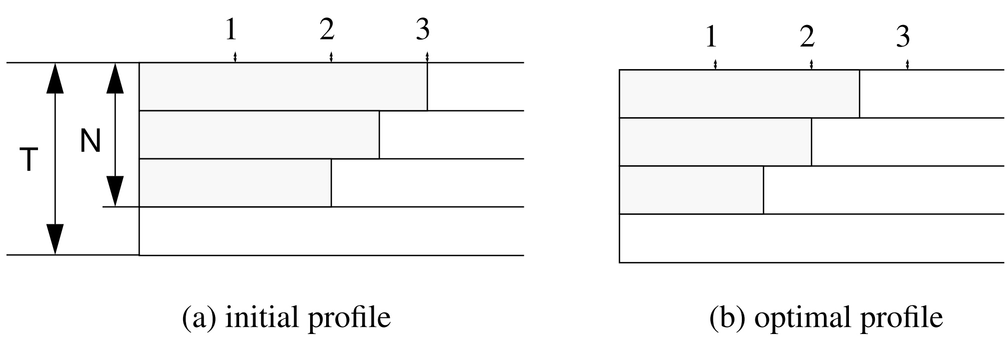


Figure 4.6 Generation of the optimal profile for the conditional construct in figure 4.5. (a) initial profile in our algorithm, which is same as E. Lee's profile [Lee88]. In this specific example, it is also same as the Loeffler et. al.'s profile. (b) The optimal profile obtained by our algorithm.

We do not gain further by increasing δ by more than *0.5*. The resulting profile is shown in figure 4.6 (b), and is optimal.

4.2.3 M-way Branching: Case Construct

We can generalize the proposed algorithm to the *M*-way branch construct. To realize an *M*-way branch, we use a nested if-then-else structure, or a single *case* construct (figure 4.7). For a nested if-then-else structure, we have to apply our algorithm recursively for each if-then-else construct. Since our technique seeks a locally optimal solution, repeated application of the proposed algorithm may be detrimental to the overall performance. On the other hand, we can encode the conditional booleans into a single integer and choose the intended branch in the *case* construct. Using the case construct is

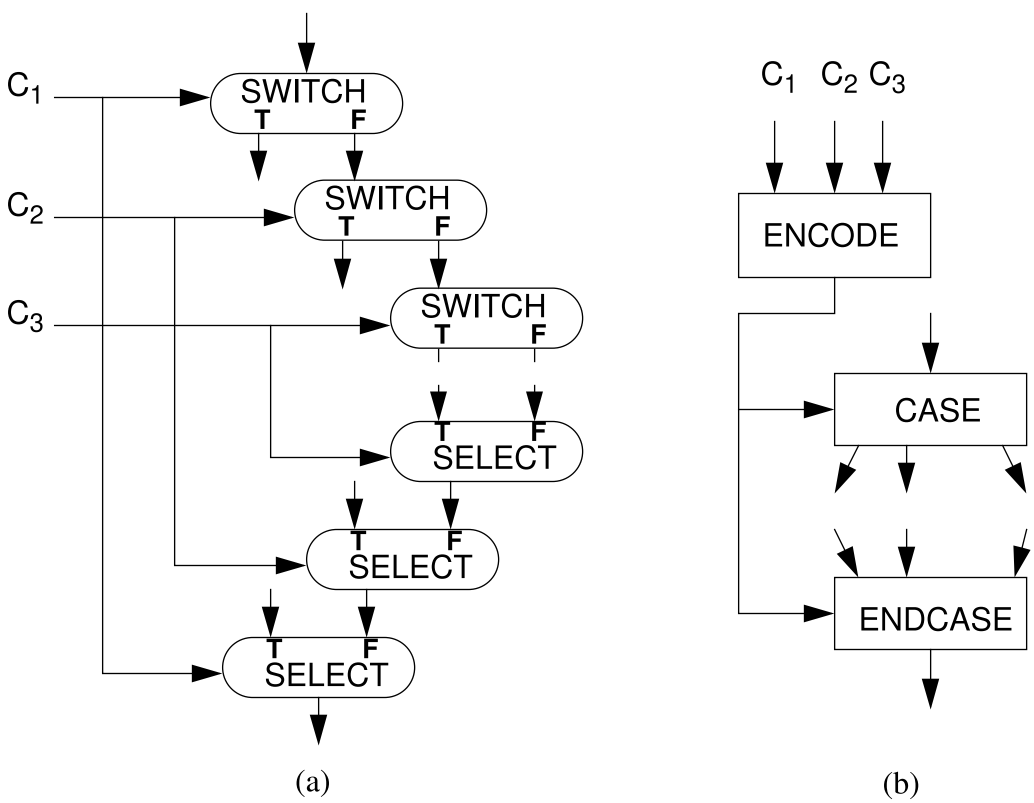


Figure 4.7 Two possible realizations of an M-way branching: (a)by a nested if-then-else structure, or (b) by a single *case* construct.

simpler in this representation. In this section, we generalize our algorithm for conditionals to the *M*-way branching case.

The proposed algorithm can be generalized as follows;

(1) Increase the range of the branch index, from [1,2] to [1, M].

(2) Scan the branch indices as many times as necessary until no more changes are made for $\{\hat{\tau}_j\}$ during one complete cycle in step 3 of the numerical optimization.

For if-then-else construct, only one complete cycle is enough in step 3.

Theorems 4.2 to 4.4 still hold. Unfortunately, the optimality of the generalized algorithm has not yet been proved. It is still an open problem whether the generalized algorithm is optimal or not. The generalized algorithm, however, always performs better than the generalized versions of E. Lee's profile and Loeffler et. al.'s [Lee88][Loe88].

4.2.4 Processor Partitioning

In the previous discussion, we assumed that we can somehow allocate the optimal number of processors *N* to the conditionals. For a given *N*, we obtained the optimal profile $\{\hat{\tau}_j\}$ minimizing the expected cost in equation (4-38). How to select the optimal number *N* is the next question we have to answer. Since no closed form for the optimal *N* minimizing the expected cost exists, we resort to a simple numerical technique. For all possible values of *N*, *N=1,2,...,T*, we compute the expected costs of equation (4-38) based on the optimal $\{\hat{\tau}_j\}$ s. Comparing those expected costs, we can select the optimal *N* that gives the smallest cost. For a moderate number of processors, this simple technique is fast enough. In a real situation, we are usually able to reduce the search space for *N* significantly. For example, the maximum number of processors that can be used for any branch can be limited. In that case, we need not examine any *N* larger than that maximum number.

4.3. RECURSION

Recursion is a construct that calls itself as a part of the computation if a termination condition is not satisfied. Most high level programming languages provide this construct since it makes a program compact and easy to understand. The number of recursive calls, called the *depth of recursion*, is usually not known at compile-time since the termination condition is calculated at run-time. In the dataflow paradigm, a recursion construct can be represented as a large actor which contains a SELF actor, which is understood to represent a reference to the containing actor (figure 4.8). If a recursion construct has only one SELF actor, as shown in figure 4.8, the function of the actor can be translated into a data-dependent iteration actor like figure 4.1 (b). Accordingly, the scheduling decision of

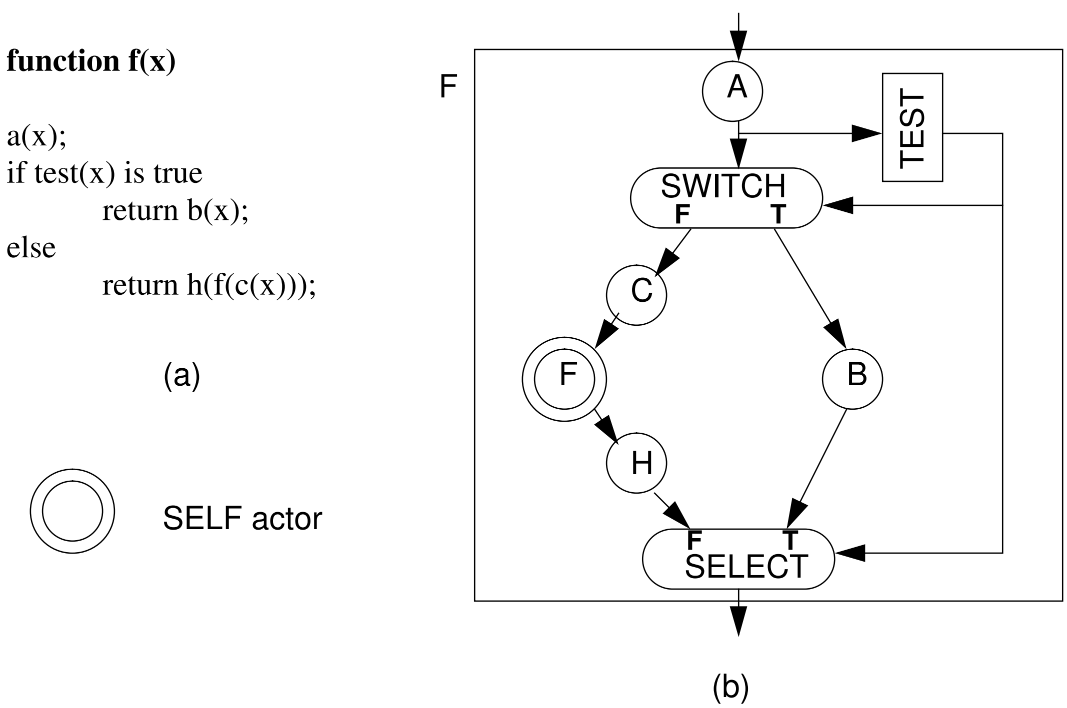
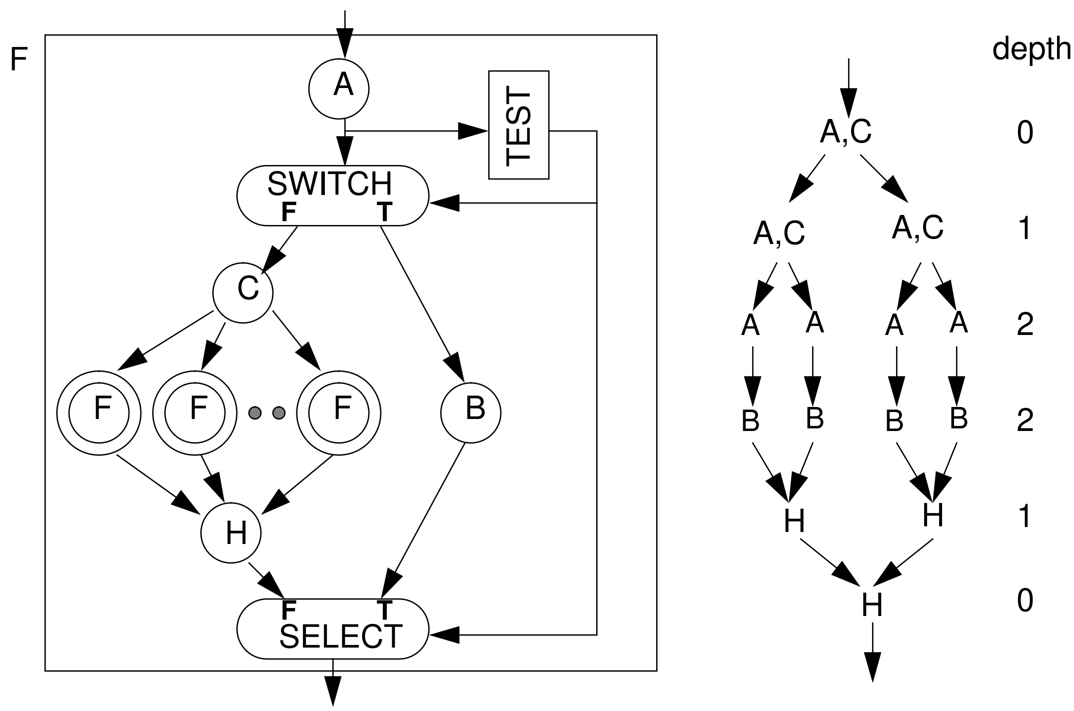


Figure 4.8 (a) An example of a recursion construct and (b) its dataflow representation. The SELF actor represents the recursive call.

the recursion actor is same as that of the translated data-dependent iteration actor, which will be verified at the end of this section. In this section, we consider a generalized recursion construct that may have more than one SELF actor. The number of SELF actors in a recursion construct is called the *width* of the recursion.

A generalized recursion construct is shown in figure 4.9 (a). In most real applications, we expect that the width of the recursion is no more than two. The computation tree of a recursion construct with width 2 and depth 2 is illustrated by a mirrored complete binary tree in figure 4.9 (b). The top node in the computation graph spawns two children since the termination condition is “FALSE”. Each child also spawns two children since its termination condition is also “FALSE”. Hence, the number of descendents at depth *2* becomes *4*. At depth *2*, each recursion construct (grandchildren of the top recursion con-



(a) (b)

Figure 4.9 (a) A generalized recursion construct. (b) The computation tree of a recursion construct with two SELF actors when the depth of the recursion is two.

struct) completes execution by executing actor B since the termination condition becomes “TRUE”. After the four grandchildren complete their execution, the two children execute actor H and complete execution. Finally, the top recursion construct completes after the execution of actor H.

We assume that all nodes of the same depth in the computation tree have the same termination condition. We will discuss the limitation of this assumption at the end of this section. We also assume that the run-time probability mass function of the depth of the recursion is known or can be approximated at compile-time. Our analysis in this section is based on these assumptions.

The potential parallelism of the computation tree of a generalized recursion construct may be huge since nodes at the same depth can be executed concurrently. The maximum degree of parallelism, however, is not known at compile-time. When we exploit the parallelism of the construct, we should consider the resource limitation. We may have to sacrifice parallelism in order to not deadlock the system. Restricting the parallelism in case the maximum degree of parallelism is too large has been recognized as a difficult problem to be solved in a dynamic dataflow system. Our approach proposes an efficient solution by taking the degree of parallelism as an additional component of the profile of the recursion construct.

4.3.1 Expected Runtime Cost

Suppose that the width of the recursion construct is k. Let the depth of the recursion construct be a random variable I with a known probability mass function $p(i)$. We denote the *degree* of parallelism by d, which means that the descendents at depth d in the computation graph are assigned to different processor groups. A descendent recursion construct at depth d is called a *ground* construct. We denote the size of each processor group by N. The total number of processors devoted to the recursion construct is Nk^d. The profile of a recursion construct is defined by three parameters: the assumed depth of recursion x, the degree of parallelism d, and the size of a processor group N. Our approach optimizes the parameters to minimize the expected cost of the recursion construct. An example of the profile of a recursion construct is displayed in figure 4.10.

Let τ be the sum of the execution times of actors A, C, and H. And, let τ_o be the sum of the execution times of actors A and B. Then the schedule length, l_x, of a ground construct becomes

$$\begin{aligned} l_x &= \tau(k^0 + k^1 + \dots + k^{x-d-1}) + \tau_o k^{x-d} \\ &= \tau\frac{k^{x-d}-1}{k-1} + \tau_o k^{x-d} \end{aligned} \quad \text{, when } x > d \text{ and } k > 1. \qquad (4\text{-}53)$$

The latter formula for l_x in equation (4-53) holds for $x \geq d$ and any positive k, though dur-

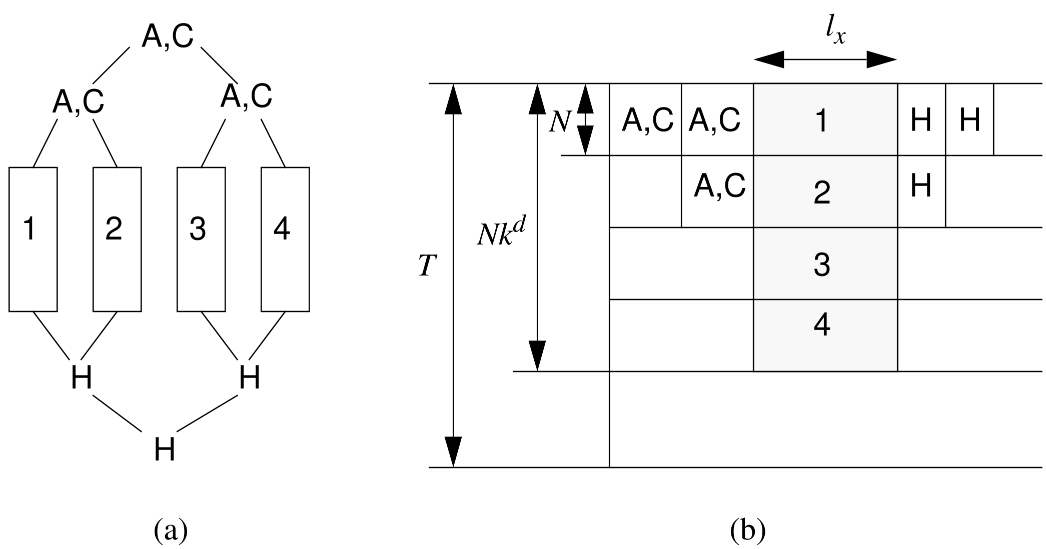


Figure 4.10 An example of the profile of a recursion construct of width 2. When the degree of parallelism is 2, the computation graph is reduced to as shown in (a). The nodes 1,2,3,4 at depth 2 correspond to grandchildren recursion constructs that are mapped to different processor groups as shown in (b). The schedule length of the grandchildren recursion constructs, l_x, is a function of the assumed depth of recursion x and the degree of parallelism d (equation (4-53)).

ing the derivation it is assumed that $x > d$, $k > 1$.

At run-time, some processors will be idled if the actual depth of recursion is different from the assumed depth of recursion, which is illustrated in figure 4.11. When the actual depth of recursion i is smaller than the assumed depth x, the recursion processors are idled. Otherwise, the non-recursion processors are idled. The processors assigned to the recursion construct are called *recursion processors*, the other processors *non-recursion processors*. We let R denote the cost of the recursion excluding the dotted area in figure 4.10 (b). This basic cost R is equal to $\dfrac{N\tau(k^d - 1)}{k - 1}$.

For $i \le x$, the runtime cost, C_1, becomes R plus the cost of the dotted area in figure 4.10 (b), or

$$C_1 = R + Nk^d\left(\tau\frac{k^{x-d} - 1}{k - 1} + \tau_o k^{x-d}\right) \tag{4-54}$$

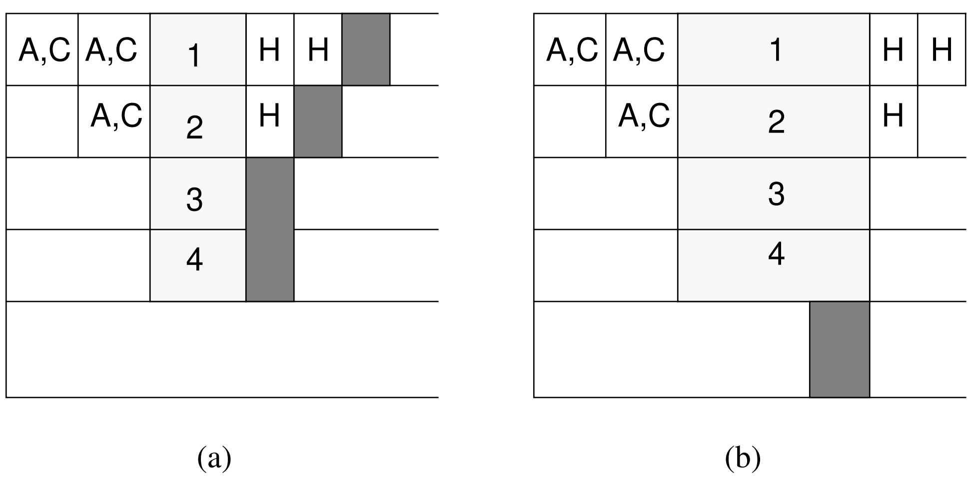


Figure 4.11 A quasi-static schedule is constructed based on an assumed depth of recursion x. The cost of the recursion actor is the sum of the dotted area (execution time) and the dark area (idle time due to the construct). Two possible cases are displayed depending on the actual depth i of the recursion in (a) for $i < x$ and in (b) for $i > x$.

assuming that x is not less than d. For $i > x$, the cost C_2 is C_1 plus the idle time on the non-recursion processors, or

$$C_2 = R + Nk^d\left(\tau\frac{k^{i-d}-1}{k-1} + \tau_o k^{i-d}\right) + (T - Nk^d)\left(\frac{\tau}{k-1} + \tau_0\right)(k^{i-d} - k^{x-d}) \quad (4\text{-}55)$$

Therefore, the expected cost of the recursion construct, *C(N,d,x)* becomes

$$C(N, d, x) = R + \sum_{i=0}^{x} p(i)C_1 + \sum_{i=x+1}^{\infty} p(i)\ C_2 \quad . \quad (4\text{-}56)$$

After a few manipulations,

$$C(N, d, x) = N\left(\tau\frac{k^x-1}{k-1} + \tau_o k^x\right) + \sum_{i=x+1}^{\infty} p(i)T\left(\frac{\tau}{k-1} + \tau_0\right)(k^{i-d} - k^{x-d}) \quad (4\text{-}57)$$

In the derivation of equation (4-57), we implicitly assume that the width of recursion, k, is greater than 1. However, the result holds even for $k=1$.

4.3.2 Assumed Depth Of Recursion And Degree Of Parallelism

First, we assume that N is fixed. Since the expected cost *C(N,d,x)* is a decreasing function of d, we select the maximum possible value for d. An upper bound of d is obtained by the processor constraint: $Nk^d \le T$. Since we assume that the assumed depth of recursion x is greater than the degree of parallelism d, the optimal value for d becomes

$$d = min\left(\left\lfloor \log_k \frac{T}{N} \right\rfloor, x\right) . \quad (4\text{-}58)$$

Next, we decide the optimal value for x from the observation that if x is optimal, the expected cost *C(N,d,x)* is not decreased when x is varied by 1 or -1. Therefore, we get

$$C(x) - C(x+1) = (\tau + \tau_o(k-1))\left(-Nk^x + Tk^{x-d}\sum_{i=x+1}^{\infty} p(i)\right) \le 0$$
$$C(x) - C(x-1) = (\tau + \tau_o(k-1))\left(Nk^{x-1} - Tk^{x-d-1}\sum_{i=x}^{\infty} p(i)\right) \le 0 \quad . \tag{4-59}$$

Rearranging the inequalities, we get the following,

$$\sum_{i=x+1}^{\infty} p(i) \le \frac{Nk^d}{T} \le \sum_{i=x}^{\infty} p(i) \quad . \tag{4-60}$$

Note the similarity of inequality (4-60) with that for data-dependent iterations, inequality (4-6). In particular, if k is *1*, the two formulas are equivalent, as expected.

The optimal values d and x depend on each other as shown in inequality (4-60) and equation (4-58). We may need to use iterative computations to obtain the optimal values of d and x. The strategy is as follows: First, we set $d = \left\lfloor \log_k \frac{T}{N} \right\rfloor$ from equation (4-58) assuming that x is greater than d. Based on this value of d, we compute the optimal x from inequality (4-60). If the computed value of x is greater than d, stop. Otherwise, decrease d by one and compute x again.

Geometric Distribution

Let's consider an example in which the probability mass function for the depth of the recursion is geometric with parameter q. For generality, we still allow an arbitrary minimum number MIN of cycles of iteration. From inequality (4-60), the optimal x satisfies

$$q^{x-MIN+1} \le \frac{Nk^d}{T} \le q^{x-MIN} \tag{4-61}$$

As a result, x becomes

$$x = \left\lfloor \log_q \frac{Nk^d}{T} \right\rfloor + MIN \quad . \tag{4-62}$$

Uniform Distribution

Suppose the probability mass function *p(i)* is uniform over the range *MIN* and *MAX*. From inequality (4-60), we get

$$\frac{MAX - x}{MAX - MIN + 1} \le \frac{Nk^d}{T} \le \frac{MAX - x + 1}{MAX - MIN + 1} \quad . \tag{4-63}$$

So,

$$\frac{Nk^d}{T}(MAX - MIN + 1) - 1 \le MAX - x \le \frac{Nk^d}{T}(MAX - MIN + 1) \quad . \tag{4-64}$$

Therefore, the optimal x becomes

$$x = MAX - \left\lceil \frac{Nk^d}{T}(MAX - MIN + 1) - 1 \right\rceil \quad . \tag{4-65}$$

4.3.3 Processor Partitioning

Up to now, we assumed that N is fixed. Since τ is a transcendental function of N, the dependency of the expected cost on the size of a processor group N is not clear. Instead, we examine all possible values for N, calculate the expected cost from equation (4-57), and choose the optimal N rendering the minimum cost. In most real applications, we can reduce the search space of N significantly.

In the case of geometric distribution for the depth of the recursion, the expected cost in equation (4-57) is simplified to

$$C(N, d, x) = N\left(\tau\frac{k^x - 1}{k - 1} + \tau_o k^x\right) + T(\tau + \tau_o(k-1))k^{x-d}\frac{q^{x - MIN + 1}}{1 - qk} \quad . \tag{4-66}$$

For $k = 1$,

$$C(N, d, x) = N\tau x + T\tau\frac{q^{x-MIN+1}}{1-q} + N\tau_o \quad , \tag{4-67}$$

which is equivalent to equation (4-26) when $k = 1$ except the third term. The third term is added as an overhead to detect the loop termination.

In case of uniform distribution,

$$\begin{aligned} C(N, d, x) = {} & N\left(\tau\frac{k^x - 1}{k - 1} + \tau_o k^x\right) \\ & + \frac{T(\tau + \tau_o(k-1))k^{x-d}}{(MAX - MIN + 1)(k-1)}\left(\frac{k - k^{MAX - x + 1}}{1 - k} - MAX + x\right) \end{aligned} \quad . \tag{4-68}$$

4.3.4 Limitation Of The Assumption

Recall that our analysis is based on the assumption that all nodes of the same depth in the computation tree have the same termination condition. This assumption was made to make the analysis tractable. At run time, however, we do not enforce that restriction; All nodes at the same depth need not be synchronized.

Our assumption roughly approximates a more realistic assumption, which we call the *independenc*e assumption, that all nodes of the same depth have equal probability of terminating the recursion and they are independent of each other. This equal probability is considered as the probability that all nodes of the same depth terminate recursion in our assumption. Note that the expected number of nodes at a certain depth is the same between both assumptions even though they describe different behaviors.

Under the independence assumption, the shape of the profile would be the same as shown in figure 4.11; The degree of parallelism d is maximized. And all recursion processors have the same schedule length of the ground constructs because the recursion processors are indistinguishable and all nodes at the same depth are assumed independent. However, the optimal schedule length l_x of the ground construct would be different. The length l_x is proportional to the number of executions of the recursion construct inside a ground construct. We call this number the *ground* number. The ground number can be any integer under the independence assumption while it belongs to a subset $\{0, k^1, k^2, \ldots\}$ under our assumption. Since the probability mass function of the ground number is likely to be too complicated under the independence assumption, it seems be impossible to obtain the optimal schedule length of the ground construct by the proposed approach. Therefore, we sacrifice performance by choosing a suboptimal schedule length under a simpler assumption. If the optimal ground number under the independence assumption is y, we expect that our simpler assumption will choose either $\lfloor \log_k y \rfloor$ or $\lfloor \log_k y \rfloor + 1$ as the assumed depth of recursion.

4.4. ADDITIONAL IDLE TIMES

In the preceding sections, we have determined the optimal profiles of three well-known types of dynamic constructs, minimizing the expected runtime cost of those constructs. The runtime cost of a dynamic construct includes not only the computation time of the construct but also the idle time padded to some processors after executing the construct to make the pattern of processor availability the same as the compiled one. In reality, however, there are two additional but significant sources of idle time caused by the construct. They should be added to the expected runtime cost before processors are partitioned (or, the optimal N is decided).

A compile-time profile of a construct assumes a fixed pattern of processor availability at the beginning. If the actual availability pattern is different from the assumed pattern, idle time should be inserted on the processors assigned to the construct (figure

4.12). In figure 4.12 (a), a data-dependent iteration actor is scheduled according to an optimal profile in figure 4.3 (a). We assume that N is equal to one for simplicity. When scheduling the dynamic construct, we position the actual pattern and the assumed pattern of processor availability to minimize the amount of idle time filled in the gap. Note that we may need to interchange the processors in the assumed profile of the construct. In figure 4.12 (b), we schedule an if-then-else actor, whose profile is modified from figure 4.6 (b). The pattern of processor availability at the beginning of the constructor need not be flat, though we assume it is in section 4.2 to derive a simple formula for the total cost. We position the assumed profile of the construct to minimize the idle time inserted in the gap between two mismatched patterns. In this case, the runtime cost may be reduced from what we obtain in section 4.2.

Up to now, we have assumed that the unassigned processors can be kept busy during the execution of the construct. If the program graph is highly parallelizable or the data-dependent execution time of the dynamic construct is small enough, this is not a bad

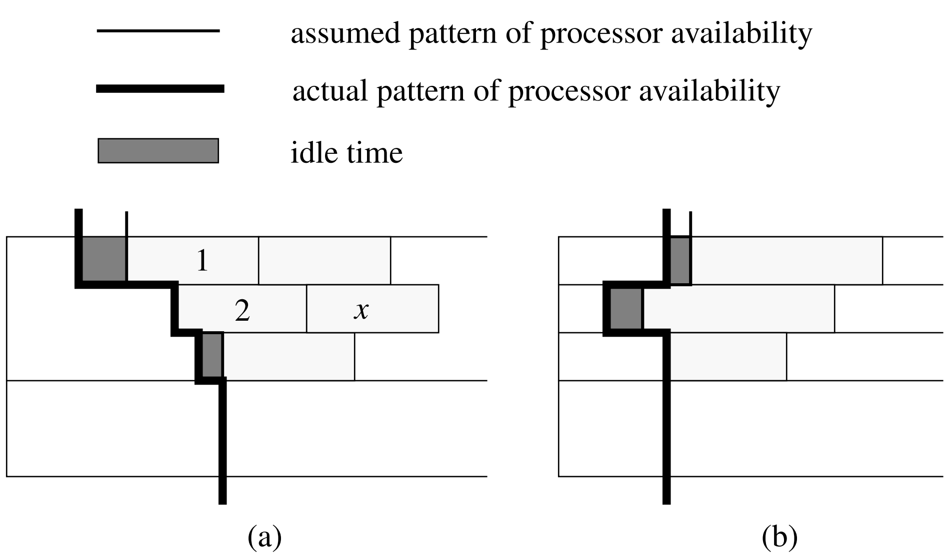
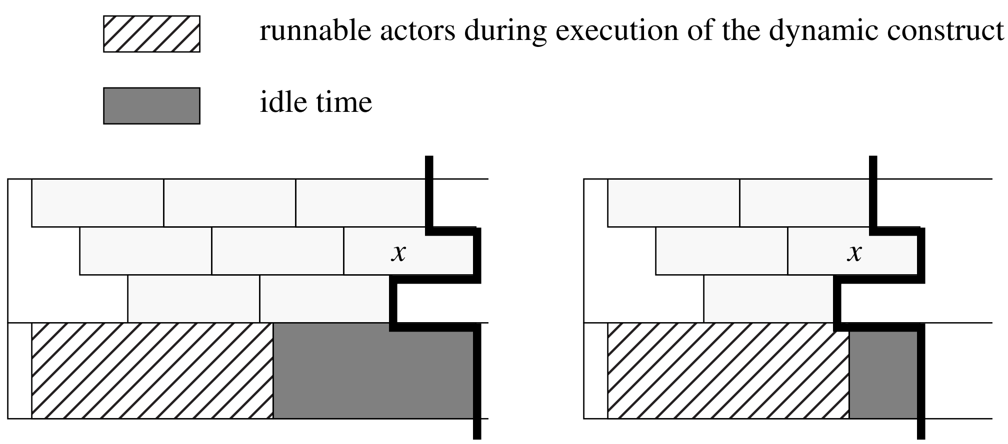


Figure 4.12 Idle time must be inserted before a dynamic construct is scheduled due to the mismatched pattern of processor availability between the actual one and the compiled one. Two examples are shown: (a) a data-dependent iteration actor in figure 4.3 (a), (b) an if-then-else actor modified from figure 4.6 (b).

assumption. Otherwise, we should consider the idle time on the unassigned processors during execution of the dynamic construct (figure 4.13 (a)). To reduce the unnecessary idle time on those processors, we modify the assumed profile of the dynamic construct. In figure 4.13 (b), for example, we decrease the assumed number of iteration cycles so that the pattern of processor availability is consistent with the original one. Though the inserted idle time in the compile-time schedule of figure 4.13 (b) is smaller than that of figure 4.13 (a), the expected runtime cost of the construct is the same whichever schedule is used at compile-time. For a recursion construct, we may decrease the assumed depth of the recursion for the same purpose. For a conditional construct, however, we do not change the profile, but just add the idle time into the cost of the actor.

The idle time before and during execution of a dynamic construct can not be computed before the main scheduling is performed. Before processor partitioning is made for a dynamic construct, we revise the expected cost adding that idle time. Thus, an optimal



(a) (b)

Figure 4.13 When the runnable actors are not enough to keep the unassigned processors busy during execution of a dynamic construct we should add the idle time on those processors to the runtime cost of the dynamic construct. An example is shown for a data-dependent iteration in (a). To reduce the unnecessary idle time, we reduce the assumed number of iteration cycles as shown in (b). Note that the revised profile keeps the same pattern of processor availability at the end.

profile of a dynamic construct is determined during the main scheduling.

5

IMPLEMENTATION: PTOLEMY

For as the body without the spirit is dead, so

faith without works is dead also.

--- James 2:26

Use of multiple programmable processors has become an attractive alternative to custom VLSI for many real-time digital signal processing applications. Widespread use of this alternative, however, will depend on the development of an effective software and hardware development environment. The typical environment pursued so far translates a block-diagram algorithm description into real-time code for multiple programmable processors [Lee89a][Zis87][Tha90]. The biggest impediment to the use of such a parallel computing system is the scarcity of techniques which effectively partition and schedule programs onto them, so that parallel hardware can be effectively utilized.

Blosim [Mes84] and **Gabriel** [Bie89][Lee89a] are respectively the first and the second generation design environments for signal processing applications developed at the University of California at Berkeley. While Blosim is aimed at algorithm development and simulation, Gabriel is aimed at real-time prototyping on parallel processors. Applications of Gabriel are limited to those with deterministic control flow that can be described using the synchronous dataflow (SDF) model of computation. By restricting to this model, several automated scheduling and code generation schemes have been developed [Bha91][Lee87a][Sih91].

Ptolemy [Buc91b] is a third generation software system being developed at Berkeley. It is a much more flexible and extensible software framework for simulation and rapid prototyping. The key difference between Ptolemy and its predecessors is that Ptolemy does not have a fundamental model of computation and scheduling built into its foundation. Instead, Ptolemy, as an object-oriented toolkit, provides a set of internal object-oriented interfaces so that heterogeneous environments built in the Ptolemaic framework can be easily merged as necessary.

While it is not fundamental to Ptolemy, the graphical user interface deals with descriptions of systems represented as block diagrams. It is therefore convenient to think of the basic module in Ptolemy as a *block*. An atomic block is called a *Star* and a hierarchical block is called a *Galaxy*. The outermost block, which contains the entire application, is known as a *Universe*. The entity that determines the order of the execution of the blocks is the *scheduler* (at compile time or run-time). The combination of a scheduler and a set of blocks that conform to the behavior expected by this scheduler is called a *domain*. Ptolemy is named after a famous astronomer because of the extensive use of cosmological metaphors in its basic structure.

Ptolemy is unique in that it supports the coexistence and interaction of diverse computational models, domains, in the same system. For instance, a dataflow domain can exist within a discrete-event domain. In figure 5.1, the network transport of one specific service - packet speech - is illustrated. This transport device divides into two pieces, the signal processing (compression, silence detection), which is best modeled with a time-driven synchronous sampling rate (the SDF domain), and the network (packet assembly/disassembly, switching and queueing), best modeled by the discrete event domain.

Ptolemy is written in C++, an object-oriented programming language. Through the object-oriented abstraction mechanism of polymorphism, new domains, including new computational models, new types of blocks, and new communication primitives among blocks, can readily be added to Ptolemy without any modifications to the Ptolemy core or to the previously implemented domains. Moreover, Ptolemy provides a seamless interface between heterogenous domains.

The proposed scheduling technique is being implemented in Ptolemy as a tool for multiprocessor code generation. The target applications of the proposed technique are expressed as combinations of the Synchronous Dataflow (SDF) domain and the Dynamic Dataflow (DDF) domain in the Ptolemy environment. The SDF domain is renamed as the Code Generation (CG) domain[1], and the DDF domain as the CGDDF domain, whose name is a juxtaposition of CG and DDF meaning the DDF domain for Code Generation. In this chapter, we discuss how to express the target applications and how to schedule

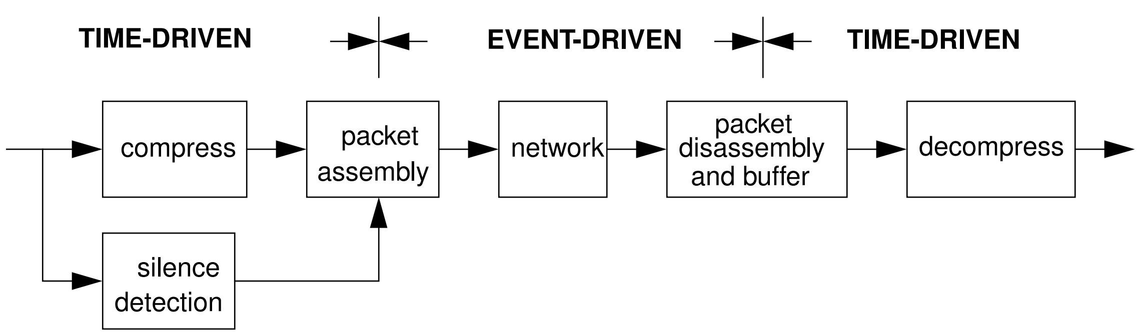


Figure 5.1 A packet speech system simulation requires the combination of signal processing (compression, silence detection) and queueing (packet assembly/disassembly and packet transport).

1. The SDF domain and the DDF domain in Ptolemy refer to domains for simulations. The CG and the CGDDF domains are respectively the code-generation versions of the SDF and the DDF domains.

them in the Ptolemy environment. As of this writing, the code generation functionality is not fully supported; scheduling is performed, but multiprocessor code is not generated yet. The scheduling examples of the proposed technique in Ptolemy will be discussed in the next chapter.

5.1. MIXED-DOMAIN APPLICATION

The system in figure 5.1 represents a mixed-domain application, where a signal processing subsystem (SDF domain) exists within a discrete-event domain. Suppose that an **X** domain contains a subsystem of **Y** domain. The subsystem of **Y** domain within the outside **X** domain is implemented as a C++ object called a *Wormhole*. A Wormhole is derived from the class *Star*. It behaves exactly like any other star in the outside **X** domain. Internally, however, it encapsulates an entire foreign domain **Y** invisible from the outside universe. The internal computation model can be totally different from the external model, in that the specification language, semantics, and scheduling paradigm can be totally different. Ptolemy provides a seamless interface between domains.

Mixed domain simulation capability in Ptolemy has been demonstrated with combinations of signal processing, control, and networking (such as packet speech and video), as well as real-time telecommunications switching control software [Buc91a][Buc91b]. Domains in Ptolemy are classified into two groups: *timed* and *untimed*. In a timed domain, the scheduler keeps track of the global timing relations among tokens. On the other hand, an untimed domain has no notion of time and requires only the local ordering information of tokens. There are four combinations of timed and untimed domains. Timing management is the most challenging task in mixed-domain simulation, especially for the combination of two timed domains.

An important application of mixed domain scheduling lies in rapid prototyping.

Gabriel uses synchronous dataflow as the model of computation; SDF cannot express dynamic runtime behavior. To overcome this limitation, Ptolemy defines a new domain for dynamic constructs: the dynamic dataflow (DDF) domain. By putting this new domain inside of an SDF Wormhole, the whole application can be scheduled quasi-statically. An example is depicted in figure 5.2, where domains are identified in an SDF system with an if-then-else construct inside. The outermost topology lies in the SDF domain. The topology inside the if-then-else construct is in the DDF domain, which in turn contains two SDF domains associated with the "true" and "false" branches. Note that the if-then-else construct is realized as a SDF Wormhole, and two branches of the if-then-else constructs as DDF Wormholes.

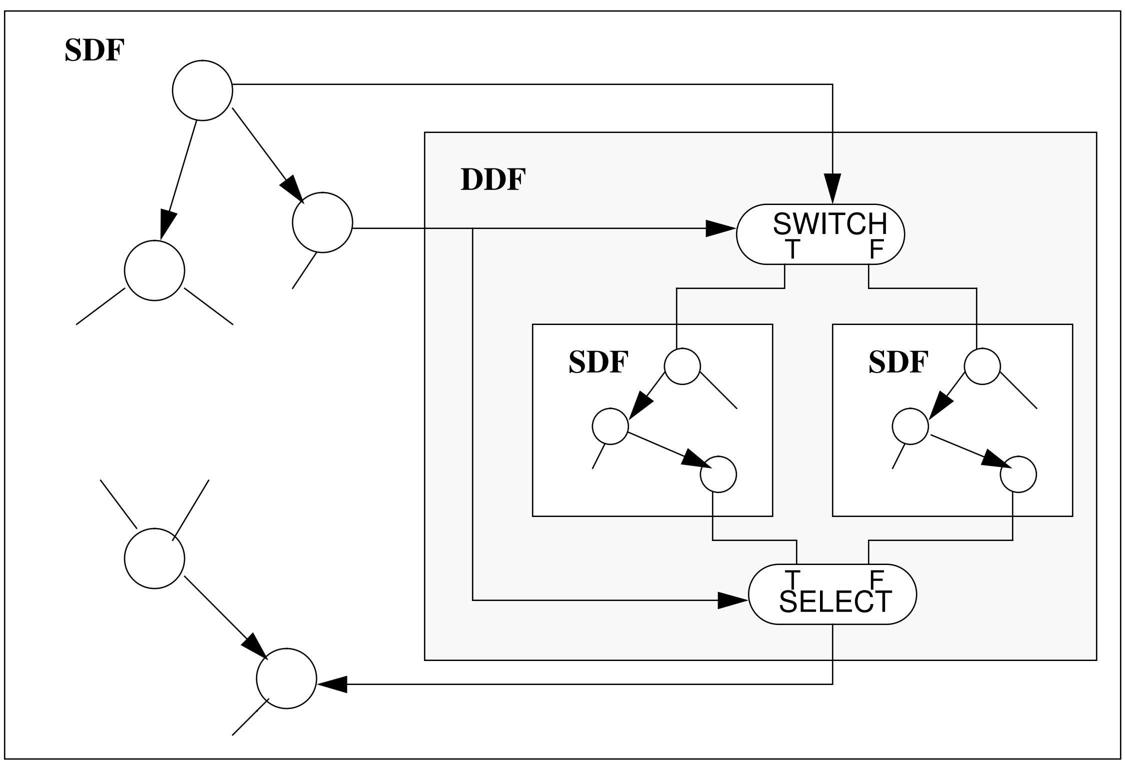


Figure 5.2 An example of a mixed domain system. The outermost level of the system is a SDF domain. A dynamic construct (if-then-else) is identified as a DDF domain, which in turn contains two SDF domains corresponding to the "true" and the "false" branches.

5.1.1 An Example

As an example of a mixed-domain (SDF and DDF) application, a simple waveform coding algorithm is screen-dumped from Ptolemy (figure 5.3). The entire universe is shown at the top level (figure 5.3 (a)). A speech waveform or image waveform is read

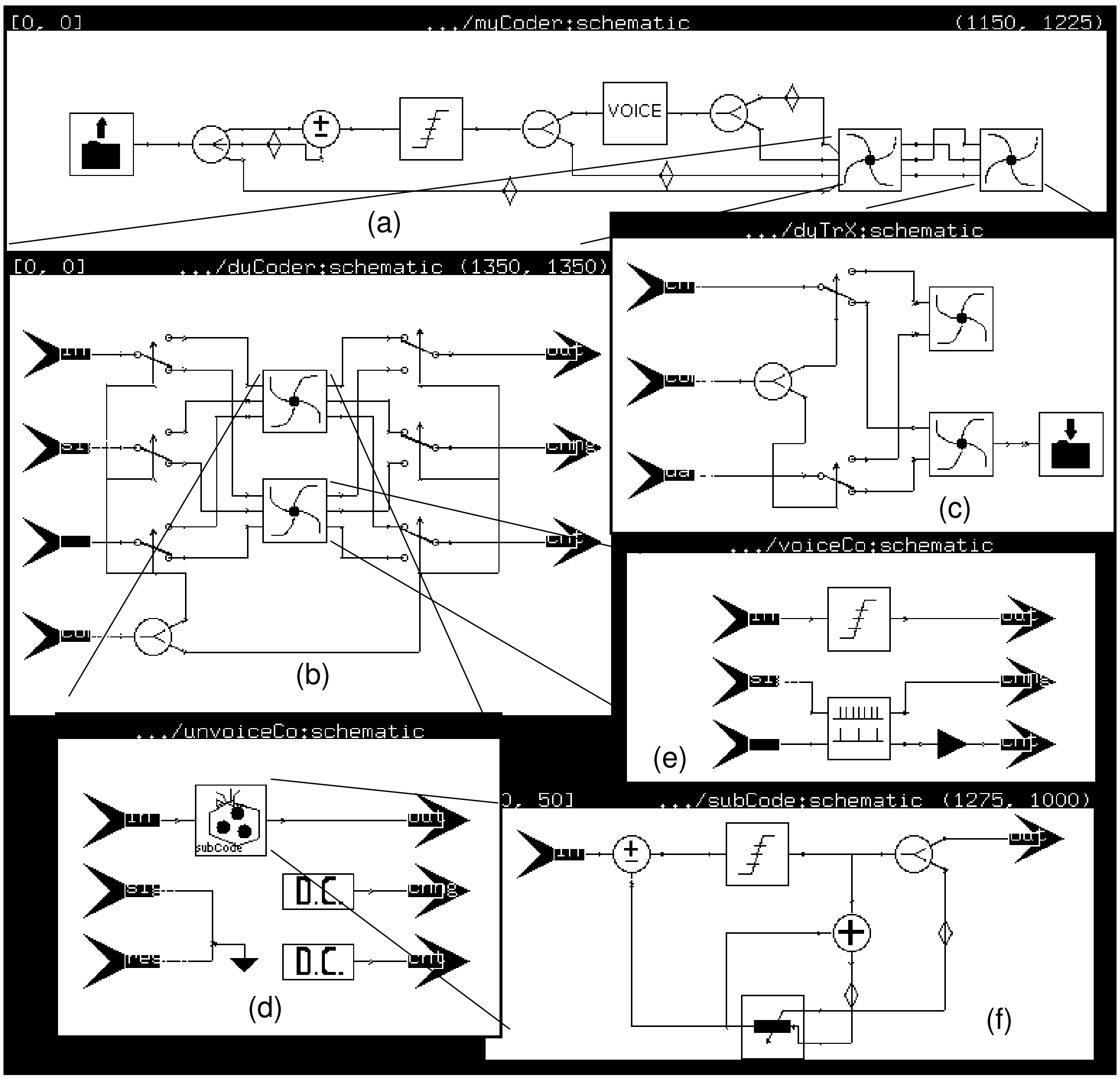


Figure 5.3 A waveform coding algorithm that uses if-then-else constructs is shown at the top level (large window). It differentiates the high-frequency parts and the low-frequency parts of the waveform and encodes them differently, using ADPCM for the high frequency parts, and using a interpolation technique for the low frequency parts. Subsystems, galaxies and wormholes, are displayed in other windows.

from the file. The slope of the waveform at the current sample is computed, and quantized into three levels: *–1 , 0*, and *1*. If the magnitude of the slope is larger than a threshold, the output of the quantizer becomes *-1* or *1* depending on the sign of the slope. By adjusting the threshold of the quantizer, we filter out some weak high frequency noisy components of the waveform. The VOICE block counts the sign changes of slopes within a fixed interval, for example within *10* samples. If the sign change of the slopes is more frequent than a threshold, the VOICE block regards the current sample as a high-frequency part of the waveform. The sample value, the current slope of the waveform, and the output from the VOICE block are fed into the first *CASE* construct (figure 5.3 (b)), which is expressed as an SDF wormhole. The “false” branch of the CASE construct encodes the high-frequency parts of the waveform by an ADPCM with high compression ratio as shown in figure 5.3 (f), since the high frequency parts are less perceptible to our eyes or ears than the low frequency parts. For the low frequency parts of the waveform, the main idea is to send the peak values and the number of samples between two peaks. At the decoder site, the intermediate samples between two peaks are linearly interpolated by a *For* construct. In figure 5.3 (e), the position of the peak values and the number of samples between two consecutive peaks are calculated by observing the slope signs. The current sample value is coarsely quantized (compression ratio 2:1). Only when the current sample value is determined as a peak value, the sample value and the number of intermediate samples between the current sample and the previous peak are transmitted, as realized by the second Case construct in figure 5.3 (c).

We simulated this universe with a speech waveform and obtained a fairly understandable replica of the original sound with good compression ratio: between 4:1 and 8:1. As a speech coder we may not want to parallelize the algorithm into a multiprocessor architecture because the entire processing can be done on a single processor within the sample period. As an image coder, we expect to improve the algorithm and implement it into a multiprocessor architecture.

5.2. SCHEDULING PROCEDURE

In this section, we outline the scheduling procedure for the system in figure 5.2. The diagram of the scheduling procedure is depicted in figure 5.4. The class names specified in figure 5.4 are different from what are actually used in Ptolemy.

The **Scheduler** class in Ptolemy is provided a *setup()* method to perform compile-time scheduling. In the body of the *setup()* method, the SDF scheduler initializes each block before performing the actual compile-time scheduling. The scheduler of the outer SDF domain regards an SDF wormhole (if-then-else construct) as an SDF actor. Therefore, the SDF wormhole is initialized before the compile-time schedule is made: **SDF Wormhole** :: *start()* method is called.

In the initialization stage, the wormhole invokes the **DDF scheduler** :: *setup()* method to initialize the inside DDF domain. Two DDF wormholes associated with the arcs of the if-then-else constructs are initialized. When a DDF wormhole is initialized, the *setup()* method of the inside SDF scheduler is called. It performs the compile-time scheduling with varying number of assigned processors from *1* to *N* (= total number of the processors). The scheduling results, which are the local schedules of the wormholes, are maintained in the **Profile** classes. Each wormhole has *N* profiles associated with *1* to *N* processors. The Profile class contains two integer arrays corresponding to the start-times and the finish-times of the local schedule on the assigned processors (figure 5.5).

After the outer SDF scheduler completes the initialization stage, it performs the compile-time scheduling based on either the basic Hu's level algorithm or Sih's dynamic level scheduling algorithm. The static level of the SDF wormhole (if-then-else) construct is chosen as the average execution time. Before the SDF wormhole is to be scheduled next, the SDF scheduler obtains the pattern of processor availability, and invokes the SDF

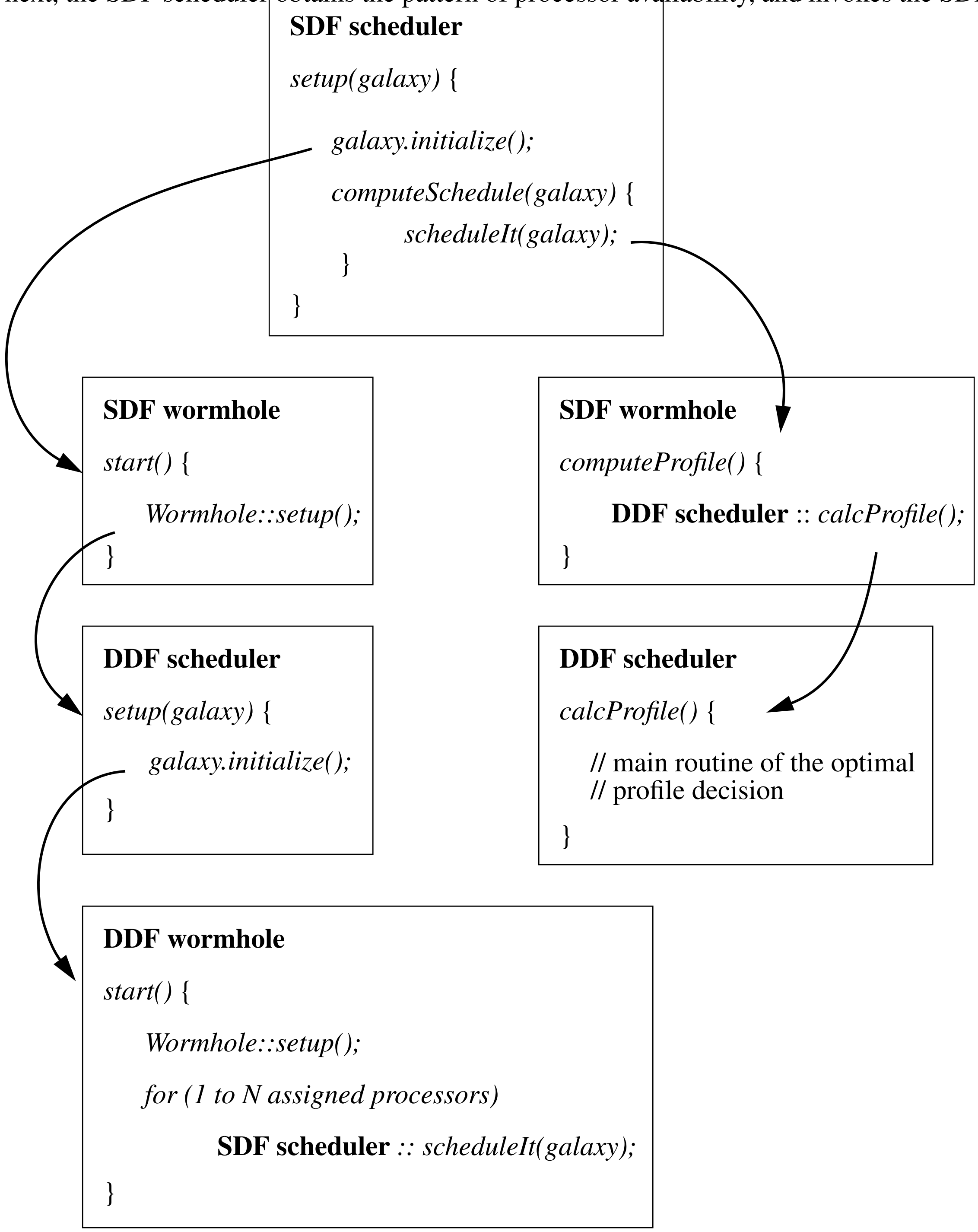


Figure 5.4 The diagram of the scheduling procedure for an SDF universe with a dynamic construct as shown in figure 5.2. It shows the order in which the scheduling methods are called. What each scheduling method does is explained in the context. Note that the class names such as “SDF scheduler” and “DDF wormhole” are different from what are actually used in Ptolemy.

Wormhole :: *computeProfile()* method. The arguments passed with this method are the number of processors, the sum of the execution lengths of the unscheduled blocks that are independent of the dynamic construct, and the pattern of processor availability. The second argument is necessary to guess the idle time caused by the dynamic construct when a profile is chosen. The SDF wormhole calls the **DDF scheduler** :: *calcProfile()* method to decide on the optimal profile: the number of processors to be assigned and the local schedule of the if-then-else construct with the assigned processors. For the details of the profile decision technique, refer to the previous chapter.

The outer SDF scheduler, then, schedules the optimal profile of the SDF wormhole as a scheduling unit. Since the optimal profile usually spans more than one processor, we need to modify the existing compile time scheduling techniques. Note that quasi-static scheduling involves several scheduling interactions between the SDF and the DDF domains, which is called “mixed-domain” scheduling.

While our discussion is based on an if-then-else construct, the same procedure is followed for other constructs such as data-dependent iteration and recursion. An SDF wormhole, which corresponds to a dynamic construct, consists of a few DDF actors and some DDF wormholes that contain the SDF domain, the body of the construct, inside. In the initialization stage, each DDF wormhole performs the compile-time scheduling of the inner SDF domain *N* times with varying numbers of assigned processors, *1* to *N*, producing *N* sets of the profiles of the body of the construct. In the compile-time scheduling stage, each SDF wormhole decides the optimal profile of the associated dynamic con-

```
class Profile {
        int           effP;          // number of assigned processors.
        IntArray      startTime;
        IntArray      finishTime;
}
```

Figure 5.5 The Profile class in Ptolemy. It shows only a few members of interest.

struct. For a nested dynamic construct, *N* optimal profiles are computed for 1 to *N* assigned processors since they are computed in the initialization stage of the outermost SDF domain.

Let us assess the complexity of the scheduling algorithm. If the number of dynamic constructs including all nested ones is *D*, the total number of profile decision steps is order of *ND*, *O(ND)*. If the number of DDF wormholes is *S*, the total number of compile-time scheduling executions is order of *NS*, *O(NS)*. Therefore, the complexity of the scheduling algorithm is *O(NS + ND)* scheduling steps (or function calls). Since the number of DDF wormholes is the same order of the number of dynamic constructs, the overall complexity is simply *O(ND)*. The memory requirements are the same order of magnitude as the number of profiles to be maintained, which is approximate to the number of the SDF and DDF wormholes. As a result, the memory requirement is also order of *ND*, *O(ND)*.

5.3. THE REPRESENTATION ISSUE

Up to now, we have assumed that the representation of a dynamic construct is completely graphical with a DDF graph. A DDF graph consists of a few DDF actors and some DDF wormholes of SDF domain that represent the body of the construct. A dynamic construct is identified by DDF actors such as SWITCH and SELECT. The DDF actors decide the data path or the amount of data consumed or produced, and thus realize the dynamic behavior of the system. Ptolemy supports this representation.

Another possible representation we can think of is the mixture of a dataflow graph and a textual description of the dynamic constructs. The body of the dynamic construct is expressed as an SDF graph. But, the dynamic construct itself is described in textual form. An example of a possible realization is shown in figure 5.6. This representation is appealing to the programmer since the textual forms of dynamic constructs are more familiar than the graphical representations.

One difficulty in this mixed representation is that we need an interpreter for the textual description of the dynamic constructs. The if-then-else construct is easy to understand, but a data-dependent iteration may be difficult to read. Sometimes, we upsample the incoming data and sometimes we count down from the incoming data to initiate the data-dependent iteration. There are also a countably infinite number of ways to terminate the data-dependent iteration. The required interpreter is likely to be complicated. In the graphical representation, we handle this problem by defining new DDF stars as necessary.

A major problem in the graphical representation is the problem of identifying a specific dynamic construct. Currently, we predefine the topologies of the dynamic constructs that Ptolemy supports. The predefined topologies, however, are very restricted since a dynamic construct is identified by comparing a given topology with the predefined topologies to find a match. Assessing the trade-offs of these two representations

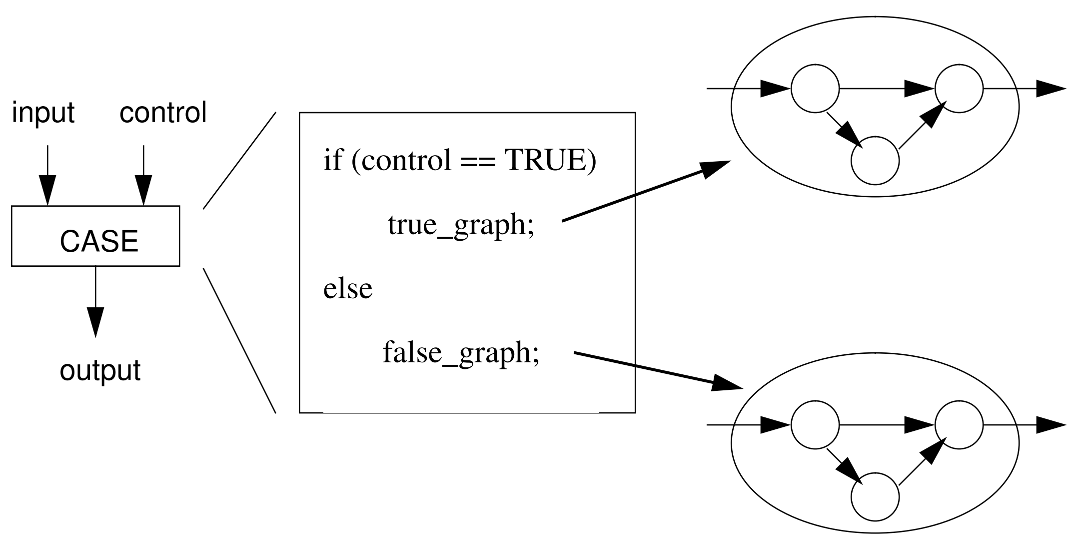


Figure 5.6 A mixed representation of a dynamic construct: if-then-else. The body of the dynamic construct is expressed as SDF graphs, but the dynamic construct itself is described in a textual form.

and determining which one is better is an open problem.

6

EXPERIMENTS

But I say to you, do not swear at all: neither by heaven ......

But, let your 'Yes' be 'Yes,' and your 'No,' 'No.' For

whatever is more than these is from the evil one.

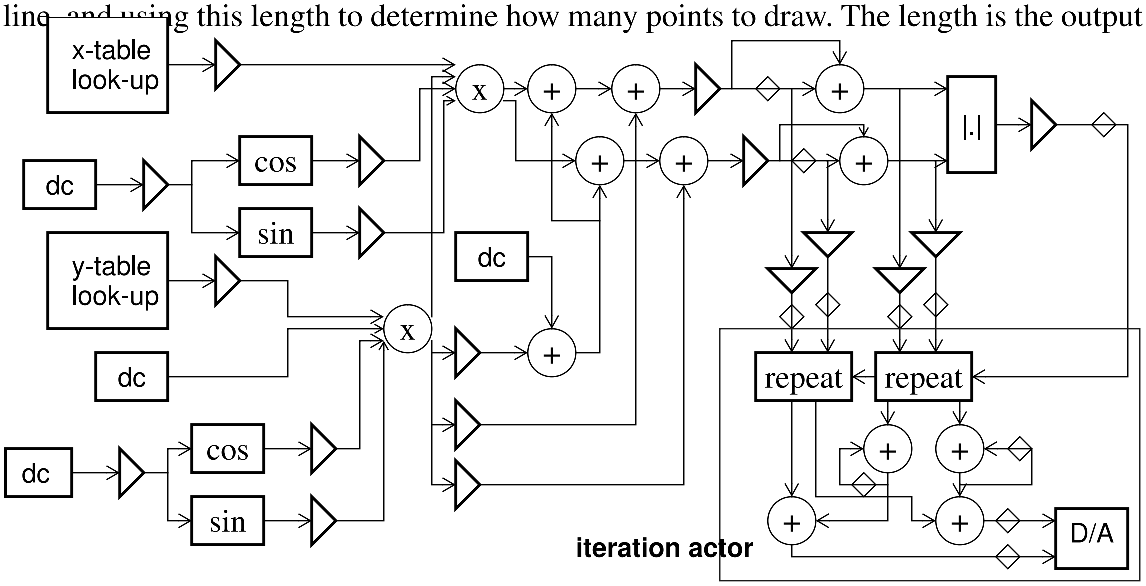
--- Matthew 5:34,37

In this chapter, we demonstrate the proposed quasi-static scheduling techniques with several examples. These experiments do not serve as a full test or proof of generality of the technique. However, they will verify that the proposed technique can make better scheduling decisions than other simple but ad-hoc decisions on dynamic constructs in many applications. In Ptolemy, where the proposed technique is implemented, we may collect a library of scheduling techniques, each of which performs effectively for a certain class of applications. Since it is highly doubtful that any single scheduling technique is effective for all applications, this approach of providing multiple scheduling techniques in a system would be useful, as envisioned by G. Sih [Sih91].

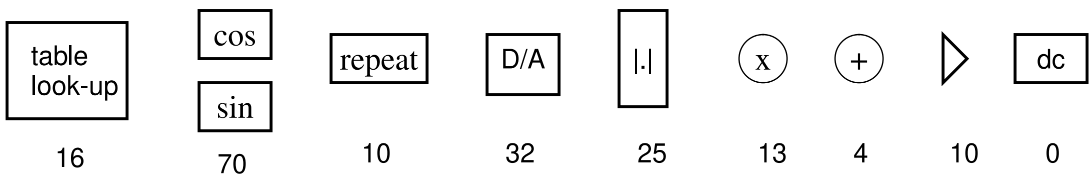
6.1. AN EXAMPLE FROM GRAPHICS

We can illustrate our method with an application from graphics in which a geometric shape is displayed and rotated in three dimensions, with perspective. This is an attractive application because the program is simple, and can be written with or without iteration, and the iteration can be data-dependent, or not. We can compare quite a variety of realizations. Not surprisingly, we find that using data-dependent iteration considerably decreases the total amount of computation compared to programs that avoid data-dependent iteration. Furthermore, when we use data-dependent iteration, our scheduling method results in a program that is only *3%* slower than the best that can be expected from dynamic scheduling, for this example. We are comparing against a hopelessly optimistic model of dynamic scheduling, so with a realistic model, our method would yield a program that is considerably faster. The target architecture is a shared-memory multiprocessor with four programmable DSP microcomputers (Motorola DSP56001's).

The dataflow graph for the program using data-dependent iteration is shown in figure 6.1. This graph is similar to an implementation we have constructed using the Gabriel signal processing environment [Lee89a], with the major difference that it uses data-dependent iteration. It works as follows: two table-lookup actors supply the x and y coordinates of the vertices of the geometric shape. A z coordinate could also be supplied, but our example assumes this is constant. Constants are supplied by the actors labeled "dc". The x,y, and z coordinates are rotated along two axes by multiplying pairwise by two complex exponentials, generated by computing sines and cosines. Next, perspective is added by using the z coordinate to modify the x and y coordinates according to a set of parameters that indicate the location of the vanishing point. The result is two coordinates only, since the image has now been mapped onto two dimensions. The last step is to draw a line between two successive vertices. This is done by first computing the length of the line, and using this length to determine how many points to draw. The length is the output



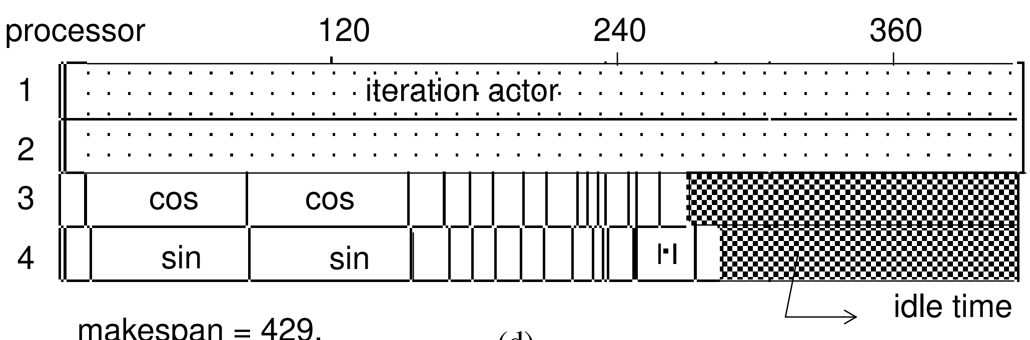
(a)



(b)

Number of iteration processors	1	2	3	4
Total cost of iteration	2255	2080	2362	2432

(c)



(d)

Figure 6.1 (a) A dataflow graph of a program that will display a rotating geometric shape in three dimensions with perspective. (b) The execution time of each actor is given in Motorola 56000 instruction cycles (currently 75ns). (c) The total cost of the iteration as a function of the number of processors assigned to the iteration. (d) One of the schedules produced by the method given in this thesis.

of the magnitude actor in the upper right of figure 6.1 (a). Since the number of points drawn depends on the length of the line, we need data-dependent iteration. The length, scaled by an empirically determined constant, serves as the control input for two "repeat" actors, which are special cases of the "upsample" actors. These actors simply repeat the tokens at their data inputs a number of times given by the control input. The right repeat actor has a vector input giving the direction of the line to be drawn. The input to the left repeat actor is the location of the position of the start of the line. In our lab, a D/A drives a vector display with an analog signal, but a bit-mapped display could easily replace this. The overall dataflow graph is repeated in an infinite iteration, thus refreshing the display continually.

A key observation is that without data-dependent iteration, the implementation requires applying the rotation and perspective operators to every point, rather than just the vertices. Since most of the computation is in these transformations, the cost is high. For a particular test shape (a block-lettered "G"), we determined that the implementation that avoided iteration required an average of 2581 instruction cycles (on four processors) to draw one line. This is the first entry in figure 6.2. Programs using iteration are much less expensive.

In figure 6.1 (b), the execution time of each actor is given in Motorola 56000 instruction cycles (currently 75 nsec). These are not ideal implementations of the actors, but they are working implementations in the Gabriel system. Suppose that we have 4 processors. Then, we may assign $1 \le N \le 4$ processors to the data-dependent iteration actor. To make the best decision on how many processors are assigned to the actor, we check the total cost of the iteration as a function of the number of assigned processors, as explained in chapter 4. For each N, we calculate the assumed execution time and corresponding expected run-time cost assuming a given probability mass function of the length of line segment. The cost of iteration is shown in figure 6.1 (c), and the schedule is shown in figure 6.1 (d). Here we assume a geometric distribution with *MIN=0* and *q=0.95*. In the actual scheduling process, the number *N* greater than *3* is not considered at all since the makespan of the subgraph within the iteration actor is not shortened with more than *2* processors. In this manner, the search space for *N* can often be reduced significantly. From the numbers in figure 6.1 (c), we choose to assign *N=2* processors to the iteration. After the decision is made, we can construct the global schedule (figure 6.1 (d)). This Gantt chart shows the assumed length of the iteration as a shaded region.

To make the program more parallelizable, we retimed the graph in front of the data-dependent iteration actor. This is perfectly reasonable for this application, and can be automated [Lei83]. With the specific example we used, we achieved reasonably high processor utilization (*82.6%*) and low makespan (*429* cycles). Of course, at execution time, the number of cycles of the iteration will vary, so the performance will vary. Since there is idle time right at the end of non-iteration processors due to the iteration actor, we

	Average number of cycles to draw 1 line
Fully-static without iteration	2581
Fully-static with worst case iteration	1293
Quasi-static (geometric, $q=0.9$: $x = 6$)	735
Quasi-static (geometric, $q=0.95$: $x = 13$)	672
Quasi-static (exact distribution: $x= 20$)	672
Fully-dynamic (ideal without overhead)	657

Figure 6.2 The performance comparison among several scheduling decisions. The performance is measured by the average number of cycles to draw one line.

expect that the schedule is not optimal. However, it is certainly near optimal in this case.

The major question that remains unanswered is how to determine which stochastic model fits an iteration best. Our choice here of a geometric model with $q=0.95$ (the probability of continuing is 0.95), is probably not very accurate. We applied the program to a simple geometric shape (letter "G") in order to compare the runtime performance with several scheduling decisions (see figure 6.2). The performance is measured by the average number of cycles to draw one line and depends on the specific shape being drawn.

As discussed earlier, fully-static scheduling without iteration gives the worst result. Another method that can use fully-static scheduling is to perform the maximum number of iteration cycles every time, which gives the second worst result, as shown in the second row of figure 6.2. Next we approximate the runtime statistics using geometric distribution with two different parameters: $q=0.9$ and $q=0.95$. The first value grossly underestimates the average length of the lines drawn. The second value results is a probability mass function with the appropriate mean but the wrong shape. For the fifth experiment shown in figure 6.2, we use exactly the correct probability mass function, computed by histograming the lengths of the lines in the geometric shape being drawn. In the sixth experiment, we calculate the performance for fully-dynamic scheduling, ignoring overhead.

The results are remarkable. Using the exact probability mass function we are within *3%* of the best that can be expected from fully dynamic scheduling, for this program (fifth line, figure 6.2). Using a function with the right mean but the wrong shape, the result is identical (fourth line, figure 6.2). Using a function with the wrong mean and the wrong shape, we are still within *12%* of the best that can be expected from fully dynamic scheduling (third line, figure 6.2).

These results are particularly promising because we are comparing against a fully-dynamic scheduling strategy that is far more sophisticated than what would be practical, and we are ignoring the scheduling overhead. Specifically, we assume the dynamic scheduler somehow knows how many cycles of the iteration will be executed before each cycle of the overall program begins. It then uses a critical path method (Hu-level scheduling) to construct a schedule for this number of cycles. Since practical dynamic scheduling algorithms are much more primitive, we view the performance of this algorithm as a bound on the performance of all dynamic schedulers. When we count the runtime overhead, the fully-dynamic scheduling will be abandoned without hesitation for this example.

The promising results for this program should be viewed only as promising results based on one example. We are developing a programming environment that will permit much more extensive experimentation with practical programs; only after those experiments are complete will we know just how general this method is. Nonetheless, the experiments we have done show that with a good stochastic model for the iteration, at least some programs will get schedules that are about as good as can be expected in practice. They also show that the scheduling method depends on the validity of the stochastic model for the iteration. However, we make the very preliminary postulate that the performance of the technique will not be highly sensitive to the stochastic model since even a crude model might give a near-optimal number for the iteration cycles. This can only be verified by trying many examples, something that requires first developing much more infrastructure. Should the sensitivity prove to be greater, then we can envision successive refinements of the schedule based on observations of the executing program.

6.2. SYNTHETIC EXAMPLES

In this section, we demonstrate several synthetic examples to test the effectiveness of the proposed scheduling technique. These examples are randomly created. We have sometimes increased the parallelism of graphs by pipelining. We assume that the statistical information of dynamic behavior in the dynamic constructs is already known. The target architecture assumed is a shared bus multiprocessor containing five processors, in which communication can be overlapped with computation. The communication cost is assumed fixed at *2* time units. The execution length of each block is assigned randomly.

To test the scheduling effectiveness of the proposed quasi-static scheduling technique, we compare it with the following scheduling alternatives for the dynamic constructs:

Method 1. Assign all processors to each dynamic construct.

Method 2. Assign only one processor to each dynamic construct.

Method 3. Apply a fully dynamic scheduling ignoring all overheads.

Method 4. Apply a fully static scheduling.

Method 1 corresponds to the previous research on quasi-static scheduling technique made by E. Lee [Lee88] and Loeffler et. al [Loe88]. Method 2 approximately models the situation when we implement each dynamic construct as a single big atomic actor. Then, the whole application lies in the SDF domain, and we can schedule very efficiently. We take the average execution time of the dynamic behavior as the runtime of the big actor for the compile time scheduling. To simulate the third method, we list all possible outcomes, each of which can be represented as an SDF galaxy, of a dynamic construct. With each possible outcome, we replace the dynamic construct, and apply a fully static scheduling algorithm: dynamic level scheduling algorithm by Gil Sih [Sih91]. The scheduling result from Method 3 is non-realistic since it ignores all overheads of the fully dynamic scheduling strategy. Nonetheless it will give a yardstick to measure the relative performance of other scheduling decisions. By modifying the dataflow graphs, we may use fully static scheduling in Method 4. For a Case construct, we evaluate all branches and select one by an SDF star, MUX (multiplexer). For a data-dependent iteration construct, we always perform the worst case of iteration and select one. For comparison, we use the average makespan of the program as the performance measure. Furthermore, we assume the self-timed scheduling strategy for the runtime execution model.

6.2.1 An Example With A Case Construct.

Figure 6.3 shows an example with a *Case* construct at the top level, which is displayed in the large window. It consists of SDF stars and an SDF wormhole of DDF domain for the Case construct; it lies in the SDF domain as a whole. The Case construct and two subsystems that correspond to two branches are expanded in other windows. The execution length of each SDF star is specified beneath the star icon. Note that the dataflow graph is pipelined at the end of the Case construct to increase the parallelism of the graph by putting delays on the output arc.

We obtained the scheduling result through the proposed technique in a Gantt chart in figure 6.4, which is a dumped screen from Ptolemy. We assumed that there are 4 processors and the probability of taking the "true" branch is 0.5, equally likely to taking the "false" branch. In this example, the optimal profile of the Case construct is equal to that of E. Lee's proposal: overlap the local schedules of the both branches and choose the maximum termination for each processor with the given number of processors, *3*. Hence, the average makespan is same as the schedule period, that is *47*. The optimum number of assigned processors in this example is *3*, with which the expected total cost of the Case construct defined in chapter 4 is minimized (table 6.1). Recall that the expected total cost

Table 6.1: The expected total cost of the Case construct with the number of assigned processors.

Number of Assigned Processors	1	2	3	4
Expected Total Cost	156	96	88	N/A

includes the idle time due to the mismatched pattern of processor availability at the beginning of the construct as well as the idle time due to the absence of runnable actors to be scheduled on the unassigned processors during the construct execution. Since neither branch can use all *4* processors, we don't have to consider assigning *4* processors to the construct.

The average makespans obtained with Methods 1 to 4 are listed in table 6.2. In

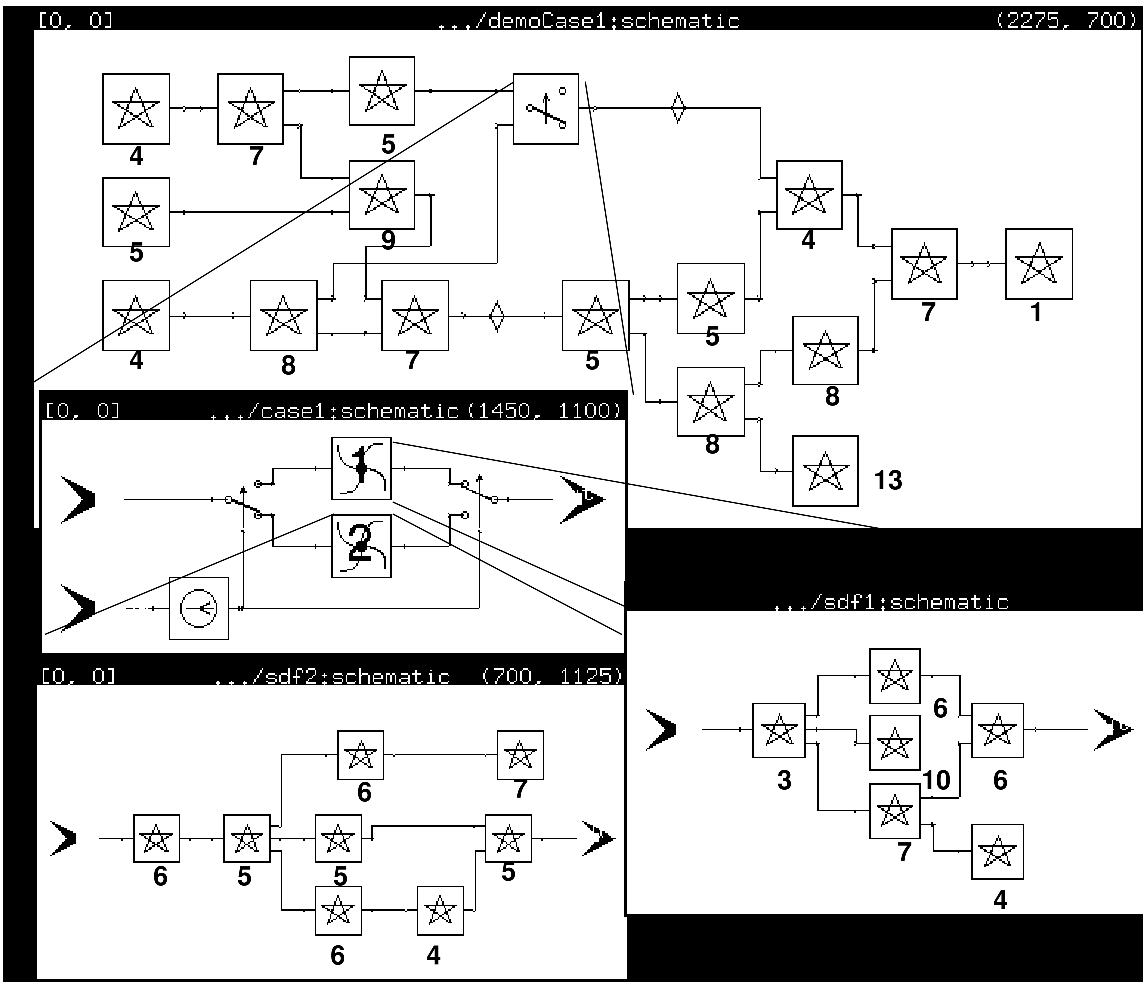


Figure 6.3 An example with a *Case* construct at the top level (large window). The subsystems associated with the *Case* construct are also displayed. This is a screen dump of the Ptolemy environment. The left bottom galaxy represents the "true" branch and the right bottom galaxy represents the "false" branch.

Table 6.2: Performance comparison among several scheduling decisions.

Method	Proposed	1	2	3	4
Avg. Makespan	47	47	60	40.5	51
% of ideal	0.86	0.86	0.675	1	0.79

this example, the proposed technique achieves *86%* of the ideal makespan obtained by Method 3. Method 1 bears the same result as the proposed one because the Case construct can not use all *4* processors, so it uses *3* processors only and leaves one processor idle. Note that assigning the construct onto a single processor produces the worst result since it fails to exploit the parallelism inside the Case construct. Since both branches do not fully utilize the assigned processors, the proposed technique does not overwhelm the static scheduling choice (Method 4), but still shows about *10%* improvement.

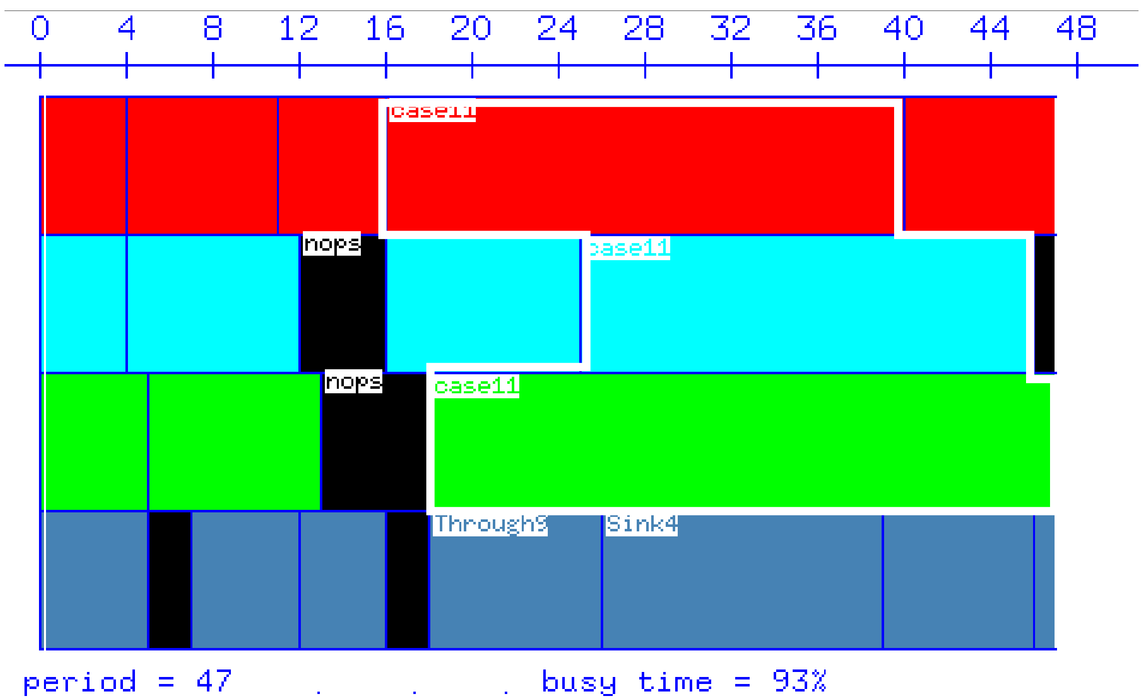


Figure 6.4 A Gantt chart display of the scheduling result over 4 processors from the proposed scheduling technique for the example in figure 6.3. The profile of the Case construct can be observed in the first three rows. We assume that the probability of taking the true branch is 0.5. The minimum period and the maximum busy time should be ignored throughout this thesis.

6.2.2 An Example With A For Construct

An example with a *For* construct is shown in figure 6.5. We make the randomly assigned execution lengths of the SDF stars at the top level statistically bigger than those inside the For construct; by doing so, we somewhat balance the average execution length of the For construct and the sum of the execution lengths of the SDF stars. To increase the parallelism of the example, we pipelined the graph at the beginning of the For construct.

The scheduling decisions to be made for the For construct are how many processors to be assigned to the iteration body and how many iteration cycles to be scheduled

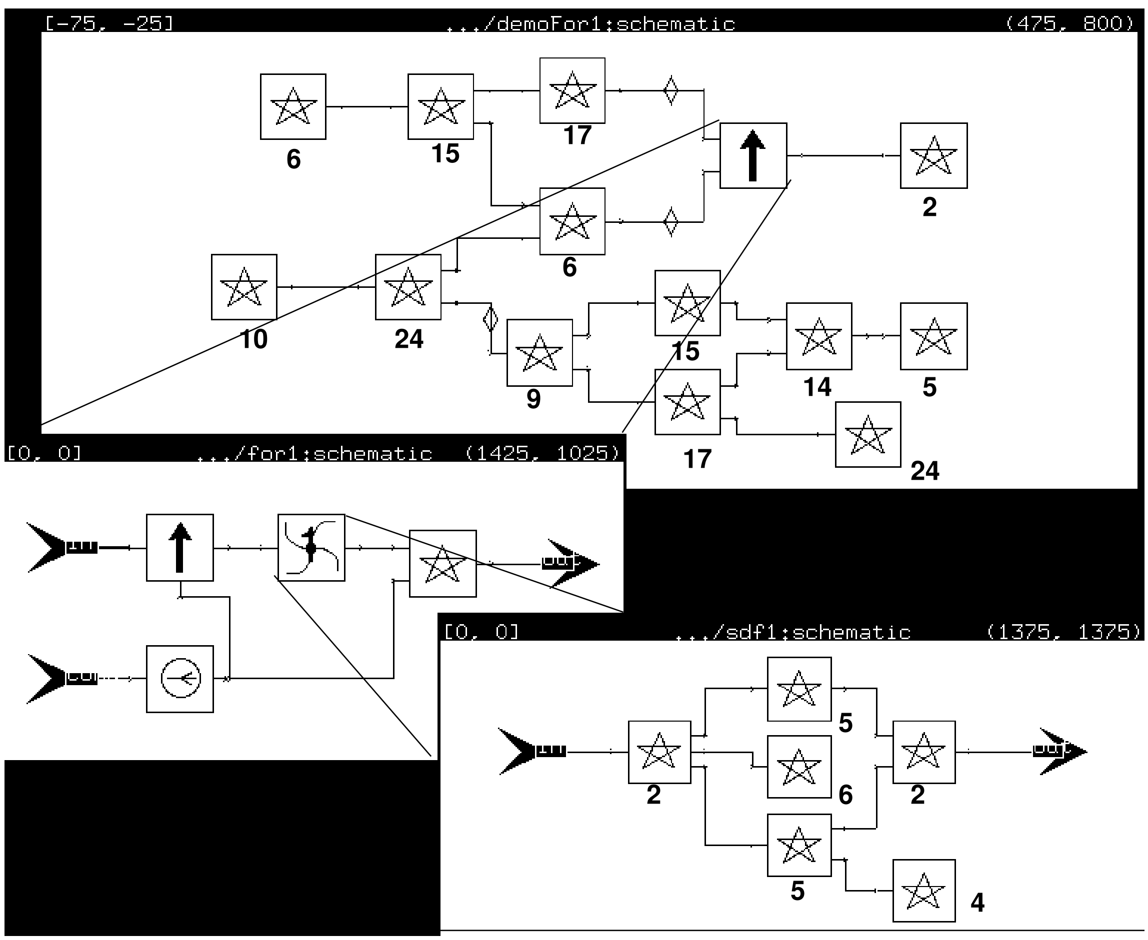


Figure 6.5 An example with a *For* construct at the top level (large window). The subsystems associated with the *For* construct are also displayed.

Table 6.3: The expected total cost of the For construct as a function of the number of assigned processors.

Number of Assigned Processors	1	2	3	4
Expected Total Cost	129.9	135.9	177.9	N/A

explicitly. We assume that the number of iteration cycles is uniformly distributed between *1* and *7*. To determine the optimal number of assigned processors, we compare the expected total cost as shown in table 6.3. The total number of processors is again assumed *4*. Since the iteration body can utilize two processors effectively, the expected total cost of the first two columns are very close. However, the schedule determines that assigning one processor is slightly better. Rather than parallelizing the iteration body, the scheduler automatically parallelizes the iteration cycles themselves. If we change the parameters, we may want to parallelize the iteration body first and the iteration cycles next. The proposed technique considers this trade-off when determining the optimal number of assigned processors. The resulting Gantt chart for this example is shown in figure 6.6. The profile of the For construct is identified with the white bold line.

If the number of iteration cycles is *1*, *2*, or *3*, the makespan of the example is same as the schedule period, *66*. If it is greater than *3*, the makespan will increase by the execution length of the iteration body, which is *24*. Therefore, the average makespan of the example becomes 79.7. The average makespans from other scheduling decisions are compared in table 6.4. The proposed technique outperforms other realistic methods and achieves 85% of the ideal makespan by Method 3. In this example, assigning *3* processors to the iteration body (Method 1) worsens the performance since it fails to exploit the intercycle parallelism. Confining the dynamic construct in a single big actor (Method 2) gives the worst performance as expected since it fails to exploit either intercycle parallelism, compared to the proposed technique, and the parallelism of the iteration body, compared to Method 1. Assuming the worst case iteration in Method 4 is not bad in this

Table 6.4: Performance comparison among several scheduling decisions

Method	Proposed	1	2	3	4
Avg. Makespan	79.7	90.9	104.3	68.1	90
% of ideal	0.85	0.75	0.65	1	0.76

example. As the distribution of the number of the iteration cycles is broader, Method 4 is expected to be much worse than the proposed technique and even Method 1.

This example reveals a shortcoming of the proposed technique. If we assign 2 processors to the iteration body and exploit the intercycle parallelism, the average

Table 6.5: Performance comparison among several choices on the assumed number of iteration cycles for the optimal profile.

The assumed number	1	2	3	4

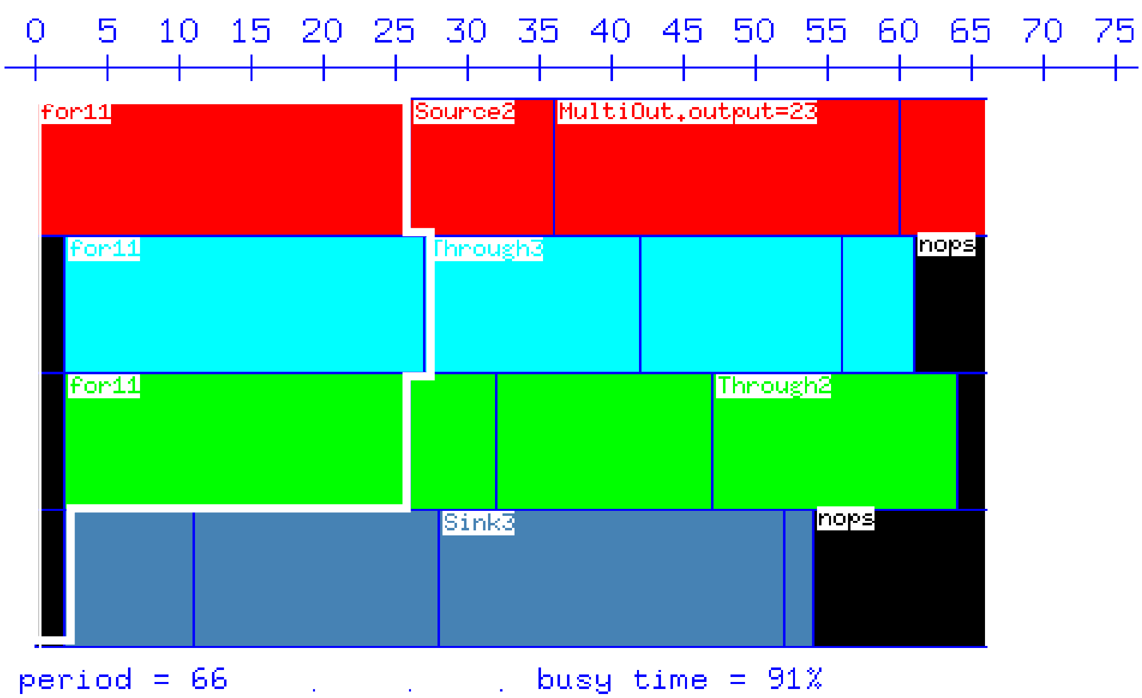


Figure 6.6 A Gantt chart display of the scheduling result over 4 processors from the proposed scheduling technique for the example in figure 6.5. The profile of the For construct is identified. We assume that the number of iteration cycles is uniformly distributed between 1 and 7.

Table 6.5: Performance comparison among several choices on the assumed number of iteration cycles for the optimal profile.

Average makespan	80.4	78.6	79.7	86.3
Processor utilization	90%	93%	91%	87%

makespan proves 77.7 to be which is slightly better than the scheduling result by the proposed technique. When we calculate the expected total cost to decide the optimal number of processors to assign to the iteration body, we do not account for the global effect of the decision. Since the difference of the expected total costs between the proposed technique and the best scheduling was insignificant as shown in table 6.3, this non-optimality of the proposed technique could be anticipated. Actually, the best scheduling utilizes 95% of the processors, while we utilize 91% of the processors (figure 6.6). To improve the performance of the proposed technique, we can add a heuristic that if the expected total cost is not significantly greater than the optimal one, we perform the scheduling with that assigned number of the processors to the iteration body, compare the performance with the proposed technique, and take the best scheduling result.

The search for the assumed number of iteration cycles for the optimal profile is not faultless either, since the proposed technique finds a local optimum. The proposed technique selects *3* as the assumed number of iteration cycles as shown in figure 6.6. The comparison of scheduling performances by varying the assumed number of iteration cycles is illustrated in table 6.5. The best assumed number proves to be *2*, not *3* in this example, due to the higher processor utilization.

Although the proposed technique is not always optimal, it is certainly better than any of the other scheduling methods compared against as demonstrated in table 6.4. To overcome the limitations of the proposed technique, we could postpone the decision of the optimal profile. Instead, we could select a few candidates for the optimal profile, and compare the final schedule results to choose the best. This will, however, complicate the scheduling somewhat, especially when the program contains nested dynamic constructs.

6.2.3 An Example With A DoWhile Construct

In the proposed technique, a *Do-While* loop is indistinguishable from a For loop except that intercycle parallelism does not exist. So, we only determine the number of assigned processors to the loop body and the assumed number of iteration cycles. An example is shown in figure 6.7.

We assume that the minimum number of iteration cycles is *1* and the maximum

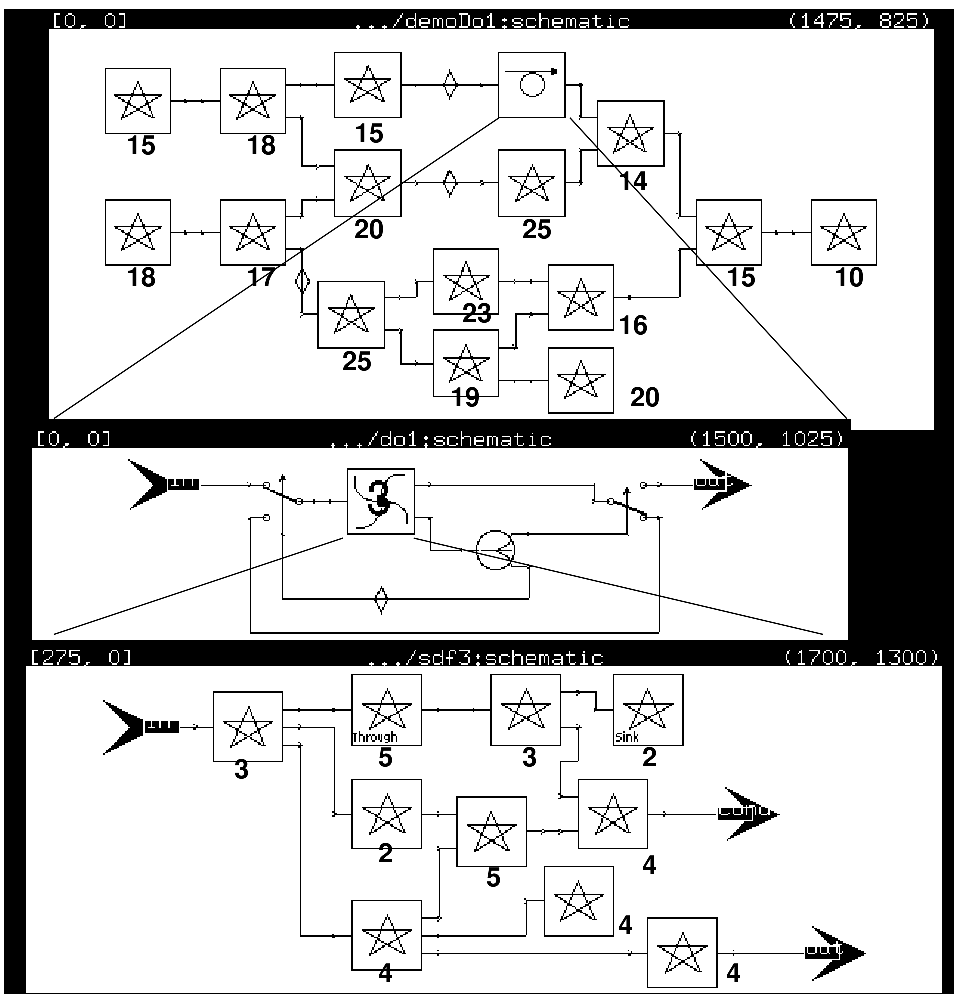


Figure 6.7 An example with a *DoWhile* construct at the top level (large window). The subsystems associated with the *DoWhile* construct are also displayed.

Table 6.6: The expected total cost of the For construct as a function of the number of assigned processors

Number of Assigned Processors	1	2	3	4
Expected Total Cost	350.6	229.2	240.3	261.3

number is *13*. We assume that the distribution of the number of iteration cycles between these two bounds can be approximated by a geometric distribution with parameter *0.7*: the probability of doing one more iteration is *0.7* after finishing the current iteration cycle.The total number of processors is again *4*. To obtain the optimal number of assigned processors, we compute the expected total cost for each assignment (table 6.6). The optimal number we obtain is *2*. The scheduling result from the proposed technique is displayed in figure 6.8.

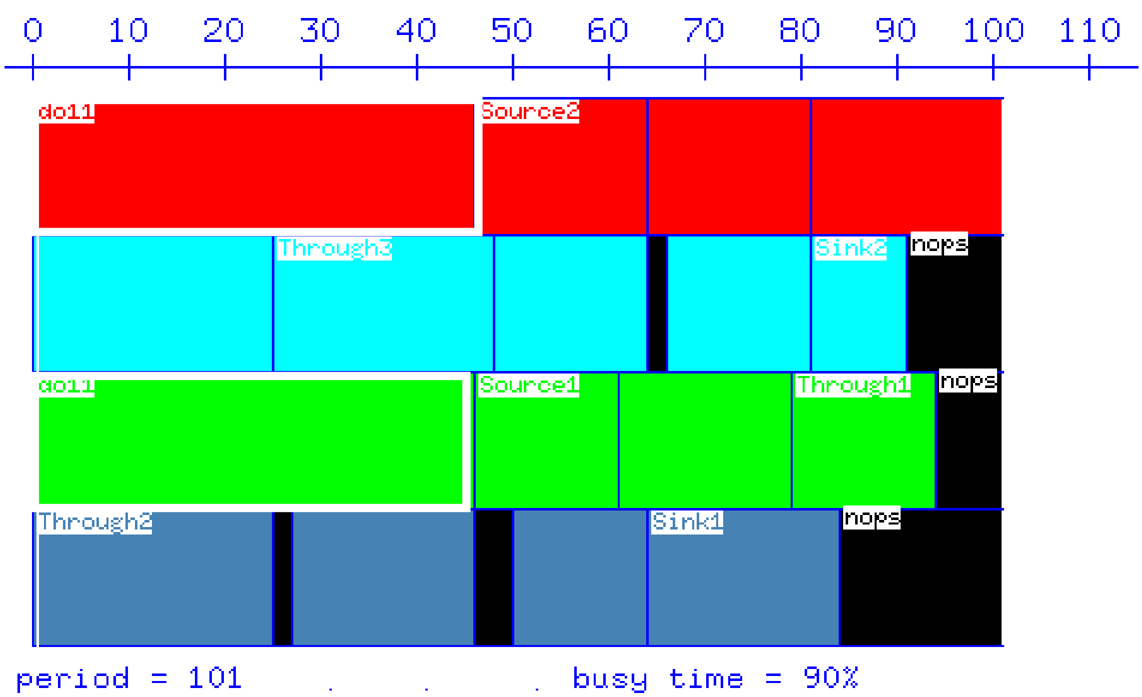


Figure 6.8 Gantt chart display of the scheduling result over 4 processors from the proposed scheduling technique for the example in figure 6.7. The profile of the DoWhile construct is identified. We assume that the number of iteration cycles is geometrically distributed with parameter *0.7*.

The average makespan obtained from the proposed technique becomes

Table 6.7: Performance comparison among several scheduling decisions

Method	Proposed	1	2	3	4
Avg. Makespan	135.4	155.9	162.2	113.4	286
% of ideal	0.84	0.73	0.70	1	0.40

$$101 + 23 \sum_{i=3}^{13} p^{i-1}(1-p)(i-2) = 135.4 \quad ,$$

where $p = 0.7$. The scheduled makespan is *101*, and the execution length of the loop body is *23* onto *2* processors in the above formula. The performance comparison with other scheduling decisions is shown in table 6.7. The makespan obtained from the proposed technique is 16% shy of the ideal makespan. Method 1 assigns all 4 processors to the DoWhile construct and does not utilize them effectively. Since the maximum number of the iteration cycle is large, Method 4 gives the worst performance as expected. The proposed technique outperforms all other realistic methods by at least *10%* of the ideal makespan in this example. If we assign *3* processors to the construct, the average makespan becomes *137.9*, which is very close to what we achieve from the proposed technique. This is not a surprising observation because the difference of the expected total costs in table 6.6 is not significant between the second and the third column.

6.2.4 An Example With A Recursion construct.

An example with a *Recursion* construct is displayed in figure 6.9. The recursion body is simple compared with the outside SDF domain to prevent the recursion construct from being dominant in the runtime profile of the program. The *width* of the recursion construct is *2*. We assume that the depth of the recursion is uniformly distributed between *1* and *4*. The total number of processors is 5. To determine the optimal profile of the

Recur construct, we have to select the number of assigned processors, the *degree* of parallelism and the assumed depth of recursion. We may assign *1* or *2* processors to the recursion body. The expected total costs when assigning *1* and *2* processors are *311* and *490* respectively. Therefore, the optimal number of assigned processors is *1*. The schedule resulting from the proposed technique is displayed in figure 6.10.

The assumed depth of recursion is taken as the same value as the degree of parallelism, which is *2* in this example. The average makespan obtained from the proposed

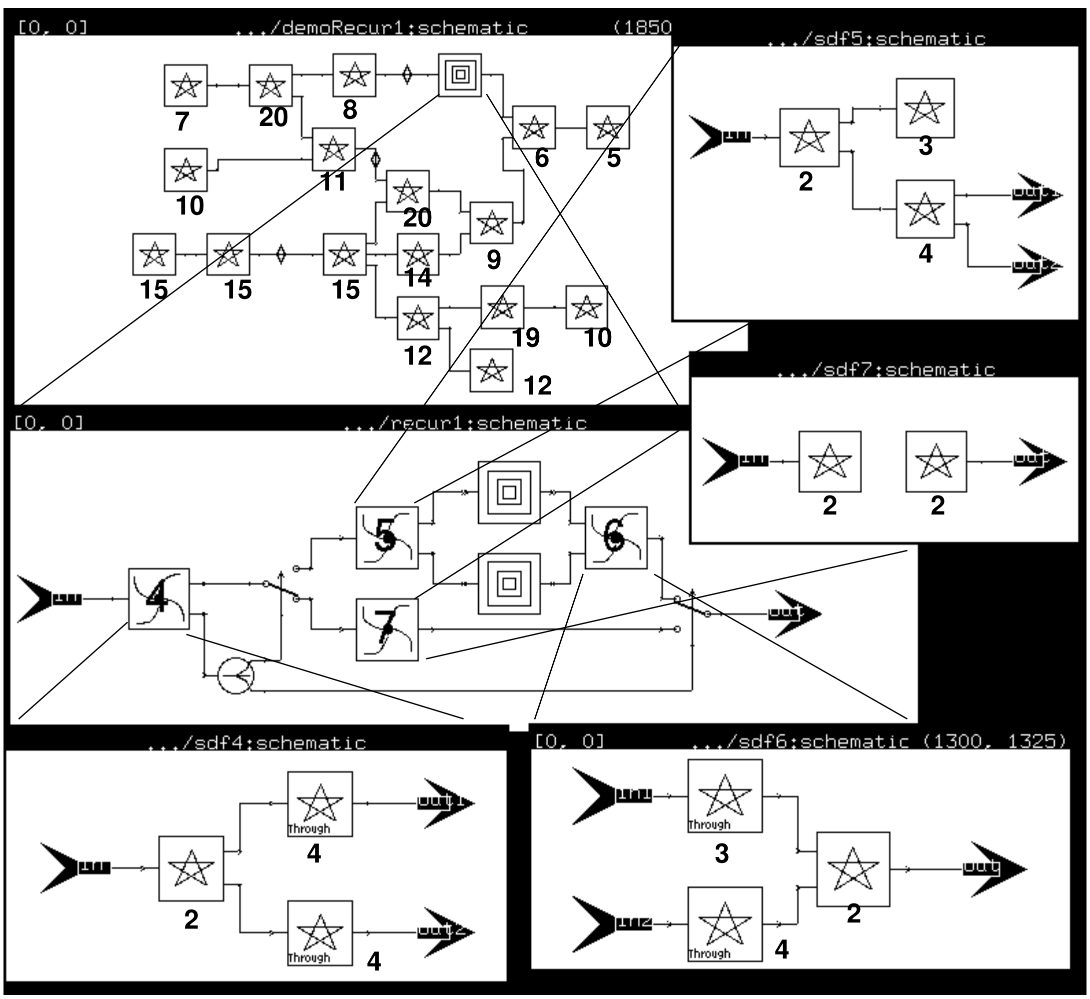


Figure 6.9 An example with a *Recursion* construct at the top level (large window). The subsystems associated with the *Recursion* construct are also displayed in other windows.

technique becomes $\dfrac{89+89+131+215}{4} = 131$. It is compared with the average

Table 6.8: Performance comparison among several scheduling decisions

Method	Proposed	1	2	3	4
Avg. Makespan	131	155	222.5	114.5	202
% of ideal	0.87	0.74	0.51	1	0.57

makespans obtained from other methods in the following table (table 6.8). The proposed technique achieves 87% of the non-realistic ideal makespan from Method 3. In Method 1, we assign *2* processors to the recursion body, and 4 processors to the recursion construct in total since the degree of parallelism becomes 1. Note that both devoting only one processor to the recursion construct (Method 2) and executing the worst case of recursion

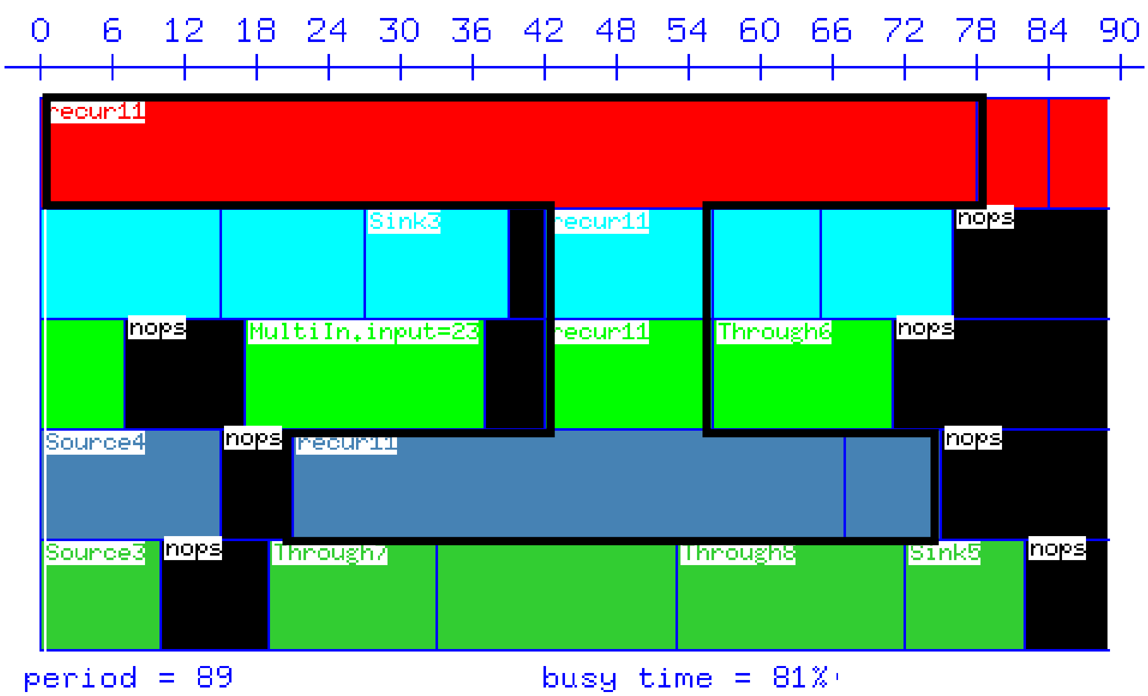


Figure 6.10 Gantt chart display of the scheduling result over *5* processors from the proposed scheduling technique for the example in figure 6.9. The profile of the Recur construct is identified. We assume that the depth of recursion is uniformly distributed between *1* and *4*.

structure (Method 4) perform very poorly. It is because the deviation of the runtime execution length of the recursion construct is huge. The proposed technique shows drastic performance improvement over other methods.

6.2.5 An Example With A Nested Dynamic Construct

As the final synthetic example, we think of an example in which a Case construct contains a For construct in its "false" branch (figure 6.11). The structure of the outer system is same as the example in figure 6.3. The "false" branch of the Case construct in figure 6.3 is replaced with a subsystem that contains a For construct. In this example, we do not compare the performance of the proposed technique with other methods. Based on the comparison results in the preceding subsections, it is evident that the other methods (except Method 3) perform more poorly as the degree of non-determinism increases. We just show how the proposed technique handles nested dynamic constructs.

In a nested dynamic construct, the compile-time profile of the inner dynamic construct affects that of the outer dynamic construct. In general, there is a trade-off between exploiting parallelism of the inner dynamic construct first and that of the outer construct first. The proposed technique resolves this conflict automatically. The scheduling result on this example is displayed in figure 6.12. We observe that the Case construct is assigned all *4* processors in figure 6.12 (a). Hence, the "false" branch of the Case construct is scheduled over *4* processors in figure 6.12 (b). The For construct inside the "false" branch is scheduled using all *4* processors again, but assigning *2* processors to the iteration body and overlapping *2* iteration cycles. Note that the compile-time profile of the "false" branch does not match the profile of the Case construct. In this example, the probability of taking the "true" branch, *0.7*, is higher than that of taking the "false" branch, *0.3*. Therefore, the optimal profile is not the maximum of the profiles of two branches.

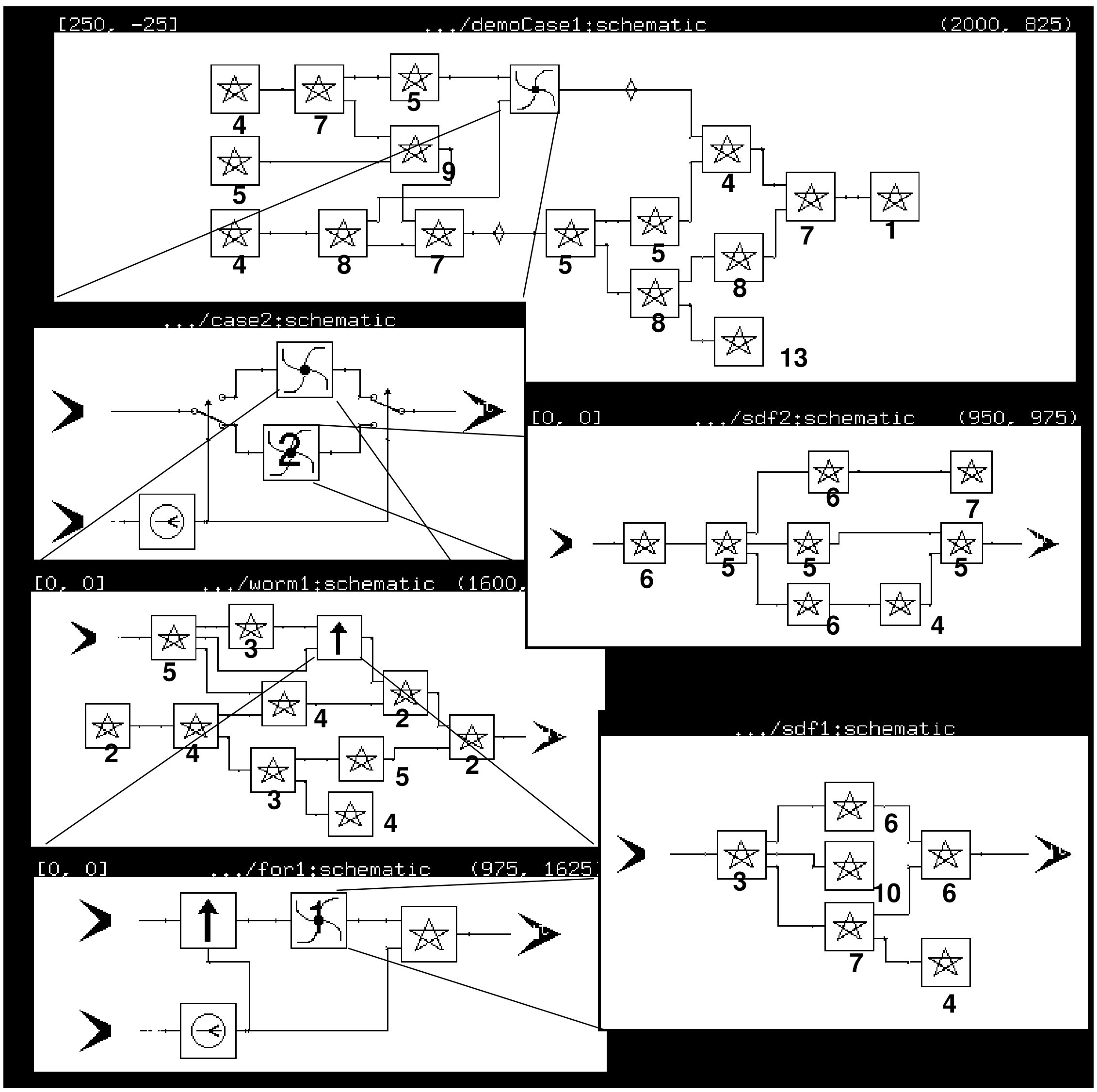
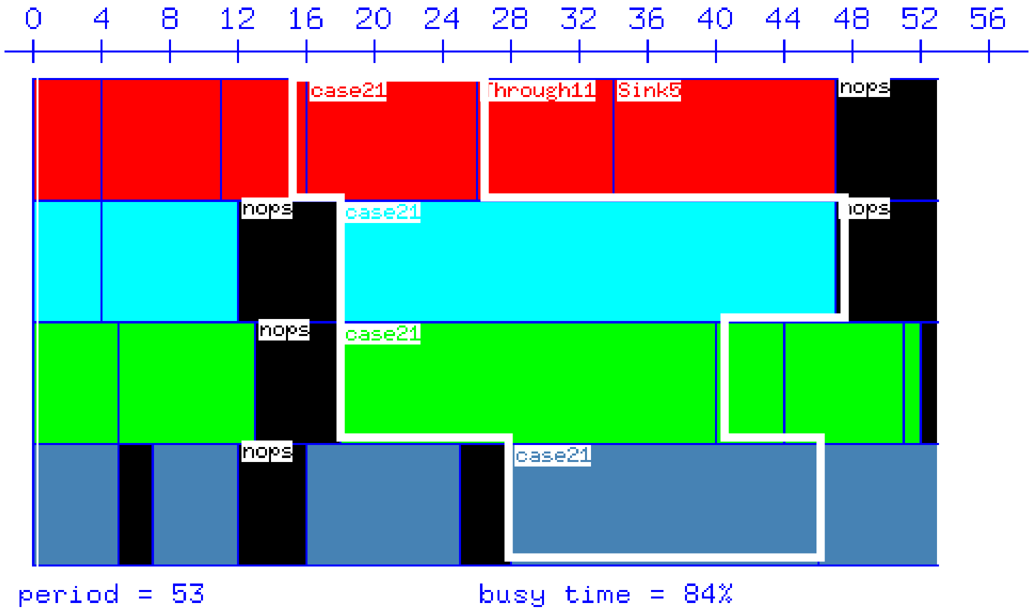
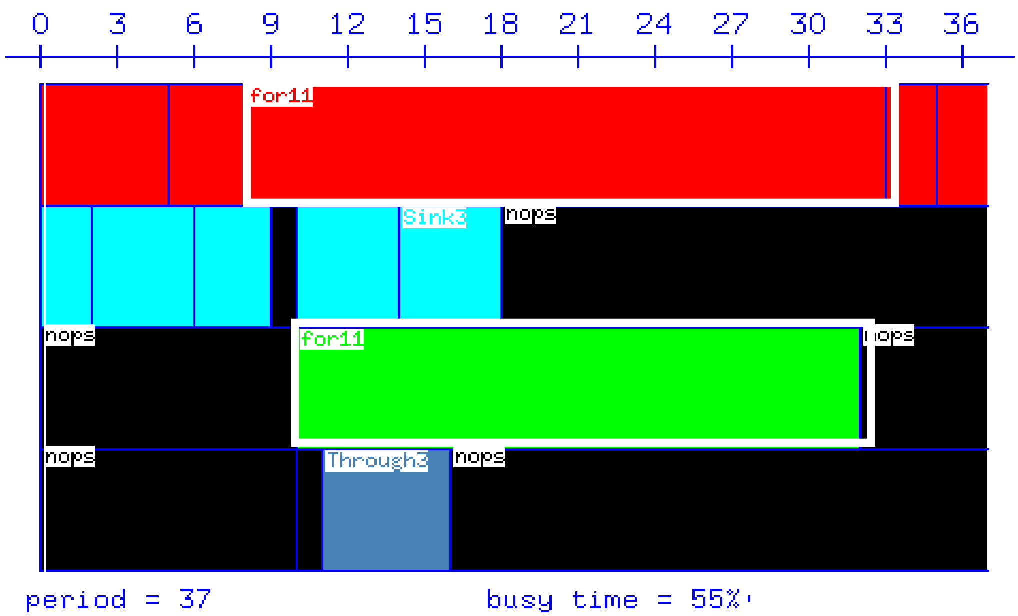


Figure 6.11 An example with a nested dynamic construct at the top level (large window). The system contains a Case construct which in turn contains a For construct in its "false" branch. The subsystems are also displayed in other windows



(a)



(b)

Figure 6.12 Gantt chart displays of the scheduling result over *4* processors from the proposed scheduling technique for the example in figure 6.11. (a) Schedule of the whole system. The profile of the Case construct is identified. We assume that the probability of taking the "true" branch is *0.7*. (b) Schedule of the "false" branch of the Case construct, which contains a For construct whose profile is identified. We assume that the number of iteration cycles is distributed geometrically with parameter *0.6*.

7

CONCLUSION

Brethren, I do not count myself to have apprehended; but one

thing I do, forgetting those things which are behind and

reaching forward to those things which are ahead,

I press toward the goal for the prize ......

--- Philippians 3:13,14

In this thesis we proposed a scheduling technique for a dataflow program with dynamic constructs onto multiple programmable processors. We first categorized four scheduling strategies, among which static-assignment and self-timed scheduling strategies look like the most promising compromises between hardware cost/performance and flexibility. The choice should depend on the amount of data-dependent behavior in the expected applications. Both strategies require compile-time decisions; they require that tasks be assigned to processors at compile time, and in addition, self-timed scheduling requires that the order of execution of the tasks be specified. If there is no data-dependency in the application, then these decisions can be made optimally (or nearly so, to avoid complexity problems).

When there is data-dependency, however, optimal or near optimal compile-time strategies become intractable. Most previously proposed solutions include random choices, clustering (to minimize communication overhead), and load balancing. These solutions either ignore precedence relationships in the dataflow graph, or use heuristics based on oversimplified stochastic models. This is justifiable if there is so much data-dependency that the precedence relationships are constantly changing. However, there is a large class of applications, including scientific computations and digital signal processing, where this is not true.

Nearly all applications of parallel computers involve some data-dependent behavior. Consequently, there is a clear need for compile-time strategies that can use precedence information in these cases. Quasi-static scheduling strategies have been previously proposed that can handle conditionals and some forms of iteration [Lee88][Loe88]. The main contribution of this thesis is to extend these techniques to handle dynamic constructs in a *systematic* way. We define the compile-time profile of a dynamic construct as an assumed local schedule of the dynamic construct. Based on the profiles of dynamic constructs, we perform a fully-static scheduling. The resulting static schedules give the information needed by a compiler in self-timed and static-assignment situations. Even though the proposed technique was implemented with list scheduling methods, it may be used in other scheduling methods. The proposed method should work well when the amount of data dependency is small, but we admittedly cannot quantify at what level the technique breaks down.

We require that the statistical distribution of the dynamic behavior, for example the distribution of the number of iteration cycles for a data-dependent iteration, must be known or estimated at compile time for each dynamic construct in the program. Using these probabilities, we find an “assumed” profile of dynamic constructs that the scheduler can use to construct a static schedule. This profile is selected to minimize the expected total cost. The total cost of a dynamic construct is the sum of the execution length of a construct and the idle time on all processors at runtime due to the difference between the compile-time profile and the actual runtime profile. This total cost is computed by assuming that the processors are globally synchronized (quasi-static scheduling strategy). We proved that the profile obtained from the previously proposed methods by E. Lee and Loeffler et. al. are optimal in some special cases only.

The dynamic constructs handled in this thesis are:

1. Conditionals or Case construct. We generalize an if-then-else construct to a case construct to allow an N-way branching capability.
2. Data-dependent iterations, such as For and DoWhile. We included the ability to overlap successive cycles of an iteration.
3. Recursions or Recur construct. We invented a dataflow representation, recursive representation using a Self Star, of a recursion construct. We show how to manage the degree of parallelism of the recursion constructs optimally through the proposed technique.

The proposed technique can be applied to any other dynamic constructs similarly. If there is a nested dynamic construct, the technique decides automatically whether to parallelize the inner construct only or the outer construct only or both.

It is shown that if the execution is self-timed, then the performance can only improve over the quasi-static case, and that the information generated by the quasi-static scheduler can be used at very low cost. For static-assignment scheduling, tractable runtime scheduling algorithms may actually lead to *worse* schedules than the quasi-static case, although most of the time the schedules will be better.

We implemented the technique in Ptolemy as a part of the rapid-prototyping environment. We illustrated the proposed technique using one example from graphics and some synthetic examples. These results are only a preliminary indication of the potential practical application, but they are very promising. For the synthetic examples, we found that the resulting quasi-static schedule could be at least 10% faster than other scheduling decisions currently existent, while it is as little as 15% slower than an ideal (and highly unrealistic) fully-dynamic schedule. For the graphics example, we found that the resulting quasi-static schedule could be as little as 3% slower than an ideal (and highly unrealistic) fully-dynamic schedule. This performance depends on a reasonable (but not exact) stochastic model of the dynamic construct, assumed by the compiler. For the particular program we selected, the performance does not degrade rapidly as the stochastic model gets further from actual program behavior, suggesting that a compiler can use fairly simple techniques to estimate the model.

7.1. FUTURE RESEARCH

There are still a number of issues that require further research. Some of these include:

1. A scheduling technique that allows some actors to require more than one processor. There is very little published research on this topic; one of the earliest is Blazewics et. al.'s [Bla86]. But, their heuristic based on a linear programming formulation ignores the precedence relations among actors, and thus is not applicable in this context. Currently, we use a modified list scheduling technique. Future research is needed to assess this approach and to search for better techniques.
2. Handling of coupled non-deterministic actors. In this proposal, we assume that non-deterministic actors are decoupled so that their dynamic behaviors are independent. This assumption will be reasonable for most signal processing applications. To apply the proposed scheme to more general problems, coupled non-deterministic actors should be handled properly. One possible approach will be to make a new level in the hierarchical dataflow graph that gathers all coupled non-

deterministic actors.

3. The amount of non-determinism allowable in this scheme. Another assumption in this technique is that a program has at most a small amount of non-determinism in its behavior. If the amount of non-determinism is too large, the proposed scheme may not be efficient compared with random assignment or other scheduling schemes.
4. Efficient code generation for run-time decisions. For each dynamic construct, code that performs the run-time decision must be inserted before and after the execution of the construct. Resource management will be a crucial issue especially for a recursion construct. Since there has been no previous research, this will be a challenging task.
5. Scheduling for heterogeneous multiprocessor systems. Since we implement Sih's dynamic level scheduling algorithm, our system can be extended to schedule a heterogeneous multiprocessor system [Sih91].

REFERENCES

[Ack82]

W. B. Ackerman, “Data Flow Languages,” *Computer*, **Vol. 15, No. 2**, pp. 15-25, February, 1982.

[Ada74]

T. L. Adam, K. M. Chandy, and J. R. Dickson, ”A Comparison of List Schedules for Parallel Processing Systems,” *Comm. ACM*, **17(12)**, pp. 685-690, Dec., 1974.

[Amd67]

G. M. Amdahl, “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities,” *AFIPS Conference Proceedings*, **30**, pp. 483-485, 1967.

[Arv82]

Arvind and K. P. Gostelow, ”The U-Interpreter,” *Computer*, **15(2)**, February 1982.

[Arv86]

Arvind and D. E. Culler, “Dataflow Architectures,” *Annual Review in Computer Science*, **Vol. 1**, pp. 225-253, 1986.

[Arv87]

Arvind, R. S. Nikhil and K. K. Pingali, “Id Nouveau Reference Manual: Part II: Operational Semantics,” MIT Computation Structures Group, April, 1987.

[Arv88a]

Arvind and R. S. Nikhil, “Executing a Program on the MIT Tagged-Token Dataflow Architecture,” Computations Structures Group Memo 271, MIT, July, 1988.

[Arv88b]

Arvind, D. E. Culler, and K. Ekanadham “The Price of Asynchronous Parallelism: An Analysis of Dataflow Architecture", Computation Structures Group Memo 278, MIT, June, 1988.

[Bab84]

R. G. Babb, "Parallel Processing with Large Grain Dataflow Techniques," *Computer*, **Vol. 17**, July, 1984.

[Bac78]

J. Backus, "Can Programming Be Liberated from the von Neumann Style? AFunctional Style and Its Algebra of Programs," *Communications of the ACM*, **Vol. 21, No. 8**, pp. 613-641, August, 1982.

[Bha91]

S. Bhattacharyya, "Scheduling Synchronous Dataflow Graphs for Efficient Iteration," Master's Thesis, EECS Dept. Univ. of Calif. Berkeley, May, 1991.

[Bia87]

R. P. Bianchini, JR. and J. P. Shen, "Interprocessor Traffic Scheduling Algorithm for Multiple Processor Networks," *IEEE Trans. Computers*, **Vol. C-36, No. 4**, pp.396-409, April, 1987.

[Bie89]

J. Bier and E. A. Lee, "Frigg: A Simulation Environment for Multiprocessor DSP System Development," *Proc. of Int. Conf. on Computer Design*, Boston, MA, October, 1989.

[Bla86]

J. Blazewics, M. Drabowski, and J. Weglarz, "Scheduling Multiprocessor Tasks to Minimize Schedule Length," *IEEE Trans. Computers*, **Vol. C-35, No. 5**, pp. 389-393, May, 1986.

[Bok88]

S. Bokhari, "Assignment Problems in Parallel and Distributed Computing," *Par-*

allel Processing and Fifth Generation Computing, Kluwer Academic Publishers, 1988.

[Buc91a]

Joseph Buck, Soonhoi Ha, Edward A. Lee, and David G. Messerschmitt, “Multirate Signal Processing in Ptolemy”, *ICASSP*-91, Toronto, 1991.

[Buc91b]

J. Buck, S. Ha, E. A. Lee, and D.G. Messerschmitt, “Ptolemy: A Platform for Heterogeneous Simulation and Prototyping,” *Proc. 1991 European Simulation Conference*, Copenhagen, Denmark, June 17-19, 1991.

[Buh84]

L. N. Bhuyan and D. P. Agrawal, “Generalized Hypercube and Hyperbus Structure for a Computer Network,” *IEEE Trans. Computers*, **Vol. C-21, No. 4**, pp. 323-333, April 1984.

[Bur81]

F. W. Burton and M. R. Sleep, “Executing Functional Programs on A Virtual Tree of Processors,” *Proc. ACM Conf. Functional Programming Lang.Comput. Arch.*, pp. 187-194, 1981.

[Cam85]

M. L. Campbell, “Static Allocation for a Data Flow Multiprocessor,” *Proceedings of the 1985 Intern. Conf. on Parallel Processing*, pp. 511-516, 1985.

[Cap84]

P. R. Cappello and K. Steiglitz, ”Some Complexity Issues in Digital Signal Processing,” *IEEE Trans. ASSP*, **ASSP-32 (5)**, October 1984.

[Cha84]

M. Chase, ”A pipelined Data Flow Architecture for Signal Processing: the NEC uPD7281,” VLSI Signal Processing, IEEE Press, New York (1984)

[Cha92]

L.-F. Chao and E. H.-M. Sha, “Unfolding and Retiming Data-Flow DSP Programs for RISC Multiprocessor Scheduling,” *ICASSP*, San Francisco, 1992.

[Chu80]

W. W. Chu, L. J. Holloway, L. M.-T. Lan, and K. Efe, “Task Allocation in Distributed Data Processing,” *IEEE Computer*, pp. 57-69, November, 1980.

[Chu87]

W. W. Chu and L. M.-T. Lan, ”Task Allocation and Precedence Relations for Distributed Real-Time Systems,” *IEEE Trans. on Computers*, **C-36(6)**, pp. 667-679, June 1987.

[Cof76]

E. G. Coffman, Jr., Computer and Job Scheduling Theory, Wiley, New York (1976)

[Cor79]

M. Cornish, D. W. Hogan, and J. C. Jensen, “The Texas Instruments Distributed Data Processor,” *Proc. Louisiana Computer Exposition*, Lafayette, La., March 1979, pp. 189-193.

[Dav78]

A. L. Davis, ”The Architecture and System Method of DDM1:A Recursively Structured Data Driven Machine,” *Proc. Fifth Ann. Symp. Computer Architecture*, April, 1978, pp. 210-215.

[Dav81]

H. A. David, “Order Statistics,” Wiley Press, 1981.

[Den75]

J. B. Dennis and D. P. Misunas, “A Preliminary Architecture for a Basic Dataflow Processors,” *Proc. 2nd Ann. Symp. Computer Architecture*, New York, May, 1975.

[Den80]

J. B. Dennis, ”Data Flow Supercomputers,” *Computer*,**13(11)**, November 1980.

[Efe82]

K. Efe, “Heuristic Models of Task Assignment Scheduling in Distributed Systems,” *IEEE Computer,* pp. 50-56, June, 1982.

[Fen81]

T.-y. Feng, “A Survey of Interconnection Networks,” *Computer*, pp. 12-26, December, 1981.

[Fin81]

R. A. Finkel and M. H. Solomon, “The Lens Interconnection Strategy,” *IEEE Trans. Computers*, pp. 291-295, April 1981.

[Fis84]

J. A. Fisher, ”The VLIW Machine: A Multiprocessor for Compiling Scientific Code,” *Computer*, July, 1984, 17(7).

[Gao83]

G. R. Gao, “A pipelined Code Mapping Scheme for Static Dataflow Computers,” Ph.D. dissertation, Laboratory for Computer Science, MIT, Cambridge, MA (1983).

[Gao88]

G. R. Gao, R. Tio, and H. H. J. Hum, ”Design of an Efficient Dataflow Architecture without Data Flow,” *Proc. Int. Conf. on Fifth Generation Computer Systems*, 1988.

[Gau85]

J. L. Gaudiot, R. W. Vedder, G. K. Tucker, D. Finn, and M. L. Campbell, “A Distributed VLSI Architecture for efficient signal and data processing,” *IEEE Trans.*

Computers, **Vol. C-34**, pp. 1072-1087, December 1985.

[Gau87]

J. L. Gaudiot, “Data-Driven Multicomputers in Digital Signal Processing,” *IEEE Proceedings,* **Vol. 75, No. 9**, pp. 1220-1234, September, 1987.

[Gel91]

P. R. Gelabert and T. P. Barnwell, III, “Optimal Automatic Periodic Multiprocessor Scheduler for Fully Specified Flow Graphs,” submitted to *IEEE Trans. ASSP*, 1991.

[Gir87]

E. F. Girczyc, “Loop Winding - A Data Flow Approach to Functional pipelining,” *ISCAS*, pp. 382-385, 1987.

[Gon77]

M. J. Gonzalez, ”Deterministic Processor Scheduling,” *Computing Surveys*, **9(3)**, September, 1977.

[Gos82]

K. P. Gostelow, “The U-Interpreter,” *Computer*, pp. 42-49, 1982.

[Gra87]

M. Granski, I. Korn, and G.M. Silberman, “The Effect of Operation Scheduling on the Performance of a Data Flow Computer,” *IEEE Trans. on Computers*, **C-36(9)**, September, 1987.

[Gur85]

J. R. Gurd, C. C. Kirkham, and I. Watson, “The Manchester Dataflow Prototype Computer,” *Communications of the ACM*, **28**, January, pp. 34-52, 1985.

[Ha90]

S. Ha and E. A. Lee, ”Compile-time Scheduling and Assignment of Dataflow Program Graphs with Data-Dependent Iteration,” to appeared in *IEEE trans. on*

Computers.

[Ha91]

Soonhoi Ha and Edward A. Lee, “Quasi-Static Scheduling for Multiprocessor DSP,” *ISCAS-91*, Singapore, 1991.

[Han72]

M. Hanan and J. M. Kurtzberg, “A Review of the Placement and Quadratic Assignment Problems,” *SIAM Review*, **Vol. 14, No. 2**, pp. 324-342, April, 1972.

[Hil89]

P. N. Hilfinger, “Silage Reference Manual, DRAFT Release 2.0,” Computer Science Division, EECS Dept., UC Berkeley July 8, 1989.

[Hoa78]

C. A. R. Hoare, ”Communicating Sequential Processes,” *Communications of the ACM,* August 1978, **21(8)**

[Hoa90]

P. Hoang and J. Rabaey, “Program Partitioning for a Reconfigurable Multiprocessor System,” *IEEE Workshop on VLSI Signal Processing IV*, November, 1990.

[Hu61]

T. C. Hu, “Parallel Sequencing and Assembly Line Problems,” *Operations Research*, **9(6)**, pp. 841-848, 1961.

[Hum91]

H. H. J. Hum and G. R. Gao, “Efficient Support of Concurrent Threads in a Hybrid Dataflow / von Neumann Architecture,” *The third IEEE symposium on Parallel and Distributed Processing*, Dallas, December, 1991.

[Hwa84]

K. Hwang and F. A. Briggs, “Computer Architecture and Parallel Processing,” McGraw Hill, 1984.

[Ian88]

R. A. Iannucci, “A Dataflow / von Neumann Hybrid Architecture,” Ph.D. dissertation, MIT, 1988.

[Iqb86]

M. A. Iqbal, J. H. Saltz, and S. H. Bokhari, “A Comparative Analysis of Static and Dynamic Load Balancing Strategies,” *Int. Conf. on Parallel Processing*, pp. 1040-1045, 1986.

[Kel84]

R. M. Keller, F. C. H. Lin, and J. Tanaka, “Rediflow Multiprocessing,” *Proc. IEEE COMPCON*, pp. 410-417, February, 1984.

[Kim88]

S. J. Kim and J. C. Browne, ”A General Approach to Mapping of Parallel Computations upon Multiprocessor Architectures,” *Proc. Int. Conf. on Distributed Computing Systems*, 1988.

[Kon90]

K. Konstantinides, R. T. Kaneshiro, and J. R. Tani, “Task Allocation and Scheduling Models for Multiprocessor Digital Signal Processing,” *IEEE Trans. Acoustics, Speech, and Signal Processing,* **Vol. 38.No. 12,** pp. 2151-2161, December, 1990.

[Kun87]

J. Kunkel, ”Parallelism in COSSAP,” Internal Memorandum, Aachen University of Technology, Fed. Rep. of Germany,1987.

[Kun88]

S. Y. Kung, VLSI Array Processors, Prentice-Hall, Englewood Cliffs, NJ (1988).

[Lee87]

S. Lee and J. K. Aggarwal, “A Mapping Strategy for Parallel Processing,” *IEEE*

Trans. Computers, **Vol. C-36, No. 4**, pp. 433-442, April, 1987.

[Lee87a]

E. A. Lee and D. G. Messerschmitt, “Static Scheduling of Synchronous Data-Flow Graph for Digital Signal Processing,” *IEEE Trans. on Computers*, January, 1987.

[Lee87b]

E. A. Lee and D. G. Messerschmitt, “Synchronous Data Flow,” *IEEE Proceedings*, September, 1987.

[Lee88]

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

[Lee89a]

E. A. Lee, W.-H. Ho, E. Goei, J. Bier, and S. Bhattacharyya, ”Gabriel: A Design Environment for DSP,” *IEEE Trans. on ASSP*, November, 1989.

[Lee89b]

E. A. Lee and S. Ha, “Scheduling Strategies for Multiprocessor Real-time DSP,” *GLOBECOM*, November, 1989.

[Lee90]

E. A. Lee and J. C. Bier, “Architectures for Statically Scheduled Dataflow,” *Journal on Parallel and Distributed Systems*, December, 1990.

[Lee91a]

E. A. Lee and J. C. Bier, “Architectures for Statically Scheduled Dataflow,” reprinted in *Parallel Algorithms and Architectures for DSP Applications*, Kluwer Academic Pub., 1991.

[Lee91b]

E. A. Lee, “Consistency in Dataflow Graphs,” *IEEE Trans. on Parallel and Dis-*

tributed Systems, **Vol. 2**, **No. 2**, April, 1991.

[Lei83]

C. E. Leiserson, “Optimizing Synchronous Circuitry by Retiming,” *Third Caltech Conference on VLSI*, Pasadena, CA, March, 1983.

[Loe88]

C. Loeffler, A. Ligtenberg, H. Bheda, and G. Moschytz, “Hierarchical Scheduling system for Parallel Architectures,” *Proceedings of Euco*, Grenoble, September, 1988.

[Lu86]

H. Lu and M. J. Carey, “Load-Balanced Task Allocation in Locally Distributed Computer Systems,” *Int. Conf. on Parallel Processing*, pp. 1037-1039, 1986.

[Ma82]

P. R. Ma, E. Y. S. Lee and M. Tsuchiya, “A Task Allocation Model for Distributed Computing Systems,” *IEEE Trans. on Computers*, **Vol. C-31,No. 1**, pp. 41-47, January, 1982.

[Mar69]

D. F. Martin and G. Estrin, ”Path Length Computations on Graph Models of Computations,” *IEEE Trans. on Computers*, **C-18**, pp. 530-536, June 1969.

[Mcg82]

J. R. McGraw, “The VAL Language: Description and Analysis,” *ACM Trans. on Programming Languages and Systems*, **4(1)**, pp. 44-82, January, 1982.

[McG83]

J. McGraw, “Sisal: Streams and Iteration in a Single Assignment Language,” *Language Reference Manual*, Lawrence Livermore National Laboratory, Livermore, CA94550, 1983.

[Mes84]

D. G. Messerschmitt, “A Tool for Structured Functional Simulation,” *IEEE Journal on Selected Areas in Communications*, **SAC-2(1)**, January, 1984.

[Muh87]

H. Muhlenbeim, M. Gorges-Schleuter, and O. Kramer, “New Solutions to the Mapping Problem of Parallel Systems: The Evolution Approach,” *Parallel Computing*, **4,** pp. 269-279, 1987.

[Nik89]

R. S. Nikhil and Arvind, “Programming in Id: a Parallel Programming Language,” MIT Draft, 1989.

[Ona89]

J. S. Onanian, “A Signal Processing Language for Coarse Grain Dataflow Multiprocessor,” Technical Report MIT/LCS/TR-449, June, 1989.

[Pap88]

G. M. Papadopoulos, ”Implementation of a General Purpose Dataflow Multiprocessor,” Dept. of Electrical Engineering and Computer Science, MIT, Ph.D. Thesis, August, 1988.

[Par89a]

K. K. Parhi and D. G. Messerschmitt, “Pipeline Interleaving and Parallelism in Recursive Digital Filters - Part I: Pipelining Using Scattered Look-Ahead and Decomposition,” *IEEE Trans. on ASSP*, **Vol. 37**, **No. 7**, pp. 1099-1117, July, 1989.

[Par89b]

K. K. Parhi and D. G. Messerschmitt, “Pipeline Interleaving and Parallelism in Recursive Digital Filters - Part II: Pipelined Incremental Block Filtering,” *IEEE Trans. on ASSP*, **Vol. 37**, **No. 7**, pp. 1099-1117, July, 1989.

[Pla76]

A. Plas, et. al., ”LAU System Architecture: A Parallel Data-driven Processor Based on Single Assignment,” *Proc. 1976 Int. Conf. Parallel Processing*, pp. 293-302.

[Pot91]

M. Potkonjak : “Algorithms for High Level Synthesis: Resource Utilization Based Approach,” Ph.D. Thesis, University of California at Berkeley, 1991.

[Pre81]

F. P. Preparata and J. Vuillemin, “The Cube-Connected Cycles: A Versatile Network for Parallel Computation,” *Comm. ACM,* **Vol. 24, No. 5**, pp. 300-309, May 1981.

[Rao85]

S. K. Rao, ”Regular Iterative Algorithms and their Implementations on Processor Arrays,” *Information Systems Laboratory*, Stanford University, October, 1985, Ph.D. Dissertation.

[Ree87]

D. A. Reed and D. C. Grunwald, “The Performance of Multicomputer Interconnection Networks,” Computer, pp. 63-73, June 1987.

[Ren81]

M. Renfors and Y. Nuevo, “The Maximum Sampling Rate of Digital Filters Under Hardware Speed Constraints,” *IEEE Trans. Circuits and Systems*, **Vol. CAS-28**, **No. 3**, pp. 196-202, 1981.

[Sar87]

V. Sarkar, “Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors,” Ph.D. dissertation, Stanford University, 1987.

[Sch85]

D. A. Schwartz, ”Synchronous Multiprocessor Realizations of Shift-Invariant

Flow Graphs," Georgia Institute of Technology Technical Report DSPL-85-2, Ph.D. Thesis, July 1985.

[Sch86]

D. A. Schwartz and T. P. Barnwell, III, "Cyclo-Static Solutions: Optimal Multiprocessor Realizations of Recursive Algorithms," *VLSI Signal Processing*, IEEE Press (1986).

[Sih90a]

G. C. Sih and E. A. Lee, "Scheduling to Account for Interprocessor Communication Within Interconnection-Constrained Processor Networks," *Proceedings of the International Conference of Parallel Processing*, pp. 9-16, 1990.

[Sih90b]

G. C. Sih and E. A. Lee, "Dynamic-Level Scheduling for Heterogeneous Processor Networks," *2nd IEEE Symposium on Parallel and Distributed Processing*, pp. 42-49, 1990.

[Sih91]

G. C. Sih, "Multiprocessor Scheduling to Account for Interprocessor Communication," Ph.D. dissertation, U. C. Berkeley, 1991.

[Smi85]

B. Smith, "The Architecture of HEP," In J. S. Kowalik, editor, *Parallel MIMD Computation: HEP Supercomputer and its Application*, pp. 41-55, MIT Press, 1985.

[Sri86]

V. P. Srini, "An Architectural Comparison of Dataflow Systems," *Computer*, pp. 68-88, March, 1986.

[Sto71]

H. S. Stone, "Parallel Processing with the Perfect Shuffle," *IEEE Trans. Comput-*

ers, **Vol. C-20, No. 2**, pp. 153-161, February 1971.

[Suh90]

P. A. Suhler, J. Biswas, K. M. Kohner, nd J. C. Browne, "TDFL: A Task-Level Dataflow Language," *Journal of Parallel and Distributed Computing*, **9**, pp. 103-115, 1990.

[Tha90]

M. Thaler and G. S. Moschytz, "A Data Flow Technique for the Efficient Design of a Class of Parallel Non-Data Flow Signal Processors," *IEEE Trans. on Acoustics, Speech, and Signal Processing*, pp. 2162-2173, December, 1990.

[Ull75]

J. D. Ullman, "NP-Complete Scheduling Problems," *Journal of Computer and System Sciences* **10**, pp. 384-393, 1975.

[Veg84]

S. R. Vegdahl, "A Survey of Proposed Architectures for the Execution of Functional Languages," *IEEE Trans. Computers*, **Vol c-33,No. 12**, pp. 1050 - 1071, December, 1984.

[Wat82]

I. Watson and J. Gurd, "A Practical Data Flow Computer," *Computer* **15(2)**, February 1982.

[Wit81]

L. D. Wittie, "Communication Structures for Large Networks of Microcomputers," *IEEE Trans. on Computers*, **Vol. C-30, No. 4**, pp. 264-273, April 1981.

[Zis87]

M. A. Zissman and G. C. O'Leary, "A Block Diagram Compiler for a Digital Signal Processing MIMD Computer," *IEEE Int. Conf. on ASSP*, pp. 1867-1870, 1987.

APPENDIX I

ANALYSIS OF A TWO PHASE SCHEDULING STRATEGY

In a two-phase scheduling strategy, we first partition the program into a multiprocessor target machine ignoring the communication network topology (partitioning phase), and next assign the partitioned actors to the physical processors (assignment phase). The objective of the assignment phase is to minimize the total communication delays, or message traffic. In this appendix, we analytically investigate the expected performance improvement of an optimal processor assignment over random assignment.

We make some simplifying assumptions about the communication network topology in order to get analytical results. They are:

1. A dedicated link exists between each pair of processors.
2. Each link is categorized as either low-cost link or a high-cost link.

For an *N* processor system, the total number of links is $M = \frac{N(N-1)}{2}$. We also ignore the network congestion.

In real network configurations except the completely connected network, a processor has direct links to less than *N* other processors. For example, the total number of links is $\frac{N}{2}\log N$ in a binary hypercube network, and *N-1* in a ring network. Two processors are called neighbors if they are connected by a direct link. A unit of message between a processor pair not connected by a direct link has to be routed through a path of direct links connecting them. The number of direct links on the path, called *hops*, defines the cost of the message. The cost of communication between two processors is defined by the minimum cost of a message between them. A unit of message between neighbors can be transmitted by one *hop*, having cost one. Hence, the cost of communication between them is one. The cost of communication in a network ranges between *1* and the *diameter* of the network, which is $\lceil \log N \rceil$ in case of a binary hypercube network, and $\left\lfloor \frac{N}{2} \right\rfloor$ in case of a ring network.

Our oversimplified network may roughly model realistic situations by the following transformation.

Transformation A

1. Define a threshold value, c_{th}.
2. Connect each processor pair with a high-cost link if the communication cost between them is greater than the threshold value. Let the high-cost be c_h and let the number of high-cost links be *L*.
3. Connect remaining processor pairs with low-cost links. All direct links are transformed to low-cost links. Let the low-cost be c_l. The number of low-cost links is *M-L*.

If we set $c_{th} = 1.5$, the direct links alone belong to low-cost links with $c_l = 1$. Then the number of low-cost links becomes $\frac{N}{2}\log N$ in a binary hypercube network. By controlling the parameters, c_{th}, c_l, and c_h we may compensate for the effect of our simplifying assumptions and apply the same approach to various kinds of network topology.

If a message is transmitted through a link, the corresponding message traffic is defined as the product of the volume of the communication data and the cost of the link. The total message traffic is the sum of all message traffic during an execution of a given program. The optimal binding of virtual processors to physical processors minimizes the total message traffic.

Recall that the objective is to minimize the total message traffic. The obvious solution is to map *M-L* processor pairs with the *M-L* largest message traffic loads onto low-cost links, and the other *L* pairs onto high-cost links. Because this mapping may not be realizable, however, we admit that our analysis gives only a rough estimation of the expected performance improvement over the random assignment.

We model the distribution of the volume of the communication data between a pair of processors, which is called the *routing distribution*, as uniform on the normalized interval [*0,1*] to make the analysis tractable. If the distribution is not uniform, our experiment also shows that the improvement is comparable to or larger than the analysis result, which implies that the partitioning phase should give a skewed routing distribution to make the second phase more effective.

Suppose we have a set of *M* random samples, $\{X_i\}$, valued on [*0,1*]. We assume that they are independent each other and identically distributed. We rearrange them into an ordered set $\{X_{i|M}\}$, in which $X_{i|M}$ is a random variable representing the *i*-th smallest sample among *M* samples in $\{X_i\}$. Clearly, the difference between any outcome of the set $\{X_i\}$ and any outcome of the set $\{X_{i|M}\}$ is the order. Each set represents the normalized volume of communication data between processor pairs. Then the expected total message traffic *E(T)* becomes

$$E(T) = E\left(c_h \sum_{i=1}^{L} X_{i|M} + c_l \sum_{i=L+1}^{M} X_{i|M}\right), \tag{1}$$

where *E()* means the expected value of the expression in the parenthesis. If the probability of the event $\{X_{i|M} = x\}$ is *f(x)*, *E(T)* can be reformulated as follows using conditional expectation.

$$E(T) = \int_0^1 E\left(c_h \sum_{i=1}^{L} X_{i|M} + c_l \sum_{i=L+1}^{M} X_{i|M} \middle| X_{L|M} = x\right) f(x)dx \tag{2}$$

The first part of the expectation is

$$E\left(c_h \sum_{i=1}^{L} X_{i|M} \middle| X_{L|M} = x\right) = E\left(c_h \sum_{i=1}^{L-1} Y_i\right) + c_h x \quad , \tag{3}$$

where $\{Y_i;\ i= 1 \ldots L-1\}$ is a set of uniform random variables on the interval [*0,x*]. This can be written as

$$c_h(L-1)\int_0^x \frac{1}{x} r dr + c_h x = c_h(L+1)\frac{x}{2} \quad . \tag{4}$$

The second part is

$$E\left(c_l \sum_{i=L+1}^{M} X_{i|M} \middle| X_{L|M} = x\right) = E\left(c_l \sum_{i=1}^{M-L} Y_i\right) , \tag{5}$$

where $\{Y_i;\ i= L+1 \ldots M\}$ is a set of uniform random variables on the interval [*x,1*]. Therefore, the second part becomes

$$c_l(M-L)\int_x^1 \frac{1}{1-x} r dr = c_l(M-L)\frac{x+1}{2} \quad . \tag{6}$$

From order statistics [Dav81],

$$f(x) = \frac{x^{L-1}(1-x)^{M-L}}{B(L, M-L+1)} \qquad where \qquad B(a, b) = \frac{(a-1)!(b-1)!}{(a+b-1)!} \quad . \tag{7}$$

From equations (4), (6), and (7), *E(T)* can be reduced to the following form.

$$E(T) = \int_0^1 \left[c_h(L+1)\frac{x}{2} + c_l(M-L)\frac{x+1}{2}\right] \frac{x^{L-1}(1-x)^{M-L}}{B(L, M-L+1)} dx \tag{8}$$

After a few manipulations, *E(T)* becomes

$$E(T) = c_l \frac{M}{2} + (c_h - c_l)\frac{L(L+1)}{2(M+1)} \quad . \tag{9}$$

For a given number *M* of processors, this formula for the expected amount of total message traffic has three unknowns depending on the network configuration. When the network is fully connected, both c_l and c_h are equal to 1, and *E(T)* becomes $\frac{M}{2}$ as expected. As another example, suppose we have a *4*-dimensional binary hypercube network, where *N* is 16 and *M* is 120. Suppose we set c_{th} equal to *2.5*. Then, *L* corresponds to the number of processor pairs embedded on the hypercube with hamming distance greater than *2.5*, which is *40* (sum of *32*(distance = *3*) and *8*(distance = *4*)). The corresponding high-cost c_h may be approximated as the average cost, *3.2*. Similarly the lower cost c_l may be approximated as *1.6*, resulting in *E(T)* equal to *106.8*.

To assess the above result, suppose that we assign processors randomly. Then, the expected amount of total message traffic $E'(T)$ becomes

$$E'(T) = c_l \frac{M}{2} + (c_h - c_l)\frac{L}{2} \tag{10}$$

From equations (9) and (10), the difference between *E(T)* and *E'(T)*, $\Delta E(T)$, is

$$\Delta E(T) = \frac{(c_h - c_l)}{2}\frac{L(M-L)}{M+1} \quad . \tag{11}$$

The performance improvement by adopting the ideal binding over the random binding can be seen by examining the ratio of $\Delta E(T)$ and *E'(T)*. Define the cost ratio r_c as $\frac{c_h}{c_l}$, and the high-cost link ratio r_l as $\frac{L}{M}$. Then, the ratio of $\Delta E(T)$ and *E'(T)* can be represented in terms of r_c and r_l as follows (we approximate the term *M+1* in the denominator of equation (11) as *M)*:

$$\frac{\Delta E(T)}{E'(T)} = \frac{r_l(1-r_l)(r_c-1)}{1+r_l(r_c-1)} \quad . \tag{12}$$

We show the expected performance improvement in figure 1, varying $\frac{L}{M}$ and $\frac{c_h}{c_l}$.

For a given network topology, we can calculate the high-cost link ratio and the cost ratio by mapping it to the model network topology by **Transformation A**, and obtain the expected improvement from the graph. We choose c_{th} so that $\frac{L}{M}$ is close to a half.

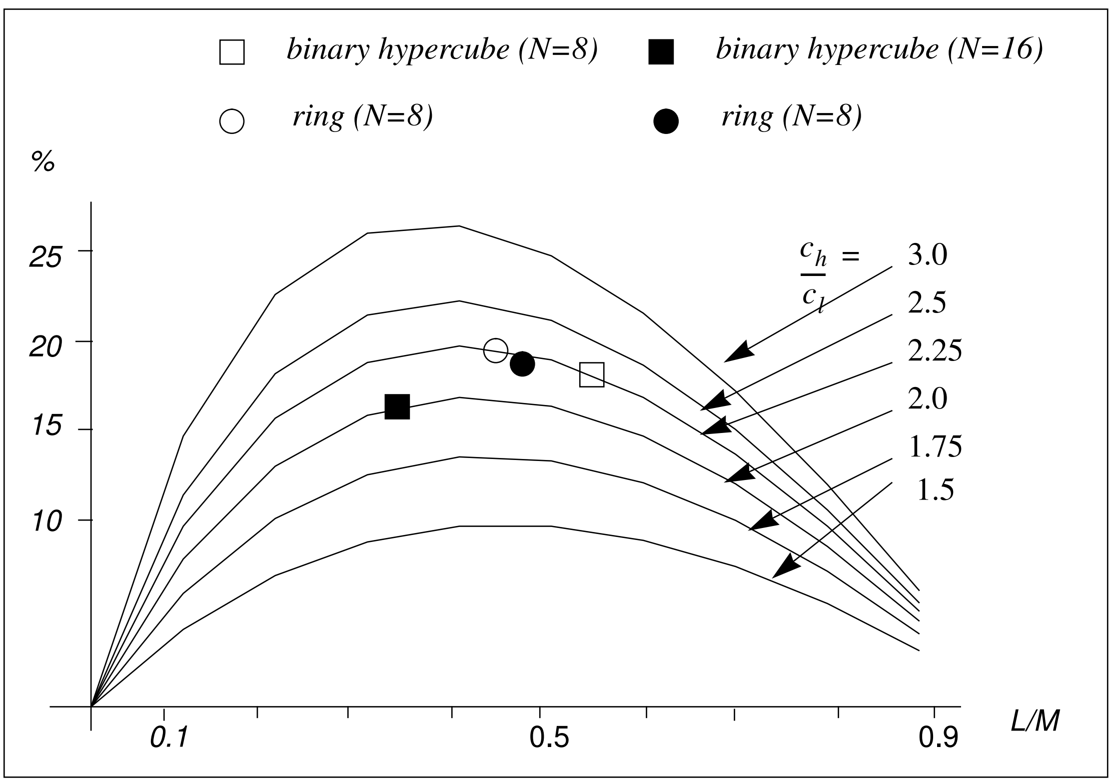


Figure 1 Analytical message traffic reduction with varying $\frac{c_h}{c_l}$ and $\frac{L}{M}$ from the optimal assignment under the assumption of uniform routing distribution.

The values of the high-cost and low-cost are made equal to the average costs of the high-cost links and low-cost links respectively in the original network. We display two square marks in the graph indicating the binary hypercube network with *8* and *16* processors. Two circle marks are for the ring network with *8* and *16* processors. The threshold values for the hypercube network are $c_{th} = 1.5$ for *8* processors and $c_{th} = 2.5$ for *16* processors respectively. For the ring network they are $c_{th} = 2.5$ for *8* processors and $c_{th} = 4.5$ for *16* processors. From the graph, we can expect about *17* or *18* percent improvement using the optimal assignment over a random assignment in a binary hypercube network of *8* or *16* processors, and *20* percent improvement with ring network.

We need to point out the inaccuracy in deciding the costs, c_h and c_l. In the case of random assignment, the average costs of the links can be regarded as the exact costs. But in the case of the optimal assignment, adopting the average costs underestimates the performance. Since we try to locate the links of higher cost as close as possible for the optimal assignment, the costs c_h and c_l might have to be smaller than the average costs. This makes the expected optimal message traffic smaller (equation (9)), and accordingly the performance improvement is larger. Since this will rarely increase the improvement more than a few percent, we will ignore this effect at the expense of some accuracy.

Although the obtained expected performance improvement is an upper bound with uniform routing distribution, we expect to achieve as much improvement with other skewed distributions, in which a processor tends to communicate with a certain processors more heavily than with other processors. Our simulation results with some skewed distribution models shows that we achieve *15-20 %* performance improvement (reduction of total message traffic density) compared with a random assignment, which by and large coincides with the analysis. On the other hand, unified partitioning and assignment leads to performance improvements as large as *50%* in terms of the total message traffic reduction as shown by G. Sih [Sih91]. He compared his dynamic level scheduling algorithm with a conventional list scheduling algorithm. It implies that the unified strategy is more effective in reducing the IPC overhead.