Activation Frame Memory Management for the Monsoon Processor

by

Derek Chiou

B.S.E.E. Massachusetts Institute of Technology
(1989)

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of
the Requirements for the Degree of
Master of Science in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

September 1992

© Massachusetts Institute of Technology 1992

Signature of Author Signature redacted
Department of Electrical Engineering and Computer Science
August 31, 1992

Certified by Signature redacted
Prof. Gregory Michael Papadopoulos
Assistant Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by Signature redacted
Campbell L. Searle
Chairman, Departmental Committee on Graduate Students

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY
OCT 30 1992
ARCHIVES
LIBRARIES

Activation Frame Memory Management for the Monsoon Processor

by

Derek Chiou

Submitted to the Department of Electrical Engineering and Computer Science
on August 31, 1992
in partial fulfillment of the requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

Multiprocessor architectures require sophisticated, yet fast memory management primitives to support procedure calls in implicitly parallel high level languages. These primitives serve two intertwined purposes — allocation of activation area, called a *frame*, for an invocation of a procedure, and, if a frame resides on a single processor, distribution of work across the nodes of the multiprocessor.

We will describe several implementations of an activation frame memory management system written for the implicitly parallel language Id and running on the experimental dataflow processor Monsoon. We will also present and analyze their run-time behavior on several benchmarks. For an eight-processor Monsoon, we found that a simple quick-fit allocator employing round-robin load distribution gave the best results.

Thesis Supervisor: Gregory Michael Papadopoulos

Title: Assistant Professor of Electrical Engineering and Computer Science

Acknowledgments

My advisor, Professor Gregory Papadopoulos, has led me through another thesis. His ability to figure out when to let me go my own merry way and when to reign me in has made the thesis process as smooth as possible.

Professor Arvind and Jonathan Young introduced me to the world of frame management. Jonathan designed the first interesting frame manager and helped me learn the Monsoon instruction set.

The Monsoon software team including Ken Traub, Jamey Hicks, Mike Bekerle, Boon Ang, Peter deWolf, Andy Shaw, Christine Flood, Alejandro Caro and R. Paul Johnson, designed, implemented and tested the software necessary to run Monsoon. Without their work, no Monsoon work would have been done. Jamey was especially helpful, explaining fine points when I started on this topic and acting as a great sounding board for ideas later on.

Mariquita Gilfillan typed all but the final changes to this thesis. She also helped produced many of the run-time statistics presented. Without her assistance, this thesis would still be unfinished.

Contents

1 Introduction **10**

1.1 Implicitly Parallel Languages 10

1.2 Program Execution 12

1.2.1 Sequential Language Execution 12

1.2.2 Parallel Language Execution 13

1.3 Parallel Execution on Monsoon 13

1.4 Overview 16

2 Multiprocessor Frame Management **17**

2.1 What is an activation frame? 17

2.2 Frame management 18

2.2.1 Frame Management for Sequential Languages 18

2.2.2 Frame Management for Parallel Languages 19

2.3 Goal: Application Speed 20

2.4 Frame manager characteristics 20

2.4.1 Latency and Throughput 20

2.4.2 Load balancing 21

2.4.3 Fragmentation 22

2.4.4 Memory Usage 22

2.5 Design Space 22
2.5.1 Resource Set 25
2.5.2 Number of Resource Sets 25
2.5.3 Frame Sizes 25
2.5.4 Load Balancing 26
2.5.5 Frame Bucket Datatype 27
2.5.6 Where the Load is Balanced 27
2.5.7 Where a Frame Request is Processed 28

3 Monsoon **30**

3.1 Dataflow Background 30
3.2 Monsoon: a Dataflow Processor 32
3.2.1 Explicit Token Store 33
3.2.2 Microcode 35
3.2.3 Hardware Hazard 38
3.2.4 Statistics 39
3.2.5 Threaded Code 39

4 Frame Management for Monsoon **41**

4.1 Monsoon's Effect on Frame Management 41
4.2 Black-box functionality 42
4.3 Multiple frame sizes 43
4.4 Dynamic allocation of frames 43
4.5 Single Frame Allocation 44
4.6 Frame Manager Evolution 44
4.7 Prototype Frame Managers 46
4.7.1 Split-Phase Get-Frame 47
4.7.2 Queue frame manager 47
4.7.3 Linked-List Frame Managers 49

5 Performance **52**

5.1 The Benchmarks 53

5.1.1 Fibonacci 54

5.1.2 Matrix Multiply 54

5.1.3 Gamteb 55

5.1.4 Simple 56

5.1.5 Paraffins 57

5.2 Run-time Parameters 57

5.3 Data Format 58

5.4 Frame Manager Time 59

5.5 Efficiency of Memory Usage 61

5.6 Load Balancing Results 64

5.7 Accounting for the Idles 66

5.7.1 Lack of Parallelism 66

5.7.2 Load Imbalance 66

5.7.3 Startup and termination latencies 67

5.7.4 Hardware Hazard 68

5.7.5 Idles in Multiprocessors 68

5.7.6 Categorizing idles 69

5.8 Gamteb 70

5.8.1 Frame Manager Analysis 72

5.9 Matrix-Multiply 72

5.9.1 Frame Manager Analysis 73

5.10 Paraffins 76

5.10.1 Frame Manager Analysis 79

5.11 Simple 80

5.11.1 Frame Manager Analysis 80

6 Discussion and Future Directions **83**

6.1 Future Directions 83

6.2 Related Work 84

7 Conclusion **85**

List of Figures

1.1 Id code for Fibonacci 11
1.2 Id code with a Data Dependency 11
1.3 Dynamic Call Tree for $fib(4)$ 14
1.4 Sequential execution of $fib(4)$ where activation frames are stack allocated 14
1.5 One possible parallel execution of $fib(4)$ 15

2.1 A Sequential Stack During a Procedure Call 19

3.1 Monsoon Processing Pipeline Overview 32
3.2 Activation Frame and Code 34
3.3 Token waiting and matching 35
3.4 Instruction Decoding Tables and Maps 36

4.1 Frame manager genealogy 45

5.1 FIBONACCI colors, n = 17 62
5.2 FIBONACCI opmix, n = 17 63
5.3 GAMTEB-9 colors, 40000 particles 70
5.4 GAMTEB-9 opmix, 40000 particles 71
5.5 Matrix-Multiply colors, 500x500 74
5.6 Matrix-Multiply opmix, 500x500 75
5.7 Paraffins colors, n = 22 77

5.8 Paraffins opmix, n = 22 78

5.9 SIMPLE 100 iterations, 100 x 100 81

5.10 SIMPLE 100 iterations, 100 x 100 82

Chapter 1

Introduction

This thesis describes the design, implementation and performance evaluation of a set of frame managers written for the implicitly parallel Id language running on the Monsoon multiprocessor system. This chapter will examine our paradigm for executing parallel programs. We will then introduce Monsoon our target experimental dataflow machine.

1.1 Implicitly Parallel Languages

In sequential programming languages, textual code order defines execution order. A line of code that textually precedes another will execute first. In *implicitly-parallel* languages, however, code order does not define execution order. Execution order is constrained only by *data dependences* which occur when the result of one instruction is an argument to another instruction. When an instruction's data is available, that instruction is ready to execute. *Strict* implicitly parallel languages, such as SISAL[4], are data-driven within each procedure but are not data-driven across procedure boundaries. In other words, all arguments to a procedure must arrive before the procedure can start. Limiting a language in this way makes it easier for the compiler to produce more efficient code. *Non-strict* implicitly parallel languages are data-driven across procedure boundaries — procedures can start (if there is work to do) when any argument arrives. Though compiling non-strict

```
def fib n =
  if n < 2 then 1
  else fib (n-1) + fib (n-2);
```

Figure 1.1: Id code for Fibonacci

```
def test n =
  { b = a * 2;
    a = n + n;
  In
    b};
```

Figure 1.2: Id code with a Data Dependency

implicitly parallel languages is especially challenging, non-strictness allows the maximum amount of parallelism and is often convenient for the programmer.

Id[21, 20, 22] is a non-strict implicitly parallel language being researched at MIT. An implementation of Fibonacci in Id is shown in Figure 1.1. Because there are no data dependences between the recursive calls to Fibonacci, it is possible for both recursive calls to be *active* simultaneously. For example, if we called *fib* with 2 as its argument, potentially three invocations of *fib* may be active simultaneously. It is also possible for either of the two recursive calls to start and/or end before the other starts/ends.

A simple example illustrating data dependency can be found in Figure 1.2. In this example, the first line cannot execute until the second line has completed execution. The multiply on the first line must wait for the add on the second line to complete since it needs the value *a* produced by the second line. In a given block we can only assign a variable once; it would be impossible to tell which version of a variable was valid if the variable was assigned more than once in a single block of code.

An important consequence of a non-strict, implicitly parallel language is the fact that many procedures may be active simultaneously. This fact places a new burden on the management of local storage for these procedures. As we shall see, stack allocation is insufficient and more general management of local storage is required.

1.2 Program Execution

Like all languages that support re-entrant code, some state (called a *frame*) must be associated with every invocation of a procedure. To call a procedure, the parent procedure first obtains a frame. Arguments are then sent to the frame and the child procedure is started (in a non-strict implicitly parallel language, the child starts when the first argument arrives.) When the child has finished, it returns its results to the parent (except where tail calls are involved.) The parent procedure then deallocates the frame in which its child ran. Blocks of memory that are shared between different procedure invocations are allocated in the heap. The heap manager, however, is not in the scope of this thesis.

1.2.1 Sequential Language Execution

Figure 1.3 shows the dynamic call tree of Fibonacci when $n = 4$. The tree shows all the frames allocated and used by the program run. The dynamic call tree is the same whether we execute it using a sequential language or a parallel language. The difference between sequential language execution and parallel language execution is *how the tree is traversed.* Sequential languages have only one procedure active at any given time. A child procedure *must* return control to its parent before the parent can continue to execute. Since only one invocation is active at any time and the active invocation is the latest procedure call, frames can be managed by a simple stack. Figure 1.4 shows the stack every time a frame is allocated.

Stacks have been used for sequential language frame management for some time[1]. The compiler inlines the necessary stack management. Stack frames contain linkage back to the parent procedure and arguments passed to the called procedure. The rest of the frame is used to store temporary procedural variables.

1.2.2 Parallel Language Execution

In contrast to a sequential language, during the execution of an Id program, any procedure may have one or more active child procedures. Child procedures do not have to terminate in the same order in which they were invoked. It is possible that any subtree of frames containing the root frame to be *extant* or active at any time. An extant frame is a frame that is allocated and that might be active. There is a tree of executing invocations in the parallel case, rather than just a single executing invocation in the sequential case. A tree of executing invocations is a very intuitive way of exploiting parallelism. Figure 1.5 shows some checkpoints of a possible traversal of the dynamic call tree for *fib* where $n = 4$. Note that all of the subtrees include the root frame.

Because frame managers for parallel languages must manage a tree-like structure of frames, they are essentially heap managers. A general heap manager is likely to be too expensive to run as a frame manager, however, motivating a special case frame manager. By placing a few restrictions on the frame manager, we can make it much more efficient than a heap manager. In addition, the decision of *where* to allocate a frame (i.e., on which processor) has a direct effect on the distribution of computation across the system. Thus, a frame manager is also different than a heap manager to the extent that it allocates storage and affects load balance.

1.3 Parallel Execution on Monsoon

Monsoon[23, 25, 9, 30, 24] is an experimental eight-stage pipelined parallel processing node designed at MIT and built by Motorola. Monsoon was designed to run Id programs in a dataflow style. Its pipeline stages are interleaved like the Denelcor HEP[29] — a single thread[1] can only execute one out of every eight cycles.

[1] A thread is a sequential sequence of code that executes uninterrupted. Interruptions only occur when waiting for synchronization or when a thread explicitly stops. Jumps do not cause interruptions.

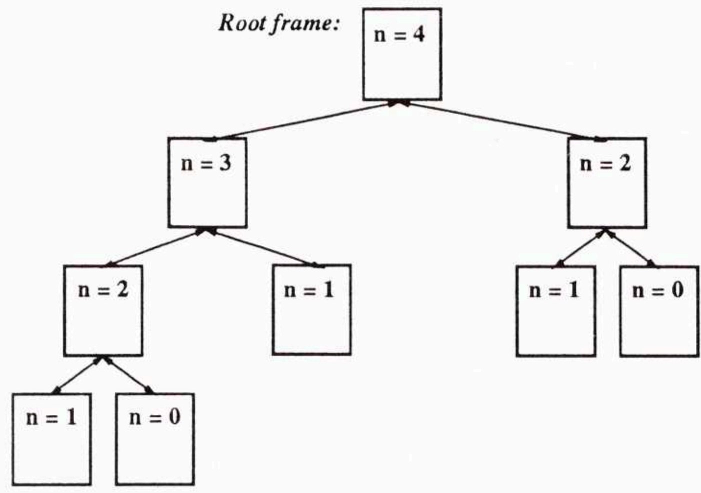

Figure 1.3: Dynamic Call Tree for $fib(4)$

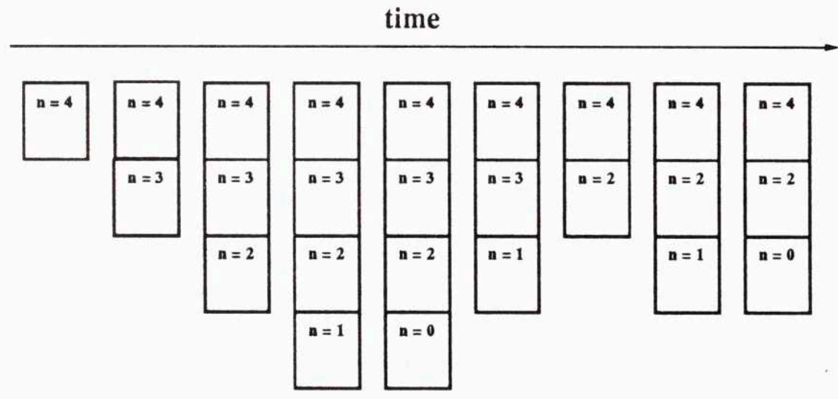

Figure 1.4: Sequential execution of $fib(4)$ where activation frames are stack allocated

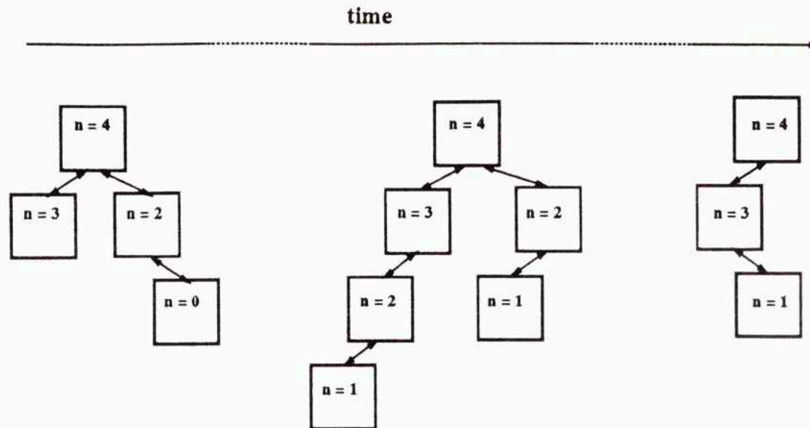

Figure 1.5: One possible parallel execution of $fib(4)$

Our largest configuration contains eight Monsoon processing elements and eight I-structure (remote memory) boards. We refer to a processing element or an I-structure board as a "node". The eight processor, eight I-structure configuration is the largest that will be built. The processors run at ten megahertz.

Monsoon processes *tokens* that are packets containing a continuation consisting of an instruction-pointer, frame-pointer, processor number and a value. The value can be thought of as an accumulator register. In a given procedure call, the frame pointer remains constant. Offsets from the frame pointer are encoded into the instruction stream. When a token leaves the bottom of the pipeline, it is automatically passed to the correct processor be it the same processor or a different one. Please refer to Chapter 3 for a more complete description of Monsoon.

On Monsoon, a frame is currently the smallest quantum of work. Frames exist on a single processor and are the smallest unit of load-distribution. Frame managers return full continuations — a node number is part of a continuation. Thus, load balancing is part of frame management, since each frame lives on a single processor and frames are the smallest quantum of work.

A frame is required for every executed iteration of a loop body. There are three loop schemas available in the Id language: sequential, unbounded and bounded. The

sequential loop schema allocates a single frame for all iterations of that instance of the loop. The *unbounded* loop schema allocates a new frame for every iteration of an instance of a loop. The *bounded* schema allocates a set number (determined at run-time, usually as a parameter input by the user) of frames — all iterations of a instance of the loop execute within those frames. Bounded loops are used to control the amount of work available within the system so that the program does not overrun the available resources.

Our frame managers are called via trap instructions inserted into compiled Id code. They are all written in MONASM, Monsoon's assembly code. Frame managers have a single goal — to allow application programs to run as efficiently as possible. Efficiency is achieved by good load balancing and low frame manager latencies and critical sections. Reducing memory fragmentation is a secondary concern. Although better use of frame memory will allow more parallelism to be exposed, a real machine can only exploit a certain amount of parallelism at a time. Therefore, as long as a frame manager effectively runs application code, memory fragmentation is ignored.

Our experience indicates that round-robin load distribution coupled with low latency frame management performs best over a range of benchmarks. Overall, simplicity and speed win out over more sophisticated and time-consuming schemes.

1.4 Overview

The next chapter discusses the frame management problem for parallel languages in greater detail. We examine problems faced by all frame managers for parallel languages, independent of the language and the system on which it is running. We then discuss the Monsoon system for which we designed and implemented frame managers. We explain the frame managers we have implemented and some of our design decisions. The benchmarks are then defined and results are presented along with analysis. We close with future work, related work, and conclusions.

Chapter 2

Multiprocessor Frame Management

This chapter explores the activation frame management problem for multiprocessors. We start by defining what a frame is, then explain frame management for sequential languages. Frame management for parallel languages is discussed, followed by a section on the principle goals of a frame manager for a parallel language. Lastly, we discuss important frame manager characteristics, and the design space of a frame manager for a parallel language.

2.1 What is an activation frame?

A *frame* is a block of memory used to store data associated with the invocation of a procedure[1]. A frame stores both procedural variables and linkage data required by the procedural calling convention. When a procedure finishes, its frame is returned for reuse. All of the data stored in the frame is ignored when the frame is reallocated, essentially erasing the data. That is, the lifetime of the data stored in a frame is the same as the lifetime of the invoked procedure. If the data lifetime was longer than that of a procedure, that data should be stored in a heap.

[1]Although the user defines procedures, the compiler makes the final decision as to what blocks of code make up a procedure.

2.2 Frame management

Frame allocation is easy if deallocation is unnecessary. For all but the smallest program runs, however, frame memory must be reused — a program will run out of memory otherwise. Frame reuse implies a frame manager. Frame management can be implemented in many different ways. The simplest method to implement (assuming a garbage collector) is to allocate with abandon and garbage collect/compact when necessary. Since we know precisely when a frame is allocated and when a frame is deallocated (when a procedure is invoked and when a procedure terminates, respectively) the compiler can easily generate code to allocate and deallocate frames. Virtually all sequential languages work this way. Our Id compiler also generates activation frame allocate and deallocate code. Frame managers for sequential languages are significantly different, however, from frame managers for parallel languages.

2.2.1 Frame Management for Sequential Languages

As explained in Section 1.2.1, frame management for sequential languages is simply a matter of incrementing and decrementing a stack pointer. A possible procedure calling convention, shown in Figure 2.1, requires a base frame pointer BASE and a top-of-stack pointer TOS. When a procedure is called, its arguments are pushed onto the stack in a specific order, along with the instruction and frame base pointer to return to when the called procedure is finished. The TOS and BASE pointers are incremented by appropriate values. The calling procedure then jumps to the procedure being called which reads its parameters from negative offsets from the BASE. The callee procedure finishes its computation, which may include procedure calls, puts its result(s) onto the stack, and returns control to the calling procedure by resetting the instruction pointer and BASE. The caller can then reset the TOS. Generally the BASE and the TOS pointers are stored in registers. Of course there are variations of this calling convention — the callee, rather than the caller, could reset the TOS or registers could be used to pass arguments for instance. The basic algorithm, however, is the same for all sequential languages.

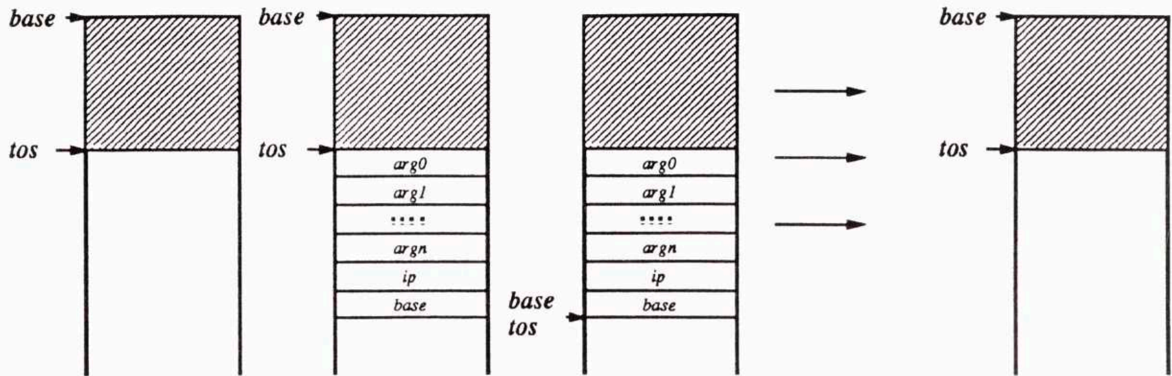

Figure 2.1: A Sequential Stack During a Procedure Call

2.2.2 Frame Management for Parallel Languages

Because frames for parallel languages may not, in general, be deallocated in the reverse order that they were allocated, frames cannot be managed by a simple stack. A solution more similar to heap management is required. System designers may also decide that a frame corresponds to the smallest quantum of work executing on a single processor. If a frame corresponds to the smallest quantum of work, the frame manager also distributes work. Load balancing with frames is rather logical in some sense. Frames correspond to code-blocks which roughly correspond to procedures. Programmers are used to trying to reduce the number of arguments passed between procedures which would cost in network traffic if each procedure invocation is executed on a different processor. Frames tend to have a reasonable number of passed arguments and are large enough to amortize load balancing costs but not so large that load balancing becomes difficult.

The complexity size of a frame manager for a parallel language makes it difficult to inline into compiled code. Frame manger services appear more like procedures themselves. Of course the longer the frame manager takes to run, the slower the total run-time. The rest of the chapter will discuss frame managers for parallel languages, the goals of their design, characteristics of the managers and a design space.

2.3 Goal: Application Speed

There is only one goal of a frame manager — to maximize application speed. To achieve this goal, a frame manager for a sequential language must minimize the number of machine cycles it consumes. A frame manager for a parallel language must not only minimize the number of machine cycles it consumes, it must also distribute work across the machine. These two requirements for parallel frame managers are coupled — better load balancing generally requires more overhead. Thus, the goal of the frame manager is to optimize application speed by trading execution overhead for load balancing performance.

2.4 Frame manager characteristics

Frame managers may be characterized by several metrics. The first set of metrics includes latency, throughput, and load balancing which apply directly to the goal (application speed) of our frame manager. Other metrics, fragmentation and how to store the waiting frames, are more implementation-specific but indirectly related to application performance. This section examines each characteristic.

2.4.1 Latency and Throughput

Latency is the amount of the time it takes to allocate/deallocate a frame. Throughput is the number of frame allocation/deallocations performable in a unit of time. Both affect application performance, but to varying degrees, depending on the application. Latency becomes more important if an application has low to moderate amounts of parallelism. The faster a frame can be allocated, the sooner the child procedure can start executing. Throughput is more important when there is lots of exposed parallelism. Higher throughput will allow more parallelism to be exploited in a shorter amount of time since the startup time will be shorter. Of course latency and throughput are coupled —

assuming a single set of resources, throughput is the inverse of latency. When the number of resources increases, however, throughput can increase with no decrease in latency.

Of course, when resources are finite, as in all real machines, one might not want to exploit all available parallelism since exploiting more parallelism requires more resources. One might argue for a frame manager with longer latencies and/or lower throughput to throttle parallelism. Our view, however, is that throttling parallelism is the scheduler's job — conceptually the frame manager is at a lower level.

2.4.2 Load balancing

Load balancing is difficult to quantify. Speedup is our basic measure of load balancing performance. Speedup is computed by comparing the number of cycles executed by a single processor running an application to the total number of cycles executed across multiple processors. We assert that a single processor executes with a minimum number of cycles, since no load balancing is necessary. Assuming a perfect architecture, an increase in cycle count from multiple processors is due to either a lack of parallelism, or a poorly balanced load. It is difficult, however, to differentiate idles caused by a lack of parallelism and the idles caused by load imbalance. One must be able to tell at any instance if some processors are working while others are idled *and* that there is enough work to keep all processors busy. We generally assume parallelism is sufficient and blame unexplained idles on less than ideal load balancing.

Conceptually, the smaller the unit of load to balance the easier to balance the load. It is easier to evenly distribute lots of sand than a few large boulders. It is also important that units of like size are distributed separate from units of different sizes. Distributing pebbles and rocks between buckets could have one bucket wind up full much before the others. Care needs to be taken when distributing work of varying sizes.

2.4.3 Fragmentation

Fragmentation[15] occurs when free memory is unusable. *Internal fragmentation* occurs when a block of memory larger than necessary is allocated — the extra memory is unusable until the object is deallocated. *External fragmentation* occurs when free memory is available but not in a usable form. Small blocks of free memory interleaved with blocks of allocated memory is an example of external fragmentation. The more efficient the memory usage, the smaller the memory requirements for a specific amount of exploited parallelism. More efficient memory usage means we can exploit more parallelism with the same machine. More exploited parallelism generally means better speedup.

2.4.4 Memory Usage

The memory requirements of the frame manager itself should be kept as low as possible as long as application speed does not suffer for it. Storing waiting frames can take a percentage of the frame memory, depending on how the frames are stored. Storing frames in linked-lists uses very little memory since only one word of extra storage per list (to store the head) is required. In a parallel system, all frames on a single list should be from the same processor to keep accesses to the list fast. If frames on a single list are from different processors, getting the next element in the list may require a remote memory reference. Using a queue to store returned frames reverses the advantages and disadvantages of a linked list - extra storage is used for the array but frames in a single queue can be from different processors.

2.5 Design Space

There are many degrees of freedom in frame manager design. Several design parameters are listed in Table 2.1. In this chapter, we will discuss each of these parameters in some

detail. In future chapters, we will describe implemented frame managers in terms of these parameters.

Clearly, we cannot cover all of the design space for frame managers, since the space is so large. If there was enough memory in the system, we could forego deallocation. Frame management would become very easy since we simply advance a pointer into free memory. We could write a frame manager that allocates from a large block of free memory, garbage collecting when necessary. We could also write a simple frame manager that would use a single free list and allocate frames in a first-fit or even a best-fit fashion. Load balancing, an orthogonal issue, introduces many options. To keep our experiments reasonable, we impose a few limits on the range of the parameters.

We assume that all frame managers will use a quick-fit algorithm to manage the frames. Quick-fit groups objects of the same size into a common "bucket". When an object of a certain size is requested, the correct bucket is determined and a frame is fetched from that bucket. The quick-fit algorithm is very efficient and is very well suited for frame management. It is powerful enough for heap management — in fact, our heap manager is also based on the quick-fit algorithm.

Unlike our heap manager, further restrictions are placed on our frame manager in order for it to run as quickly as possible. We constrain our frame managers by limiting the number of different frame sizes. If we did not have this constraint, frame management would be equivalent to heap management.

Before we discuss frame manager design parameters, we must first define some terms. $P_{request}$ is the processor from which the frame request originates. P_{frame} is the processor on which the to-be-allocated frame exists. P_{random} is a random processor and $P_{deterministic}$ is a specific processor set before run-time.

The rest of this chapter discusses some parameters of frame management design. Most of the important parameters are listed. Some combinations of parameter values that are listed were not implemented — it was decided that they had little chance of being efficient.

Category	Choices
Resource Set	r frame sizes
Number of Resource Sets	n per system
	n per processor
Frame Sizes	Static
	Dynamic
Load Balancing	Increment by l (round-robin), c counters
	Random
	Adaptive
Frame Bucket Datatype	Linked list
	Arrays
Where Load is Balanced	$P_{request}$
	P_{frame}
	P_{random}
	$P_{deterministic}$
Where Frame is Managed	$P_{request}$
	P_{frame}
	$P_{request} + P_{frame}$
	P_{random}
	$P_{deterministic}$

Table 2.1: Some Frame Manager Design Parameters

2.5.1 Resource Set

The composition of the frame manager's resource set consists of the number of allowable frame sizes. The simplest frame managers only allow one frame size. Every frame request receives a frame of that size. Unless all procedure calls require the same frame size, this scheme will produce much internal fragmentation. Depending on the number of resource sets (see Section 2.5.2), few frame sizes can also increase contention for frame management resources.

Of course, we could allow every possible frame size. That would mean there would be no internal fragmentation. The performance of the frame manager, however, would suffer on two counts. If no frames of a certain size are available, it may take a long time to get a larger sized frame. Also, because there are many different sized frames, it is very for a program to run out of a specific size frame. In this case external fragmentation can severely reduce the number of available frames a given size or larger.

2.5.2 Number of Resource Sets

The number of sets of resources indicates the number of sets of frame management resources. The most common number of sets is one per processor. It is conceivable, however, to have a larger number of resources per processor or just one set of resources for all the processors. It is difficult to share resources between different sets of resources quickly. More sets of resources, however, reduces frame management contention. Thus, the number of sets of resources is a tradeoff and depends on the length of the frame manager's critical sections and its ability to communicate with other sets of resources.

2.5.3 Frame Sizes

At some point, the frame sizes have to be chosen. It can be done either statically, before an application program is run, or dynamically, while an application is running. Frame

sizes do not have to be consistent across all processors; however, it is a lot easier to handle consistent frame sizes.

2.5.4 Load Balancing

Load balancing has been the subject of much research[11, 31, 17]. Load balancing can be done in one of two styles. The first style distributes the load based on the state of a single processor. Round-robin and random schemes fall under this category. The second style tries to figure out which processor is the least loaded by querying other processors. One algorithm in this style would poll every processor to see which one was the most idle every time a frame is allocated. This algorithm takes some time to run, however, and the load distribution may change before the frame is allocated. Algorithms requiring information from more than one processor will take more time than algorithms requiring information from only one processor. It is also unclear that instantaneous load information is an accurate predictor of future load.

All of our frame managers use round-robin load balancing. Our rational for using round-robin is simple — we expect much parallelism in the programs we run. In other words, we expect many procedures to be active at any given time. If procedures are being called all the time, it is logical for the load balancing algorithm to spread the work out as evenly as possible. Round-robin distribution of work operates like a rotating lawn sprinkler — it constantly squirts out units of work to each processor in a deterministic order. At best, $P_{request}$ balances the load that it produces across all the processors in the system. If all processors can balance their own loads across the system the system will be balanced. As long as there is enough work of small enough size to keep the distribution going and enough work to keep all of the processors busy, a round-robin scheme intuitively works.

A consideration for round-robin distribution is the number of round-robin counters and what each counter is associated with. It is possible for each processor to have a single round-robin counter. Although the counter is a bottleneck, it can be incremented within

a minimum critical section and thus will not reduce throughput by much if any. It is also possible for each frame size to have a round-robin counter which will, in general, improve load balancing over the single round-robin counter case. The improved load balancing comes from the fact that frames of the same size will often execute procedures with approximately the same amount of work. Each round-robin counter, then, distributes quanta of work that are about equivalent in size, unlike the single counter case that distributes quanta of work of varying size. For example, if a round-robin scheme tries to allocate frames containing very different amounts of work with the same counter, it is possible for one processor to get most of the work. One could also associate a counter with every procedure. There can be any arbitrary number of counters — the only problem is choosing which counter to use.

Another possibility for load balancing is choosing P_{frame} randomly. It is possible that a random scheme will do better than a round-robin scheme if one or more round-robin counters are distributing work of varying size or there is not enough parallelism to make effective use of round-robin distribution.

2.5.5 Frame Bucket Datatype

We can store ready frames in either an array or in a linked-list. An array can be accessed with a single indirect memory access while a linked-list requires two indirect memory accesses. Linked-lists, however, do not require the additional memory that arrays require.

2.5.6 Where the Load is Balanced

When a frame request is made by $P_{request}$, P_{frame} must be determined. In order for the frame manager to run quickly, the algorithm for choosing P_{frame} must be very simple. A fast distributor of work that does not balance the load well, however, will do very poorly overall. The processor that determines P_{frame} depends somewhat on the load balancing

algorithm. If a random or round-robin load balancing scheme is used we usually have $P_{request}$ decide P_{frame}.

Yet another possibility is to have a processor $P_{deterministic}$ that balances the load. Every frame request would be sent to that processor which would decide which processor the requested frame would live on. The advantage of using a single $P_{deterministic}$ is that all of the load balancing information is local and thus a good distribution of work can be done quickly. The disadvantage of this scheme is that $P_{deterministic}$ is a bottleneck, since *every* frame request must go to that processor.

2.5.7 Where a Frame Request is Processed

When processor $P_{request}$ requests a frame from the frame manager, it eventually gets one of P_{frame}'s frames. Frame manager code must run somewhere. It is possible to process the entire frame request on $P_{request}$. This technique, in its simplest, most efficient form, starts out with each processor having a set of frames it can allocate from every processor in the system. Satisfying frame requests on $P_{request}$ has a latency advantage — no network access is made to allocate a frame. If a processor runs out of frames to allocate, however, it is difficult to request frames from another processor. Since each processor, assuming an even distribution of frames, has the number of frames that exist on a single processor, it can only allocate a few frames compared to the total number of frames in the system.

It is also possible to do the actual frame allocation on P_{frame}. Every time $P_{request}$ wants a frame from P_{frame}, $P_{request}$ must send a message to P_{frame} requesting that frame. P_{frame} must send a message back containing a pointer to the requested frame. This technique increases both network traffic and the latency of getting a frame, but also permits any processor to potentially allocate all frames in the system.

Yet another option is to have part of the frame management done on $P_{request}$ and the other part of the frame management done on P_{frame}. For instance, a small pool of frames could be managed by $P_{request}$. When $P_{request}$ runs out of frames from P_{frame}, a

request is sent to P_{frame} to send back more. This technique combines the advantages of our two previous schemes and is akin to catching frames for further use. The cache size is an important number to consider.

One could also envision sending the frame request to some random processor P_{random} or a deterministic processor $P_{deterministic}$. In the latter case, perhaps one processor in the system would contain all of the frame management information. There would be, however, a bottleneck through $P_{deterministic}$. The first two methods, $P_{request}$ and $P_{request} + P_{frame}$, are the most logical since the frame management is distributed and unnecessary network traffic is avoided.

Chapter 3

Monsoon

Since we tailored our frame managers for the Monsoon processor, it is important to describe how its architecture and how it might affect our code. This chapter will examine the dataflow paradigm and the Monsoon processor and system architecture, focusing on areas that affect the frame manager. This chapter borrows heavily from Chiou[6].

3.1 Dataflow Background

Dataflow is a relatively old idea in the field of computer architectures. It was first proposed by Dennis and Misunas[10] and later refined by a number of projects including the Manchester Machine[12], Sigma-1[28], and TTDA[3]. The basic concept behind dataflow is that execution is *data-driven* rather than instruction-driven. An instruction executes only after all the data it requires have been received. Conventional computers, on the other hand, execute instructions in a specific sequential order with little run-time regard to data dependences. It is evident that dataflow computers have the potential for exploiting large amounts of parallelism inherent in many applications, and that data-driven execution readily extends to parallel architectures.

The standard abstraction of a tagged-token dataflow machine was developed independently by groups at Manchester University of Manchester, England, at the University of

California, Irvine, and later refined by the Monsoon processor, where associative memory has been replaced by explicit memory storage techniques. Data travels through a dataflow machine in packages called *tokens* consisting of data along with control information, called *tags* (also called *continuations* in Monsoon), which encode such data as the destination instruction. The advantage of using a tag is that each instruction is synchronized on its own — it requires no external information. Thus, it is easier to keep processors busy using tokens with tags than tokens without tags.

The basic operation of a tagged-token dataflow processor is described. A token enters a processor and checks for its partner, presumably by checking for equal tags. If the partner is present, it is fetched and the instruction is executed. Obviously, if the instruction is unary, such as NOT, the processor does not need to check for a partner. The part of the machine that checks for partners is generally called the *waiting/matching* area. Newly calculated data is encapsulated into tokens that are sent back into the system. A dataflow processor cycles through tokens until an answer is produced.

Though dataflow is intuitive, efficient ways of implementing it are not so intuitive. Some problems with dataflow are the necessity of immutable data-structures to prevent out-of-order errors, large, fast waiting/matching sections, garbage collection of objects being referenced in an out-of-order fashion, deadlock resolution, and so on. Efficient dataflow processors are thought to require special hardware to support the dynamic scheduling associated with the computation model. Though implementation challenges exist, however, the promise of exploiting nearly all parallelism within an algorithm is too good to pass up.

A few dataflow machines have been built, the most notable being the Sigma-1[28], EM-4[16, 27] and the Manchester Dataflow Machine[12]. Various design details, however, hamper these machines. A major detail is the implementation of the waiting/matching area. The simplest waiting/matching area has fully-associative memory so that partner searches take unit time. Fully-associative memory is much too expensive for any reasonably sized waiting/matching area, so cumbersome hashing techniques are generally used. Papadopoulos[23] provides an elegant solution to the matching problem in the

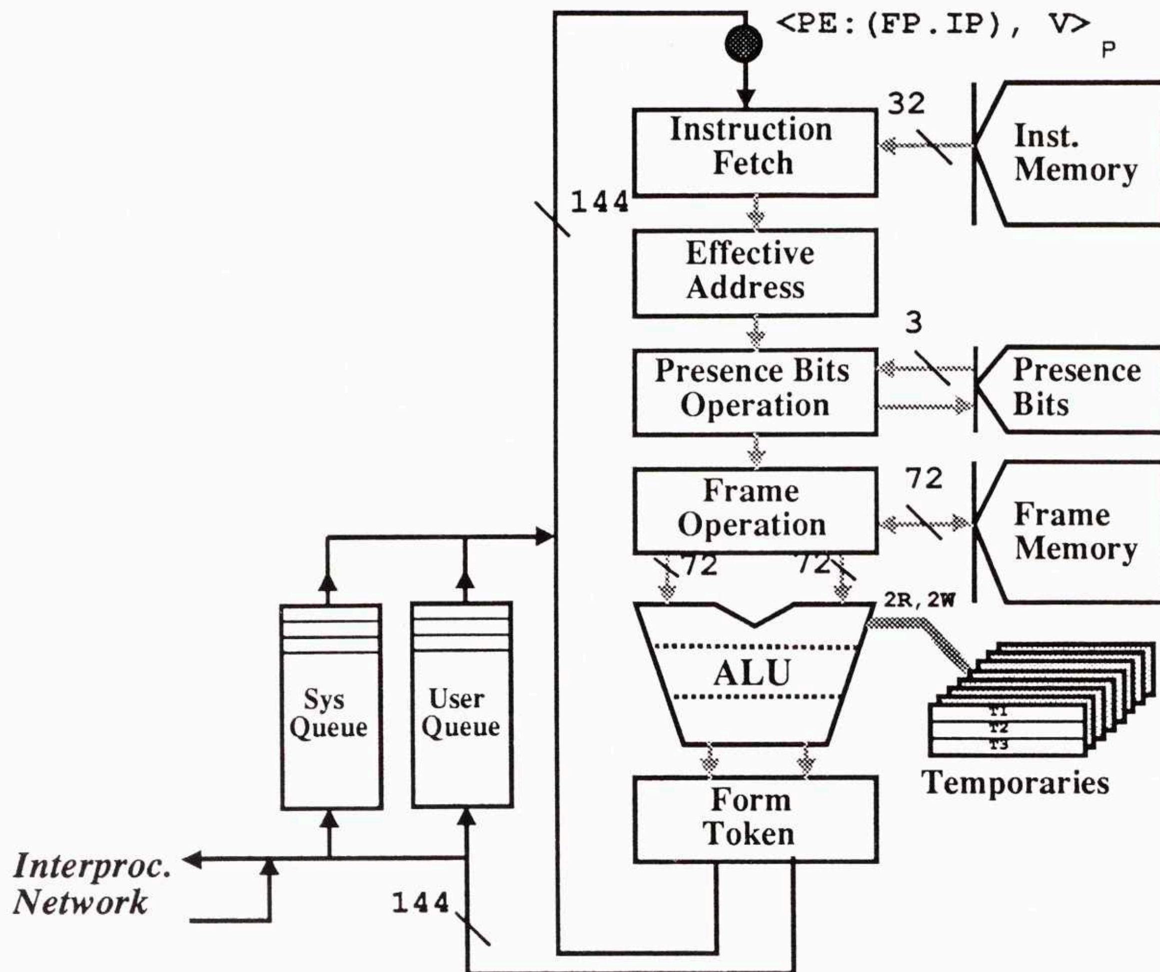

Figure 3.1: Monsoon Processing Pipeline Overview

implementation of Monsoon, a pipelined dataflow processor.

3.2 Monsoon: a Dataflow Processor

Monsoon is a fully pipelined dataflow processor — thus, none of its stages can take more than unit time to execute (there are a few exceptions, but they rarely occur.) Monsoon's basic architecture, some of its unique features, and an overview of its microcode decoding are described in this section. This discourse follows Papadopoulos[23].

The basic stages of Monsoon, as shown in Figure 3.1, are *instruction fetch*, *effective*

address generation, *fetch presence bits*, *operand fetch and/or store*, *ALU/FPU and next address generation*, and *form token.* The operation occurring in any stage of the pipeline is completely independent of the operations occurring in the rest of the pipeline. Each stage is also non-blocking except for some exceptions due to read/writes to memory and long floating-point operations. Though the pipeline can block to permit long operations to finish, the pipeline never needs to be flushed since each pipeline stage is independent of the others. This is especially advantageous for branches and hazards. Unlike standard pipelined RISC processors that must flush their pipelines whenever a wrong control path is taken, Monsoon never wastes any of the processing done on a token.

3.2.1 Explicit Token Store

The major innovation of Monsoon is its implementation of the waiting/matching area. The technique, called Explicit Token Store (ETS), allows waiting/matching to be done in unit time. This is accomplished by activation frames. Activation frames are very similar to stack frames found on conventional computers. The base address is known, and variables are referenced by offsets to that base address. Activation frames are created when a codeblock is executed and contain all necessary matching locations along with their presence state bits. Since each token carries its base frame pointer and fetches its frame offset with its instruction, it knows exactly where it should look for its partner. Activation frames, besides making waiting/matching quick, also reduce the amount of memory required. Instead of storing the entire token, including the tag, only the data needs to be stored (recall that tags are normally used for matching.)

Implementing activation frames requires pointers to the frames within the tags of the tokens. fp represents such a pointer. $ip : fp$ is often written to represent the instruction/frame pointer combination found in a Monsoon tag.

An activation frame and its corresponding program text are shown in Figure 3.2. Notice that the $*$ depends on the $+$, and the $-$ depends on both the $*$ and the $+$. Monsoon fetches its instruction that tells it the matching location of the token in terms

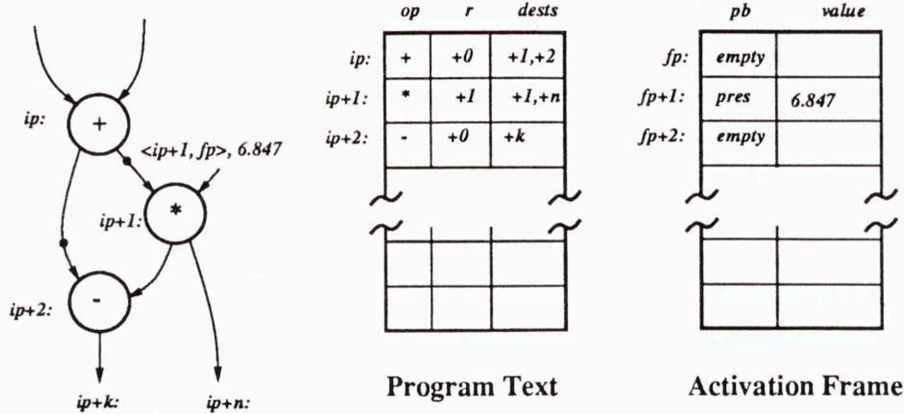

Figure 3.2: Activation Frame and Code

of an offset to the activation frame base pointer, and up to two destinations for its result tokens. An interesting point to notice is that both the + and the − instructions use the same matching location. This is possible because the − instruction can only receive input tokens after the + instruction is done and has reset the presence bits to *empty*.

Another dataflow innovation found in Monsoon is its technique for token matching. Monsoon uses presence bits to indicate the state of a particular matching location in an activation frame. An example of token waiting/matching is shown in Figure 3.3. In the standard tagged-token dataflow model, the check is done on the tags of the tokens that must be stored along with the data. Monsoon checks presence bits associated with the partner's location rather than checking for valid data. If the partner is present or the operation is unary, the instruction is fetched and applied to the data (Figure 3.3, part c.) Otherwise the data is written to memory where it will wait for its partner (Figure 3.3, part b.) Thus, the basic order of operations is as follows.

- The first token enters its processing element and checks the presence bits of its predetermined meeting point for the state of its partner. Since the state is *empty*, the token knows that its partner has not yet arrived. The token is written to the specified meeting place, and the presence bits of the location are mutated to a *present* state.

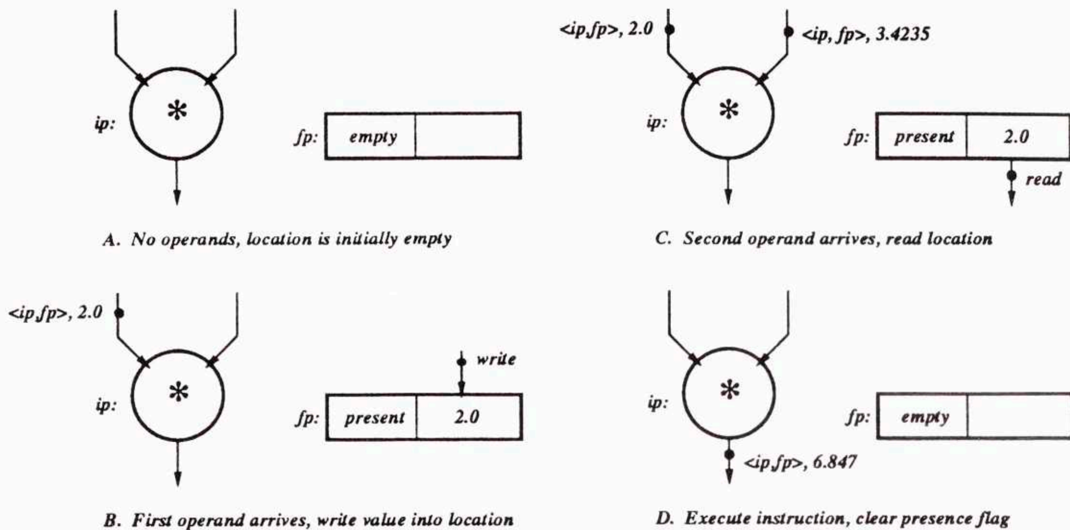

Figure 3.3: Token waiting and matching

- The second token enters the processor some undetermined amount of time afterwards. It checks the presence bits of the predetermined meeting location and finds that the *present* state is set. Thus, it knows that its partner is available, fetches that partner and executes the instruction. The presence bits are reset back to *empty*. This reset is important if the location in the activation frame is to be reused.

This matching technique has a couple of advantages. The first benefit is that the data path and the hardware necessary to check on a match are much less than a system that compares entire tags. Of course, knowing where to look for the partner (ETS) is crucial to this advantage. The second benefit is that a stored token can have more than two states of presence. This is especially advantageous for complicated instructions, such as gates with two triggers.

3.2.2 Microcode

For experimental purposes, Monsoon's pipeline is controlled by a microcode (refer to Figure 3.4.) The microcode is not microcode in the traditional sense of instructions for a

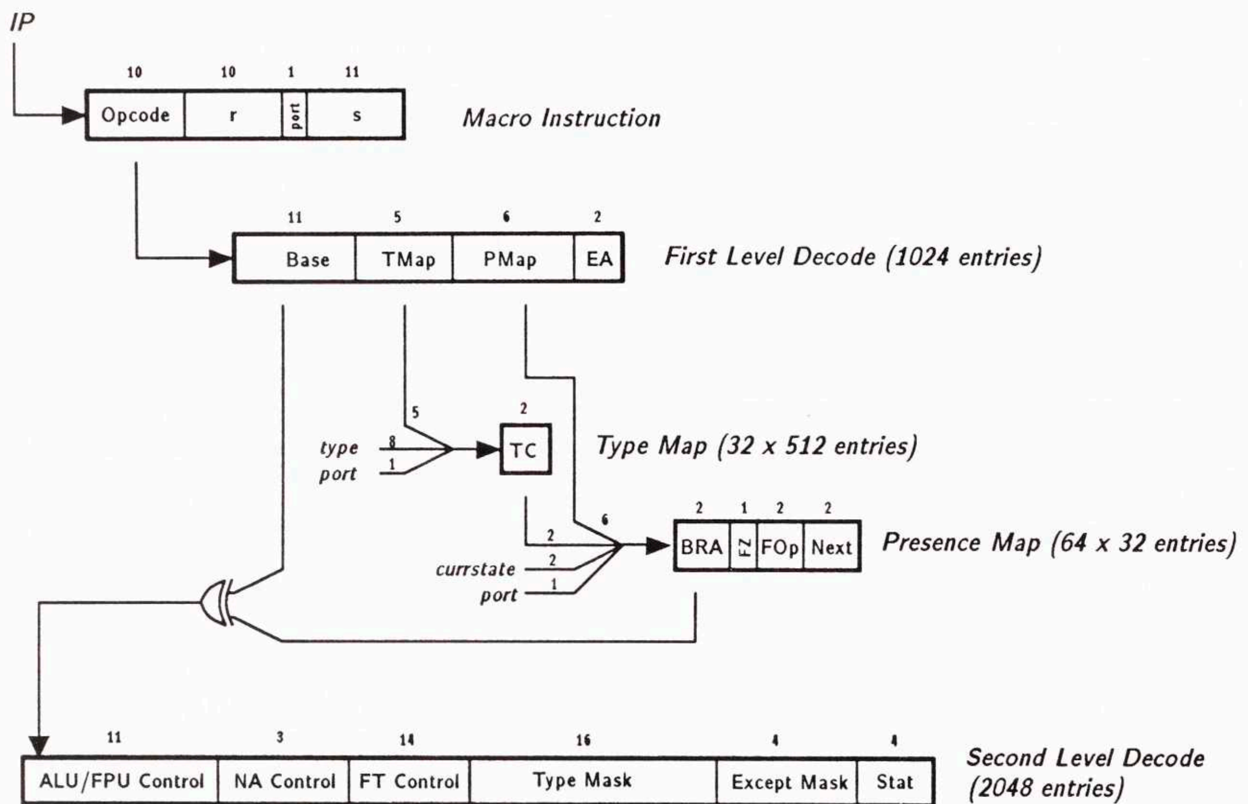

Figure 3.4: Instruction Decoding Tables and Maps

microengine processor. Monsoon's microcode controls the actions of each of the pipeline stages. For example, the microcode controls how a token is to be constructed, which functional unit is to execute the instruction, and what memory operations are to be performed. The specific microcode that will be run on a specific token is decided by the instruction pointer inside the token, the type of value of the token, the port of the token, and the presence bits of the ETS memory associated with that token.

The decoding process starts with a token coming into the processor. The token consists of the fields of data as shown below.

Token			
Tag-part		Value-part	
TYPE 8	TAG 64	TYPE 8	VALUE 64

The structure of the tag is shown below.

TAG				
PORT 1	MAP 7	IP 24	PE 10	FP 22

where,

PORT Indicates whether the token is coming on the left port (*l*) or the right port (*r*) of the instruction specified by PE:IP. This field is called INPUT-PORT throughout the rest of the document.

MAP Alias and interleave control. Increments to FP affect PE as specified by MAP. (This field is presently not operational.)

IP Instruction pointer. The absolute address of an instruction on processor number PE.

PE Processing element number. For machines with less than 1024 physical processors, the LSBs of PE can be concatenated with the MSBs of FP, extending the physical address space of each PE

FP Frame pointer. The absolute address of a 72 bit location on processor number PE. PE:FP describes a global address, so a machine is limited to a maximum of 4000 megawords of physical memory.

Once the token enters a processor, its instruction is fetched by reference to its IP (refer again to Figure 3.4.) The instruction consists of a OPCODE, offsets r and s and the OUTPUT-PORT. The OPCODE references to the first level decode (1ST-LEVEL-DECODE) table of the microcode. The 1ST-LEVEL-DECODE is comprised of a base pointer (BASE) to the second level decode tables, a type map (TMAP) pointer, a presence map (PMAP) pointer, and an effective address mode (EA.) Once we have the 1ST-LEVEL-DECODE, the proper TMAP is referenced using the TYPE of the token's data and its PORT, producing a two bit type code (TC.) The effective address is computed in parallel with the computation of TC and is used to read the presence bits (CURRSTATE) associated with the activation frame matching location. TC, CURRSTATE, and the INPUT-PORT of the token are used to reference the proper PMAP from which a presence map entry (PENT) is obtained. A PENT consists of an offset to the BASE of the second level decode (BRA), a force-to-zero bit (FZ) which forces the BASE to zero if true, a fetch and/or store operation (FOP), and the next state (NEXT) of the effective address location.

The second level decode table is referenced by the BASE, which is forced to zero if FZ if true, added to BRA. The returned result, 2ND-LEVEL-DECODE, contains information to control the functional units, the next address unit, and the form token unit. The pipeline can finish processing the token with the information from the 2ND-LEVEL-DECODE.

See Papadopoulos's[23] Appendix for details on the microcode.

3.2.3 Hardware Hazard

There is a structural hazard in Monsoon that creates an idle whenever an instruction produces a network token and no recirculating token. The end of the pipeline feeds a bus that is connected to the network and to the token queues (see Figure 3.1.) That bus is a bottleneck, only permitting one transaction per cycle. When an instruction produces a single token destined for the network, it is impossible for the processor to obtain another token from the token queues or the network. The hazard forces an idle in this case. Another way this bottleneck may cause an idle is when the processor reads a token from

the network. The processor will try to fill the idle pipeline slots from the network. If the pipeline is full, however, the processor will take a network token periodically, pushing the token it displaces onto a token queue. The same hazard prohibits pushing a token on a queue and reading another from the network on the same cycle — when the token is pushed on the queue, an idle is created.

Clearly, compiled code could use a recirculating token to avoid idles caused by the hardware hazard. Our compiler, however, currently does not generate such code.

3.2.4 Statistics

Every cycle, Monsoon increments one of 64 statistics counters. The statistics counter to increment is determined by the microcode of the opcode being executed and a `color` tag, found on every token. The `color` tag is set by the frame manager whenever a frame is allocated. By examining the statistics registers, one can account for all cycles executed on every Monsoon processor.

Each procedure can be assigned a `color`. The statistics can also be separated into different colors, effectively allowing dynamic procedure profiling with only the overhead of setting the `color` tag every time a frame is allocated. Colors can be inherited from the calling procedure, increasing the usefulness of coloring.

3.2.5 Threaded Code

Critical tokens, which are created by a special subset of the instruction set (distinguished by a bit in the microcode), creates a critical thread[26] which is very useful for system code. Critical threads have deterministic behavior since they are not permitted to do any delayed synchronization — only spinning waits are permitted. Thus, we can write critical sections into critical threads to avoid some forms of deadlock. Of course we still have to be careful of deadlock and livelock, but not as careful as we would have to be running in non-threaded mode.

The interleaving in Monsoon makes atomic operations difficult. We can create special instructions that allow atomic operations using presence bits. The problem is that to do even a simple read-modify-write operation requires 16 cycles to complete since only one "operation" is performed every eight cycles. Multiplying critical sections by eight forces us to minimize them as much as possible. Some special instructions are written to make critical sections as low as eight cycles but none of our run-time systems can take full advantage of them since they must refer to a specific, hardware memory location.

Threaded code gives the assembly programmer additional "ephemeral" state. Each of the eight interleaved threads has both a frame and a set of three registers associated with it. As long as a thread keeps its position in the pipeline (stays critical), it can use this ephemeral state. Using ephemeral state requires critical instructions and excludes delayed synchronization.

Chapter 4

Frame Management for Monsoon

This chapter focuses on frame managers for the Id language running on Monsoon processors. The first section reiterates constraints imposed by the Monsoon architecture. Subsequent sections discuss the evolution of our frame managers and the lessons we learned during the development process. Results follow in the next chapter.

4.1 Monsoon's Effect on Frame Management

Monsoon's architecture strongly affects aspects of the design and implementation of our frame managers. Interleaving, which multiplies critical section lengths by eight, greatly influenced the design of the instruction set as well as the frame manager. Special instructions were developed to reduce critical sections. Code is carefully tuned to reduce possible contention and minimize resource usage. Atomic operations depend on the presence bits to define locked/unlocked states. Monsoon's interleaving increases compiled code throughput but reduces threaded (read run-time system) code throughput because of the expansion of critical sections.

We must use ephemeral state (frames and registers) to write efficient frame managers. No instructions are executed to allow access to the registers and only one instruction is executed to open access to an ephemeral frame. Any other scheme to get temporary

space would require more instructions and would essentially be a frame manager request. It is difficult to allocate frames when a frame must be allocated to do so. Thus, we are forced to use threaded code to use the ephemeral state. It is unfortunate there are only three registers available for each executing thread — registers must often be spilled, slowing computation.

Threaded code on Monsoon has both advantages and disadvantages. Its advantages are that it is relatively easy to write critical sections and the execution time is deterministic. The disadvantages are that it is impossible to do remote fetches and hold ephemeral state and it is possible to livelock since a critically recirculating token cannot be forced out of the pipeline. When a thread wants to start another thread, all necessary state must fit in a single token — synchronization is not allowed.

Weak addressing modes affect instructions that do indirect memory references. An indirect local store takes three cycles while an indirect local read takes two cycles. The addresses must be fully computed for stores — one cannot automatically add an immediate value to the addresses before storing. These cycle penalties increase frame management latencies substantially.

4.2 Black-box functionality

An Id procedure gets a context by executing the SVC0 trap instruction, passing in a pointer to the *code-block descriptor* (CBD) of the procedure that will run in that frame. The code block descriptor contains necessary information about the procedure to be called such as its instruction pointer, its statistics color, and the frame size it needs. The frame manager will select a frame that is at least the desired size and return an instantiated pointer to that frame. "Instantiated" means that pointer has the correct `ip` and `color`(see Section 3.2.4.) When the frame is returned, the SVC2 trap instruction is called with the frame pointer as an argument. The frame manager disposes of the frame.

4.3 Multiple frame sizes

To support multiple frame sizes, we have to be able to figure out the frame size of a frame being returned. There are two physical ways to do this — we can have the frame size passed to the frame manager as an argument, or we can write the frame size into a "hidden" location in the frame. We do the latter. Every frame is actually one bigger than its reported size. The frame size is written to the location right before the pointer to the frame. Thus, it is easy for the frame manager to figure out the size of any frame and the user does not notice any difference (except a slight decrease in the maximum number of frames.) Any other scheme also needs to store a size somewhere so the overhead must be incurred. Another advantage of this scheme is that a frame only has to be at least as large as the desired size. The frame manager has the flexibility of allocating a frame larger than necessary.

4.4 Dynamic allocation of frames

The earliest versions of the queue-based frame manager required the user to specify frame sizes and the number of each sized frame. During initialization, the frame memory would be partitioned into frames, writing the frame size into each frame as described in the previous section and filling the queues with pointers to those frames. Thus, the user would have to intelligently select both frame sizes and the number of each frame size.

A frame manager that would dynamically allocate frames was then implemented. The queues (or linked-lists) start out empty. When a request for a frame arrives, the appropriate queue or free-list is checked first. If a frame is available, it is allocated from the queue; otherwise, a frame is created from the free frame memory. Creating a new frame takes between 15 and 20 instructions, adding very little to the total running time of the problems we have run. Dynamically allocating frames gives us an additional bonus — by examining the quick-fit buckets at program termination time, we can learn the maximum number of frames that are required by a particular run of a program.

4.5 Single Frame Allocation

Our model of frame management has the frame manager called every time a frame is required. It is possible to allocate a single large frame and have the requesting procedure cut that frame for its children. Since each procedure will get and return a single frame, management costs could be reduced.

Though having the calling procedure cut frames up is desirable to minimize the number of frame requests, it makes the frame manager more complicated by increasing the range of desired sizes and throttles load balancing performance. The frame size requested from the frame managers might be unbounded since any number of children may be called by a procedure. Load balancing would degrade because the large frames would exist on a single processor. It is possible that a limited form of this scheme would increase performance because children would run on the same processor, in the same frame. Argument passing would be much cheaper. Future frame manager/compilers may experiment with this scheme.

4.6 Frame Manager Evolution

Many frame managers were written for Monsoon over a period of about a year. The genealogy of the frame managers is shown in Figure 4.1. The first frame manager, written by Hicks, had a single frame size, stored its frames on a linked list and only ran on a single processor. Young[32] improved on Hicks's algorithm by increasing the number of resources per processor to eight, one per interleave. Cycles spent waiting for the critical resources were reduced immensely. The scheme, however, did not scale well to an arbitrary number of different frame sizes and would return an out-of-frames error even when frames were available in other pipeline interleaves.

As mentioned in Section 2.5, all frame managers we have investigated use a quick-fit algorithm for managing the frames. Frame managers we implemented were divided

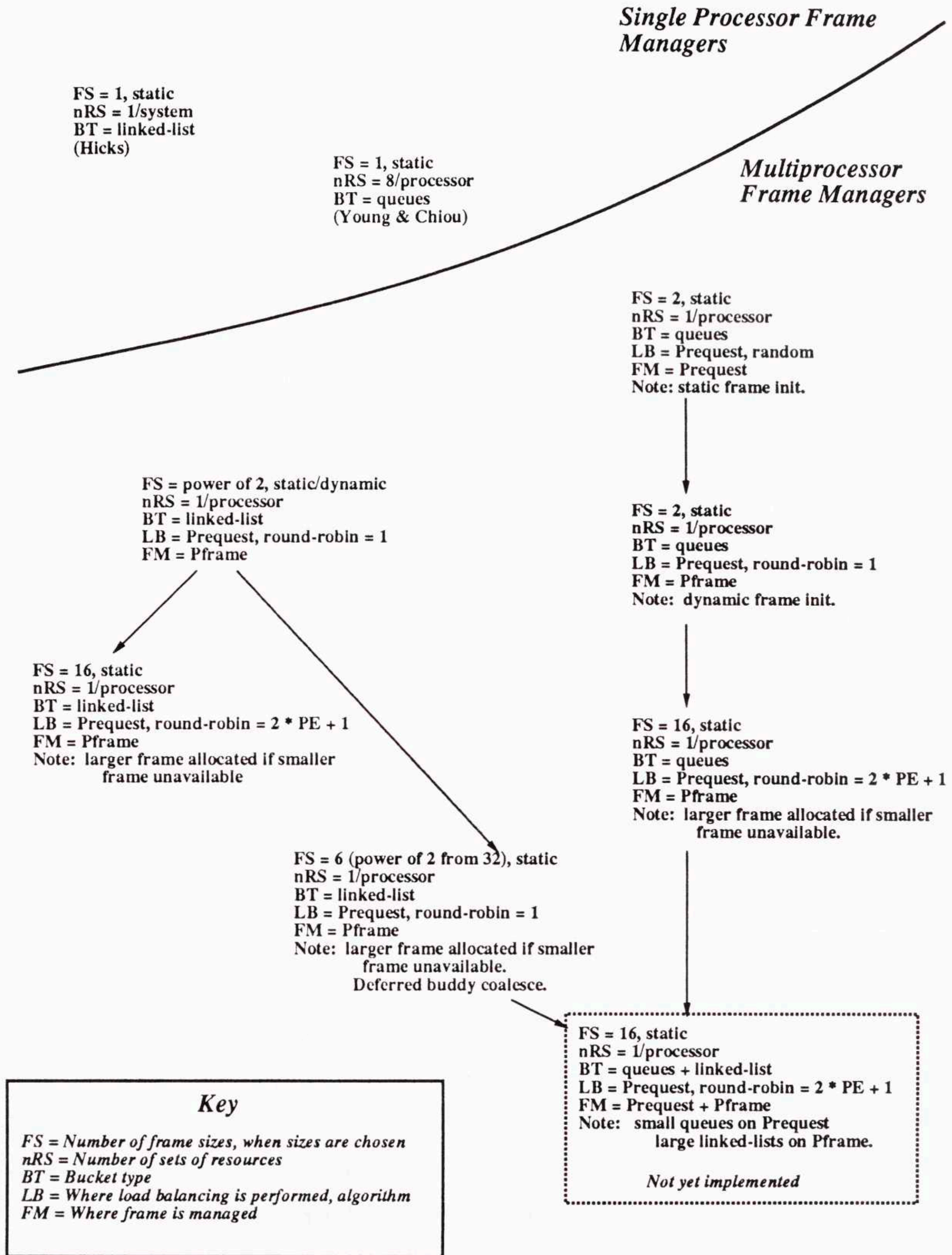

Figure 4.1: Frame manager genealogy

into ones based on queues that hold pointers to frames and ones based on linked-lists of frames. Hicks's frame manager, which was written to bootstrap the machine, was based on a single linked-list of frames. Since supervisor calls must be written in single-threaded code, a specific processor's linked-list can only contain frames that live on that processor. A linked-list with frames from different processors would require split-phase fetches that would not be single-threaded.

4.7 Prototype Frame Managers

At first, we felt that since a linked-list could not contain frames from a variety of processors, linked-lists would not be useful for multiprocessor frame managers. The first multiprocessor frame managers used queues that could contain pointers to frames on any processor.

Our first multiprocessor frame managers had $P^{*}_{request}$ load balancing and managing the frames. We wrote several versions of a frame manager that, when initialized, would build a set of queues on each processor containing frames from all processors. Each queue contained frames of a single size. When a frame was requested, the processor would take a frame off the head of the queue. When a frame was returned, its pointer was written to the end of the queue compiler convention has a frame returned by the same processor that allocated it — thus, there was no frame migration between processors.

After a queue had been run for some time, however, the frames in the queue would tend to clump with other frames from the same processor. This clumping would lead to poor load-balancing performance. We tried correcting this problem by having each processor contain only frames from other processors, but this solution did even worse when bounded-loops are used. Since a single processor will allocate all frames for a specific bound loop, having a processor allocate frames only on other processors would sometimes starve the allocating processor.

Another problem with predistribution is that the maximum number of frames a processor could request would be the number of frames that existed on itself[1]. Since a processor could only allocate frames from the queues that it held, and all processors had the same size queues, every processor's queues only contained the number of frames that existed within its own memory. Thus, the number of allocable frames was limited to the number present on a single processor.

4.7.1 Split-Phase Get-Frame

The problems with $P_{request}$ handling an entire frame request caused a revision of the frame allocation strategy. If each processor manages its own frames, there would be more load balancing flexibility and every processor could allocate every frame in the whole system. Implementing the idea, however, was tricky due to the supervisor calling convention that requires single-threaded code to hold the state. If we wanted to do a remote request for a frame, that request could only be one token large, since we had no memory to use for synchronization and the token would start a critical thread on P_{frame}. Allocating a frame requires a return continuation and a code-block descriptor. By replacing the frame pointer with the code-block-descriptor in the token tag, and carrying the return continuation in the value, we were able to squeeze both values into a single token, allowing remote frame requests.

The two network tokens required for a remote request (one there and the other back) are minimal, and the benefits of remote requests convinced us to move in that direction. Having P_{frame} manage all of its own frames allowed us to use linked-lists instead of queues to store frames.

4.7.2 Queue frame manager

We wrote several versions of a queue-based frame manager. Each of them has incrementally more functionality than the one before it. We will discuss the latest version.

[1] Of course, requests could be sent to other processors.

Performance measurements, written up in the next chapter will give numbers for all of the different frame managers and their performance tradeoffs.

Design Space Description

The latest version of a queue-based frame manager, rts_{queue}, has sixteen frame sizes, all multiples of sixty-four. There is one set of these resources per processor. Available frames are stored in queues that are allocated during initialization. The queues are large enough to guarantee that they will never overflow. Load balancing is done on $P_{request}$. Each frame size has its own round-robin counter which are incremented by $2 * pe_{number} + 1$. The frame management is performed on P_{frame}. If a smaller frame is unavailable and there is no more free frame memory to create new frames, a larger frame can be allocated to satisfy a frame request.

Functional Description

The queue sizes are computed by the initialization procedure – the queue size is equal to the total frame memory area divided by the frame size rounded up to the nearest power of 2. The queue sizes have to be a power of two since we want to use masking to simulate modulo arithmetic. The offset counter is masked by the queue size minus 1 and added to the base pointer. Thus the counter never needs to be reset to 0 and provides us with a count of the number of frames allocated.

Our early queue-based managers were the first developed. This frame manager, however, descends from our $rts_{inlined}$ frame manager (see Section 4.7.3.) The only real difference is that it uses queues instead of linked-lists to store available frames. Because queue access is faster than list access, the critical sections in rts_{queue} are shorter than the critical sections in $rts_{inlined}$. The total latency of rts_{queue}, however, is longer than that of $rts_{inlined}$.

Special instructions were written for rts_{queue} that allow us to lock a word of memory and add an integer to the contents of that memory location in a single instruction. The

new value can be put back into the memory location, unlocking the location, on the next instruction. Thus, the lock is only held for eight cycles. Unfortunately, a conditional instruction must be inserted into the critical section, extending it to sixteen cycles.

When a procedure running on processor $P_{request}$ wants a frame, the first part of rts_{queue} runs on $P_{request}$. A pointer to a data structure called a code-block descriptor defining the procedure to be called is passed to the frame manager as an argument. The frame manager reads the data structure to determine the desired frame size and selects the appropriate actual frame size. The round-robin counter for that frame size is incremented mod the number of processors to determine the processor P_{frame} where the frame will live. A message containing the return continuation and the CBD are sent to the second part of P_{frame}'s frame manager. There the appropriate queue is checked for an available frame, which is instantiated and returned if one is found. If a frame is not available, we attempt to cut one off the tail of frame memory. When cutting a new frame off the tail, rts_{queue} uses inlined blocked clears, clearing sixty-four locations with four instructions. If not enough free memory is available, the frame manager causes an error and halts the machine. All of the queues start empty — thus frames are allocated dynamically.

4.7.3 Linked-List Frame Managers

Linked-list frame managers store returned frames in linked-lists. Each linked-list contains frames of precisely the same size.

Design Space Description

We have two linked-list frame managers. The first one, $rts_{inlined}$, has sixteen different frame sizes, all multiples of sixty-four. There is one set of these frames per processor. Available frames are stored in linked lists. Load balancing is done on $P_{request}$ where there is one round-robin counter per frame size. The round-robin counters are incremented by

$2 * pe_{number} + 1$. Frame management is performed on P_{frame}. If a frame size is unavailable and there is no more free frame memory from which to allocate a new frame, a larger frame may be allocated.

The second frame manager, $rts_{coalesce}$, has six frame sizes, all a power of two starting at thirty-two. $rts_{coalesce}$ also uses one to increment its round-robin counters and only has *one* round-robin counter per processor. All of its other specifications are the same as $rts_{inlined}$. $rts_{coalesce}$, however, can coalesce frames if there is no other option.

Functional Description

Both use approximately the same algorithm for frame allocation. Both frame managers, like rts_{queue}, get frames by executing two distinctive parts of the frame manager. The first part of $rts_{coalesce}$ is essentially the same as the first part of rts_{queue}. $rts_{inlined}$, increments the round-robin counters (one counter per linked-list, which means one counter per frame size per processor) by the processor number $\times$ 2+1. This difference in the increment step seems to help load balancing to some degree since it is impossible for any two processors to choose P_{frame} with the same pattern.

When the second half the frame manager is entered, the correct quick-list is checked for a frame. If there are no frames on the correct quick-list, the algorithm attempts to allocate a frame from the tail of the frame memory. If there is not sufficient memory in the tail to allocate a frame of the desired size, the algorithm looks for a frame larger than the desired size. If a frame of a larger size is not found, behavior between the two frame managers differs. The first frame manager, $rts_{inlined}$, will return an error and halt the machine while the other frame manager, $rts_{coalesce}$, will attempt to coalesce the frames.

$rts_{inlined}$ was written after several months of using a preliminary version, rts_{list}. We found that frame sizes that are multiples of sixty-four worked best for a wide range of applications. By inlining all code for the frame manager, we were able to make it extremely fast. Because of its simplicity and the elimination of jumps by inlining, $rts_{inlined}$ is our fastest frame manager.

$rts_{coalesce}$ uses a deferred coalescing buddy-system. Because we use a stock buddy-system algorithm, the frame sizes are all powers of two, starting with thirty-two. When there are no more frames to be allocated, the system will coalesce frames in an attempt to get a frame of the desired size. Coalescing takes a long time, but can extend program runs and may allow higher loop bounds than a stock frame manager.

As with rts_{queue}, $rts_{inlined}$ and $rts_{coalesce}$ both use block clears of frames being allocated off the tail.

Chapter 5

Performance

In this chapter we will present Monsoon performance numbers and try to relate them to the frame manager. Of course, a large part of Monsoon performance is due to the compiler. The frame manager's performance becomes more evident during multiprocessor runs since load balancing becomes important.

The goal of a frame manager is to make the applications using it run quickly. The frame manager does that by balancing the load and managing the memory efficiently in both time and space. Generally the faster the frame manager can deliver a frame, the faster the application will run and the more efficiently the frame manager can manage its memory, the more parallelism the application can exploit.

This chapter will first discuss the methodology for collecting data and the data format that we will be using. Then we will explore the performance of the frame managers we have written in four application programs. The first section will describe the applications. The second section will look at load balancing performance. The third section will examine the efficiency of the memory usage.

Name	Size	Cycles	Frame	Ratio	Memory
Gamteb	40000	3233 M	8.5 M	379	232K
Matrix-Multiply	500	1058 M	98 M	10,800,000	17.5K
Paraffins	22	322 M	.00334 M	9650	99K
Simple	100 iter. 100x100	4682 M	12 M	390	139K

Table 5.1: Benchmark Summary

5.1 The Benchmarks

Our benchmarks are summarized in Table 5.1. The cycle counts, number of frame requests, and the amount of memory used were taken from one processor runs using $rts_{inlined}$. The first column states the name of the benchmark, the second column the parameters used to run the benchmark, the third column the total number of cycles used to run the benchmark, the fourth the total number of frame *requests*, the fifth column the ratio of the number of cycles to the number of frame requests, and the sixth column the total amount of frame memory used (which could be smaller than the number of frame requests times the frame sizes since frames are reused.)

This section was taken from Hicks, Ang, Chiou, and Arvind[13]. We studied five benchmarks: Fibonacci, Gamteb, Matrix-Multiply, Simple, and Paraffins. These benchmarks are described in this section. Members of CSG and CSG's collaborators have written other applications as well, although we will not discuss the performance of those applications in this paper. One of the largest Id applications currently being developed is a version of the Id compiler written in Id. The MCNP application is being written by people at Los Alamos.

All Id programs described in this document were written in Id90.1 with some annotations for storage deallocation[21]. These programs are available from Computation Structures Group upon request.

Our two large benchmark programs, Simple and Gamteb, were all originally written in Fortran. The Fortran code was written by experts and existed before the Id code.

Paraffins was originally written in Id. It was then ported to C by an expert. Matrix-Multiply is trivial and versions were written in Id, C, and Fortran.

5.1.1 Fibonacci

The Fibonacci numbers are defined recursively as follows: $F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} - F_{n-2}$ where $n \geq 2$. An Id program that generates the nth Fibonacci number is shown in Figure 1.1 code runs in exponential time. It is a great torture test for a frame manager since an exponential number of frames need to be allocated and the amount of work found within each code-block corresponding to a frame is very small. Thus, the amount of real application work to the amount of frame management is very low. We will use this program to show the effect of critical sections.

5.1.2 Matrix Multiply

This benchmark creates two matrices of size $n \times n$, multiplies them, and returns the sum of the elements of the product matrix as its result. The matrices all contain double-precision floating point numbers. Matrix-Multiply is written as a straightforward triply nested loop. In early versions of this benchmark, run during or before August 1991, the innermost loops of the matrix creation, multiplication and summation routines were all unfolded 10 times by the compiler to ameliorate the overhead of loop iteration. Compiler improvements since then reduced the overhead of loop iteration by a large amount. We scaled down unfolding to 4 times as further unfolding increases code size without giving very much improvement in run time.

This program is invoked by supplying a matrix size n and several loop bounds. The loop bounds control how much parallelism is exposed in the outer loops of the matrix creation, multiplication and summation routines.

This benchmark has been coded in C so as to compare the performance of Id with the performance of C. We have also written a 4×4 blocked Matrix-Multiply in both Id and

C. This blocked version simultaneously computes the value of 16 elements in the final matrix that forms a 4×4 block. The net effect is to reduce the number of fetches that we perform on the two source matrices. Thus, each time through the innermost loop, we fetch 8 matrix elements, 4 from each source matrix, and use them to update 16 elements of the final matrix that we are computing. This reduces the number of fetches by a factor of 4. Although the numbers reported here for Id are obtained by changes to the source code, work is underway to have the compiler perform this transformation automatically as an optimization.

Our Matrix-Multiply runs are of size 500 by 500. The Id code implementation is about 130 lines, including comments.

5.1.3 Gamteb

Gamteb[5] statistically simulates the trajectory of particles (photons) through a carbon rod that is partitioned into cells. Each particle is statistically *weighted* to emphasize particles that are in rightmost cells. Gamteb was written by researchers from Los Alamos National Laboratories and is a standard supercomputer benchmark derived from a real application, MCNP. MCNP principally run on Crays but is difficult to vectorize.

We have two versions of Gamteb, corresponding to two different rod geometries. In the first version, gamteb-2c, which corresponds to the Fortran benchmark code, the rod is divided into 2 cells with 4 surfaces. In the second version, gamteb-9c, the rod is divided into 9 cells and 11 surfaces. The second version is much more computationally intensive than the first, because each particle is split many more times. We use Gamteb-9c in this thesis.

The simulation considers n particles independently, where typical problem sizes are 40 thousand to several million particles. Particles all enter the simulation through the front surface, and may exit the simulation in one of 4 ways: *escaping* through the cylindrical surface, *back scattering* through the front surface, *transmitting* through the back surface,

or *dying* due to lack of statistical significance. The result is three histograms of the energies of the particles that exited, and counts of particles that underwent various processes.

This program is intensive in storage. It operates on particles and counts functionally, so whenever a new particle or count of events is needed, a new nine-tuple is allocated. This code has been hand annotated with some storage reclamation pragmas so that the compiler will insert calls to deallocate storage.

Gamteb runs are of forty thousand particles, which is a small but standard benchmark size. The Id code implementing Gamteb is about 750 lines, including comments. Real program runs would be in the millions or tens of millions of particles.

5.1.4 Simple

This application is a hydrodynamics and heat conduction simulation programs known as the Simple code[8]. The Simple document, along with the associated Fortran program, was developed as a benchmark (unclassified) to evaluate various high performance machines and compilers. Though Simple is supposed to reflect some "real applications", it is contrived to reflect a more complex mix of numerical methods than the usual problems in that class.

Simple uses a Lagrangian formulation of equations to simulate the behavior of a fluid in a sphere. To simplify the problem, only a semicircular cross-sectional area is considered for simulation. The area is divided into parcels by neighboring radial and axial lines. Each parcel is called a *zone*. The intersection of radial and axial lines is called a *node*. In the Lagrangian formulation, the nodes are mapped onto a 2-dimensional logical grid where grid points have coordinates (k, l) for $k_{min} \leq k \leq k_{max}, l_{min} \leq l \leq l_{max}$. The product, $k_{max}l_{max}$, is the *grid size* of the problem. A parameter of *ghost zones* is added around the rectangular grid to incorporate the appropriate boundary conditions. For each time step, the simulation computes the velocity and position of each node, and area, volume,

density, pressure, artificial viscosity, energy and temperature of each zone based on the values of these quantities in the previous time step. Some additional calculations are performed to compute the size of the time step to be taken and to check the energy balance.

The simulation is performed a specified number of cycles. Simple runs reported here are of 100 cycles with a grid size of 100 by 100. The Id code implementing Simple is about 1000 lines, including comments.

5.1.5 Paraffins

The Paraffins benchmark[2] enumerates all of the distinct isomers of each paraffin of size up to n. Paraffins are molecules with chemical formula C_nH_{2n+2}, where C and H stand for carbon and hydrogen atoms, respectively, and $n > 0$.

Paraffins is an example of a non-numeric program. The algorithm is $O(e^n)$. The program generates lists of paraffins and finally returns an array filled with the number of distinct paraffins of each size up to and including the maximum size specified by the user.

Paraffins runs are of size 22, meaning paraffins up to and including those of size 22 are generated. The Id code implementing Paraffins is about 300 lines, including comments.

5.2 Run-time Parameters

It is important to understand some of the decisions we made when running the programs. As explained in Section 1.3, loops must be bounded in order to balance exposed parallelism and memory usage. We explain our method for selecting loop bounds in this section. We also justify the use of the same frame managers for single and multiprocessor runs.

Loop bounds must be either input by the user or computed by user-code during run-time. Since there is currently no general algorithm for choosing good loop bounds, we iterated over a fixed set of loop bounds and chose the ones that performed best. Iterating over loop bounds is non-optimal and is very tedious. Performance can be strongly tied strongly to the choice of loop bounds, and the choice of which loops to execute in parallel. Further work to automate bounded loops through some sort of compiler and/or RTS mechanism is needed.

We use the same run-time system for one processor and for multiple processors. If we knew that the run-time system would be running only on a single processor, it could be made more efficient, since the load balancing could be removed. Since roughly 50% of frame management is spent balancing the load, frame management times could be reduced by at most 50%. Though reducing frame management overhead by one half may seem like a lot, the vast majority of programs spend less than 15% of their time in the frame manager. In general, optimizing a frame manager for a single processor will save at most 7.5% of the total cycles, and will often save less than that.

Id is inherently a parallel language and is compiled to be executed in parallel. Only one object file needs to be produced for each program, and we can use the same, unaltered object code for any number of processors.

5.3 Data Format

The data format we use to present our results is a set of bargraphs that show a summed total of the instructions executed on a specific configuration of Monsoon. For each application program, there are two sets of bargraphs. One shows the statistics data by "colors" (seen as grey scales on our black-and-white graphs), where each color corresponds to some part of the code. An example of such a graph is in Figure 5.3. We divide all of our data into six colors: frame management, application code, heap allocation, heap deallocation, second part of split-phase instructions, recirculations, and idles. Recirculations are simply spinning wait instructions used by the frame manager.

The second set of bargraphs shows the statistics data by operation categories. An example of this type of graph is in Figure 5.4. The different categories we show in this thesis are floating-point operations, integer operations, fetches, stores, switches, tags, identities, bubbles, second part of split-phase transactions and idles. The two categories that might need explanation are tags and switches. Tags are operations that build and send tokens. For instance, the instruction that sends an argument to a new procedure is a tag operation. The switch category contains switches, which are essentially gates that allow a value to pass depending on another value, and also contains miscellaneous operations such as conversion instructions and traps.

Bargraphs for one processor runs are self explanatory. Bargraphs for multiprocessor runs are the sum of statistics from all of the processors that participated in that run. Runs using different run-time systems are all displayed on the same graph. Runs using the same run-time system are grouped together. An example is shown in Figure 5.3. The first set of four show data from runs of Gamteb on 1, 2, 4, and 8 processors running with rts_{queue}. The second set of four were run with $rts_{inlined}$ and the third set of four were run with $rts_{coalesce}$.

5.4 Frame Manager Time

Frame manager cycle counts and frame manager critical sections are shown in Table 5.2 and Table 5.3. $rts_{inlined}$ takes the fewest number of total cycles. rts_{queue} has the shortest critical sections. Most of the time shorter latencies, even with longer critical sections, allow the application to run faster.

We ran the doubly recursive Fibonacci on Monsoon with our three frame managers to see how well the frame managers performed under heavy loading. Since the recursive version of Fibonacci performs so many procedure calls, and the work done in each procedure is so little, the ratio of frame allocations to other work is very high.

The results of Fibonacci 17 are shown in Table 5.4. All relevant statistics are listed.

Frame Access	rts_{queue}	$rts_{inlined}$	$rts_{coalesce}$
Get-frame, found	41	37	51
Get-frame, create	46	43	67
Return-frame	21	19	19
Critical Get-frame, found	12	8	23
Critical Get-frame, create	20	18	33
Critical Return-frame	20	18	18

Table 5.2: Frame manager cycle times

Frame Access	rts_{queue}	$rts_{inlined}$	$rts_{coalesce}$
Get-frame, found	8+16	8+24	8+48
Get-frame, create	8+24+8	8+16+8	8+32+8
Return-frame	8	48	48
Critical Get-frame, found	16	24	48
Critical Get-frame, create	24+8	16+8	32+8
Critical Return-frame	8	48	48

Table 5.3: Frame manager critical sections (in cycles)

The first row, total cycles, shows the total number of cycles executed to complete the problem. The second row, *fib* cycles, shows how much time was spent executing user-written Fibonacci code. The third row shows the number of instructions executed by the frame manager. The fourth and fifth rows, recirculate and second-phase, indicate overhead associated with frame management. Recirculate indicates the number of cycles spent spin-waiting for a resource, most likely the head of a list. Second-phase operations are the second halves fetches and stores performed on processor local memory.

Our fastest frame managers, rts_{queue} and $rts_{inlined}$, take about 364,000 cycles and 387,000 cycles respectively while the actual Fibonacci code takes about 229,000 cycles. For each call to Fibonacci, however, a call is made to the frame manager. Thus, our frame managers take about the same order of time as the body of a compiled Fibonacci.

Though rts_{queue} performs better than $rts_{inlined}$ on Fibonacci, it generally does not do as well. rts_{queue} has a very short critical section. Because Fibonacci always requests

	rts_{queue}	$rts_{inlined}$	$rts_{coalesce}$
total cycles	596	619	737
fib cycles	229	229	229
frame management	307	258	328
recirculate	3	60	74
second-phase	54	69	103

Table 5.4: Fibonacci Statistics in Thousands of Cycles

frames of the same size, the same quick-fit bucket is always used. Critical section length becomes very important in this case, as can be seen in Figure 5.2. Although $rts_{inlined}$ is significantly faster than rts_{queue} in terms of frame management time, $rts_{inlined}$ spends a lot more time spin waiting (recirculating) for resources. Thus, for Fibonacci, rts_{queue} performs best.

Fibonacci is a worst case program. Most real programs will not tax the frame manager nearly as much. Though we always want more performance, we judged the performance of rts_{queue} and $rts_{inlined}$ to be acceptable.

5.5 Efficiency of Memory Usage

One might consider frame management space to be two distinct parts: the space efficiency of the frame manager itself (how much memory it takes) and how well it manages the frame memory. rts_{queue} takes $O(n)$ space where the constant is generally very small (around .01 or less) while $rts_{inlined}$ and $rts_{coalesce}$ take $O(1)$ space. The memory used by the frame manager, therefore, is negligible.

How well the frame manager manages the memory for the user, however, is sometimes difficult to measure. We can think of two ways to measure memory usage efficiency. The first is comparing maximum loop bounds — one would think the larger the loop bounds, the greater the amount of allocated frame memory. This is not always true, since a slower frame manager running a program with a certain loop bound might use

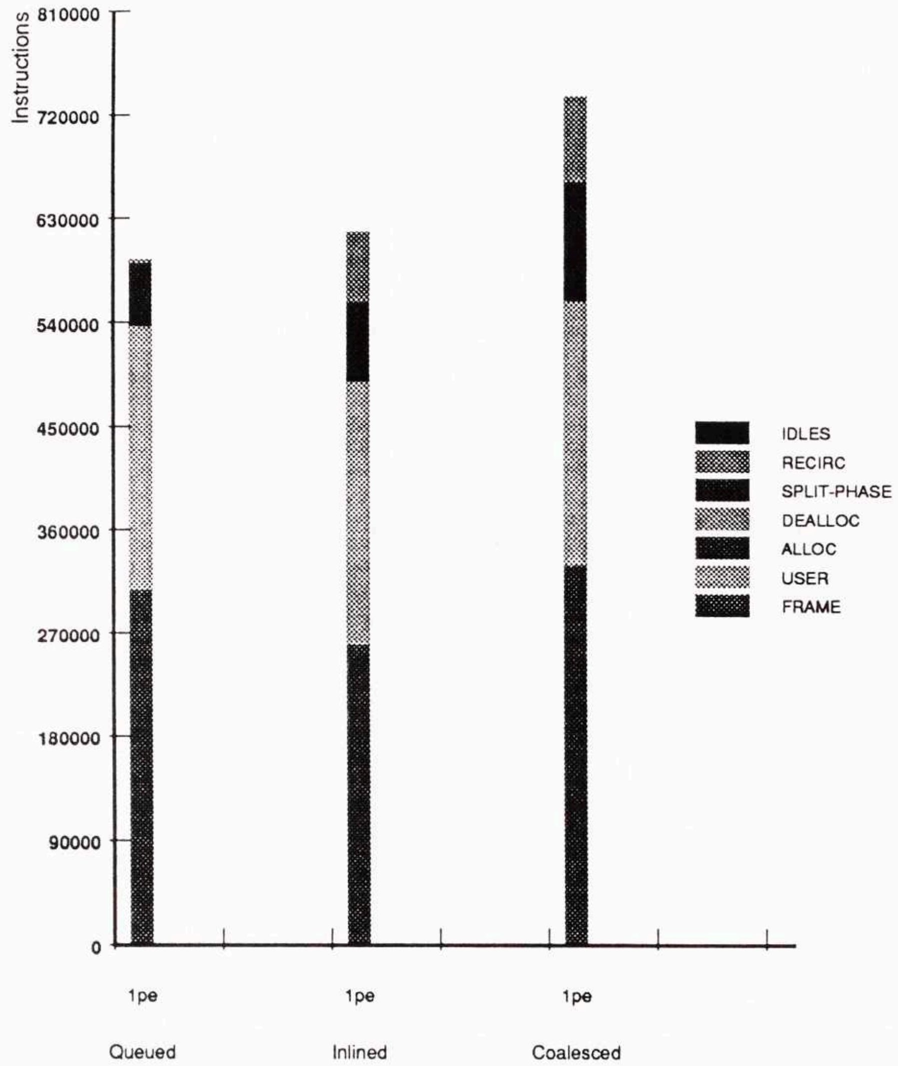

Figure 5.1: FIBONACCI colors, n = 17

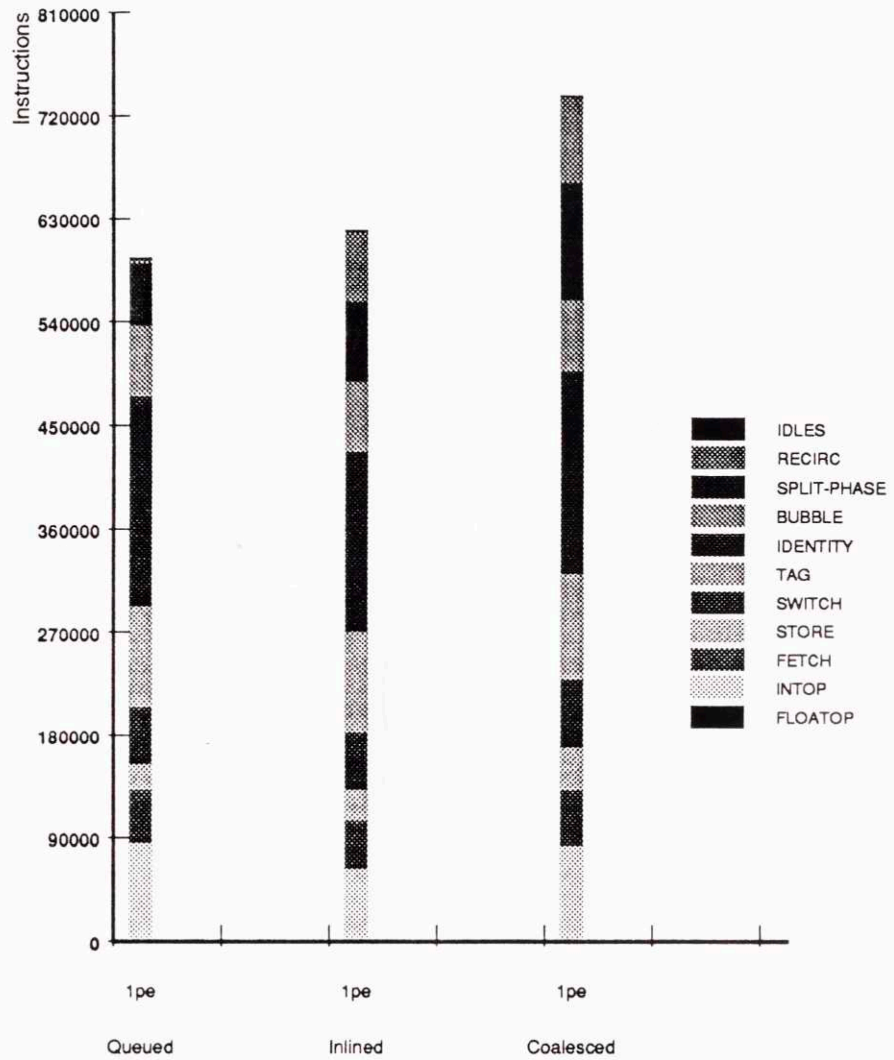

Figure 5.2: FIBONACCI opmix, n = 17

less memory than a faster frame manager running the same program with the same loop bounds. This is because the faster frame manager will deliver the bounded loop frames faster, thus expanding the parallelism faster. Since the amount of parallelism for a fast frame manager might be higher than for a slow frame manager, the amount of active frames could be higher for the faster frame manager. The reverse could also happen. The slower run-time system could have more frames outstanding than the fast frame manager, causing a lower maximum loop bounds. If the total run-time is different between the two frame managers being compared, it becomes difficult to use maximum loop bounds as a measure of frame management efficiency.

Another way to measure frame manager memory efficiency is to look at the amount of frame memory tail that has been allocated. For the same reasons as maximum loop bounds, this measurement is flawed if the total run-time is different between runs with different run-time systems.

We believe that frame memory usage efficiency is *not* an important part of frame management characteristics since it only *indirectly affects application speed.* As stated before, the only real measure of a frame manager is how fast it allows the applications to run.

5.6 Load Balancing Results

It is difficult to measure load balancing performance since it is nearly impossible to distinguish poor load balancing from a lack of parallelism using Monsoon[1]. We assert that if a program has close to linear speedup, the load is balanced — if the load is not balanced, linear speedup is impossible. Table 5.6, Table 5.5, Table 5.7 list speedups and cycle-times with different run-time systems for each application program. Gamteb, Matrix-Multiply and Paraffins speed up well. Simple could do better, but loops to bound were chosen poorly so we have not yet been able to run Simple at its full potential.

[1] Of course, our simulator could give us the numbers we need. The simulator, however, is at least 10,000 times slower than our hardware and we could not run any real examples on it.

Configuration / Program	1pe		2pe		4pe		8pe	
	$\frac{1pe}{1pe}$	$\times 10^6$ cycles	$\frac{1pe}{2pe}$	$\times 10^6$ cycles	$\frac{1pe}{4pe}$	$\times 10^6$ cycles	$\frac{1pe}{8pe}$	$\times 10^6$ cycles
Matrix-Multiply 500x500	1	1057	1.99	531	3.90	271	7.62	139
Gamteb-9 40000 particles	1	3307	1.93	1716	3.78	875	7.55	437
Simple 100 iters, 100x100	1	4753	1.86	2557	3.47	1369	6.30	754
Paraffins n = 22	1	322	1.99	162	3.93	82.0	7.26	44.4

Table 5.5: Speedup for rts_{queue}

Configuration / Program	1pe		2pe		4pe		8pe	
	$\frac{1pe}{1pe}$	$\times 10^6$ cycles	$\frac{1pe}{2pe}$	$\times 10^6$ cycles	$\frac{1pe}{4pe}$	$\times 10^6$ cycles	$\frac{1pe}{8pe}$	$\times 10^6$ cycles
Matrix-Multiply 500x500	1	1057	1.99	531	3.90	271	7.62	139
Gamteb-9 40000 particles	1	3233	1.93	1678	3.79	853	7.56	427
Simple 100 iters, 100x100	1	4994	1.87	2670	3.48	1435	6.21	804
Paraffins n = 22	1	322	1.99	162	3.92	82.2	7.25	44.4

Table 5.6: Speedup for $rts_{inlined}$

Configuration / Program	1pe		2pe		4pe		8pe	
	$\frac{1pe}{1pe}$	$\times 10^6$ cycles	$\frac{1pe}{2pe}$	$\times 10^6$ cycles	$\frac{1pe}{4pe}$	$\times 10^6$ cycles	$\frac{1pe}{8pe}$	$\times 10^6$ cycles
Matrix-Multiply 500x500	1	1057	1.99	531	3.90	271	7.62	142
Gamteb-9 40000 particles	1	3418	1.93	1773	3.78	905	7.53	454
Paraffins n = 22	1	323	1.99	162	3.91	82.5	7.29	44.2

Table 5.7: Speedup for $rts_{coalesce}$

5.7 Accounting for the Idles

There are four sources of all the idles on Monsoon. These sources are as follows.

- Lack of parallelism
- Load imbalance
- Startup and termination latencies
- Hardware hazard

Generally, there will be idle cycles from all four sources in every program that runs on Monsoon. We can often reduce the number of idle cycles by adjusting loop bounds and by using clever compile-time optimizations and run-time system strategies.

5.7.1 Lack of Parallelism

If an algorithm does not have enough parallelism to keep a specific machine configuration busy, the machine will idle. A lack of parallelism could be caused by the compiler, poorly chosen loop bounds, the size of the problem being run, or the algorithm itself. Loop bounds are easy to change. Sometimes, however, the only fix is to rewrite the program with new a algorithm that it has more inherent parallelism.

5.7.2 Load Imbalance

If the load is not balanced between all of the processors in a specific configuration, the processors with less work will idle. Load imbalance simply means that one or more processors are idle because they have no work, while the other processors have more than enough work to keep them busy. Note that load imbalance cannot occur on single processor configurations. If only a few processors are busy, but there is not enough work

to keep all of the processors busy, there is a lack of parallelism rather than a misbalanced load. For Monsoon, balancing the load is the responsibility of the frame manager.

From the run time statistics, it is difficult to differentiate idles caused by a lack of parallelism and the idles caused by load imbalance. One must be able to tell at any instance if some processors are working while others are idled *and* that there is enough work to keep all processors busy. On Monsoon, it is impossible to get all of this information simultaneously. MINT, our simulator, could be modified to do so, but the size of problem runable on MINT are very small and mostly unrealistic.

5.7.3 Startup and termination latencies

The execution of Id on Monsoon starts with the invocation of the top level procedure on one processor and spreads out to other processors. The computation ends in a similar way, but in reverse order on the same processor that computation started on. Startup and termination costs are incurred whenever the architecture and/or the run-time system limit the expansion or contraction of work to/from the processors. Startup and ending latencies essentially cause an artificial lack of parallelism.

In order to call a procedure, a frame needs to be allocated and arguments need to be sent to the new invocation. Frame allocations and network sends take time — waiting for these events to occur delays the parallelism that will be available in the called procedure.

Startup and end costs are often constant per processor, regardless of how many processors there are in the system. When a single processor shows 1% idles due to startup and end costs, an n processor system will generally show at least n% accumulated idles due to startup and end costs. Multiprocessor systems cannot startup or end any faster than a single processor system and thus every processor in the system must pay at least the same startup and end cost as a single processor. The more processors involved, the longer it takes to distribute the initial work to fill up the machine (startup) and to synchronize the final termination.

5.7.4 Hardware Hazard

We have already described the hardware hazard in Section 3.2.3. Most fetches and stores in our compiled Id code will excite the hardware hazard, causing an idle. Instructions that send arguments to other procedures, when the callee procedure is on another processor, may also excite the hardware hazard. Sending arguments on a single processor will never cause a hazard — the hazard is excited only when arguments are sent to other processors. An argument send instruction is very similar to a store instruction.

5.7.5 Idles in Multiprocessors

Overall, multiprocessor systems are guaranteed to have more idles than single processor systems. A lack of parallelism due to the algorithm, problem size, or loop bounds will show up much more on a multiprocessor since more parallelism is required to keep it busy. A single processor cannot have a load imbalance, but a multiprocessor configuration can have idle processors even though there is plenty of parallelism to exploit. Since all programs start and finish from a single processor, startup and termination costs are at best constant per processor in the system. Finally, multiprocessor configurations require more tokens to and from the network and so there are more idles due to Monsoon's hardware hazard.

Startup and termination latencies can be reduced by improving the run-time system. The latency of frame allocation, in the absence of other data dependences, limits the rate at which parallel activities can be spawned. The latency of heap allocation also often adds to the idle time at program startup. Likewise, the latencies of frame and heap deallocation often contribute to the idle time at program termination. Reducing these latencies will reduce the overhead of program start and termination.

5.7.6 Categorizing idles

It is difficult to sort the idles into our four categories. It is especially difficult, however, on the hardware to differentiate idles caused by load imbalance as opposed to idles caused by lack of parallelism. Startup and end costs are visible using our performance visualization tools as a ramp up and a ramp down. The number of hardware hazard idles is determined by counting the number of instructions that would cause them.

One heuristical way to find the source of idles is to watch the LED's found on all Monsoon processors. These LED's indicate when a token enters the pipeline. When there is a lack of parallelism across the system, all of the lights dim simultaneously. When there is a load imbalance, some lights are brighter than others. Startup and termination costs, special cases of lack of parallelism, show up as a dimming of the lights during the startup and termination of a program run. Unfortunately, watching the lights is not very precise, but it does give us hints as to where to look. Though we have visualization tools, they do not work well on multiprocessor configurations since it is difficult to start and stop all of the processors exactly at the same time. The inherent skew warps the statistics to a point where they are often useless.

One way to examine the source of idles is to compare simulation runs with real hardware runs. Our simulator, MINT, does not model network latencies, hardware hazards, and can be made to execute run-time system trap instructions in a single cycle. Thus, a run on MINT can be made very idealized and can give a lower limit on the number of idles. The only problem is that MINT runs much slower than Monsoon and thus cannot perform full runs with large data sets. MINT statistics, however, can help us understand where idles come from.

For each of our benchmarks, we will try to explain the idles found in all runs. Although we cannot explain idles extremely precisely since many of them are due to dynamic conditions in the hardware, we will try our best.

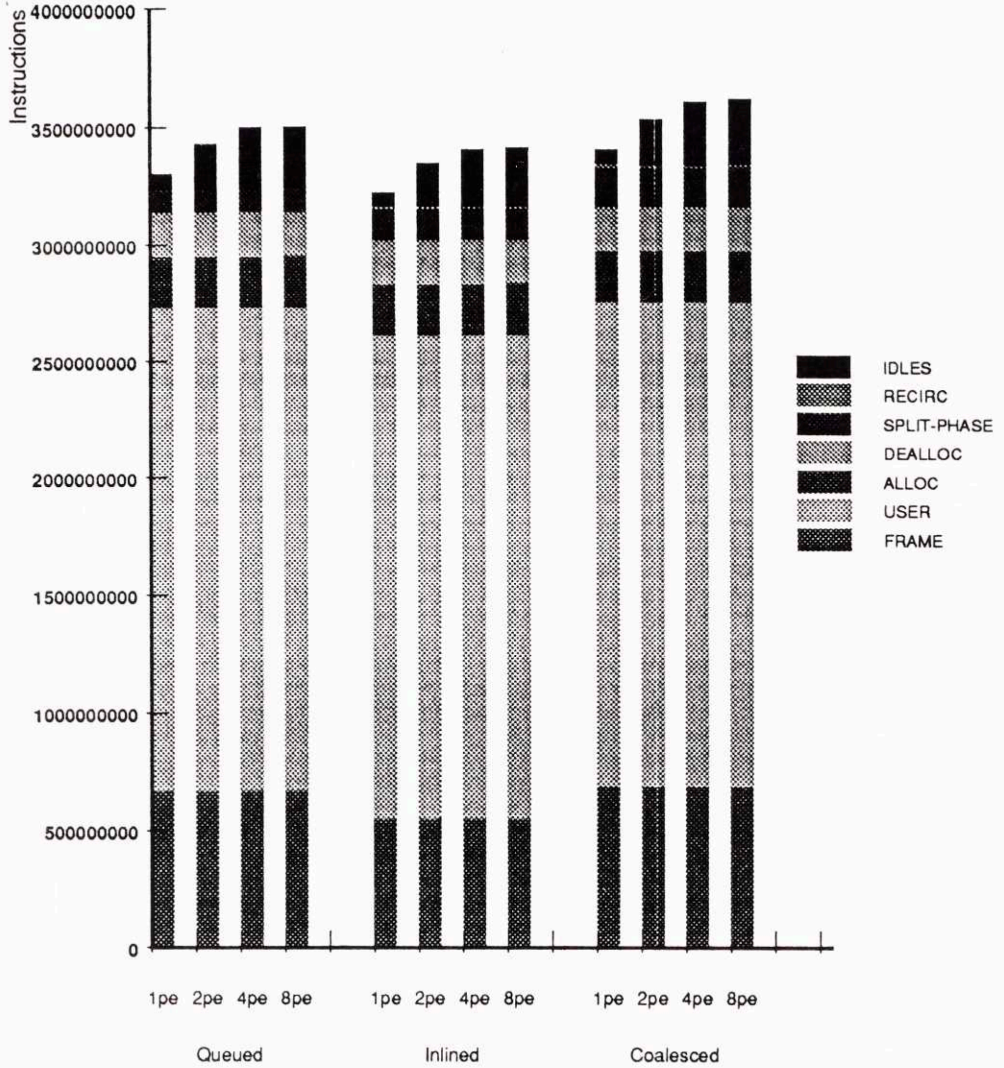

Figure 5.3: GAMTEB-9 colors, 40000 particles

5.8 Gamteb

The program Gamteb was described in Section 5.1.3. We understand Gamteb and its run-time behavior very well and can explain virtually every idle from the program runs. In Figure 5.4 in the $rts_{inlined}$ and $rts_{coalesce}$ bargraphs, we noticed that the number of idles seemed to exponentially approach some maximum. We knew that the idles in a single processor case were caused by a hardware hazard that prohibits a token going to the network while a token is taken from the token queue. This hazard produces an

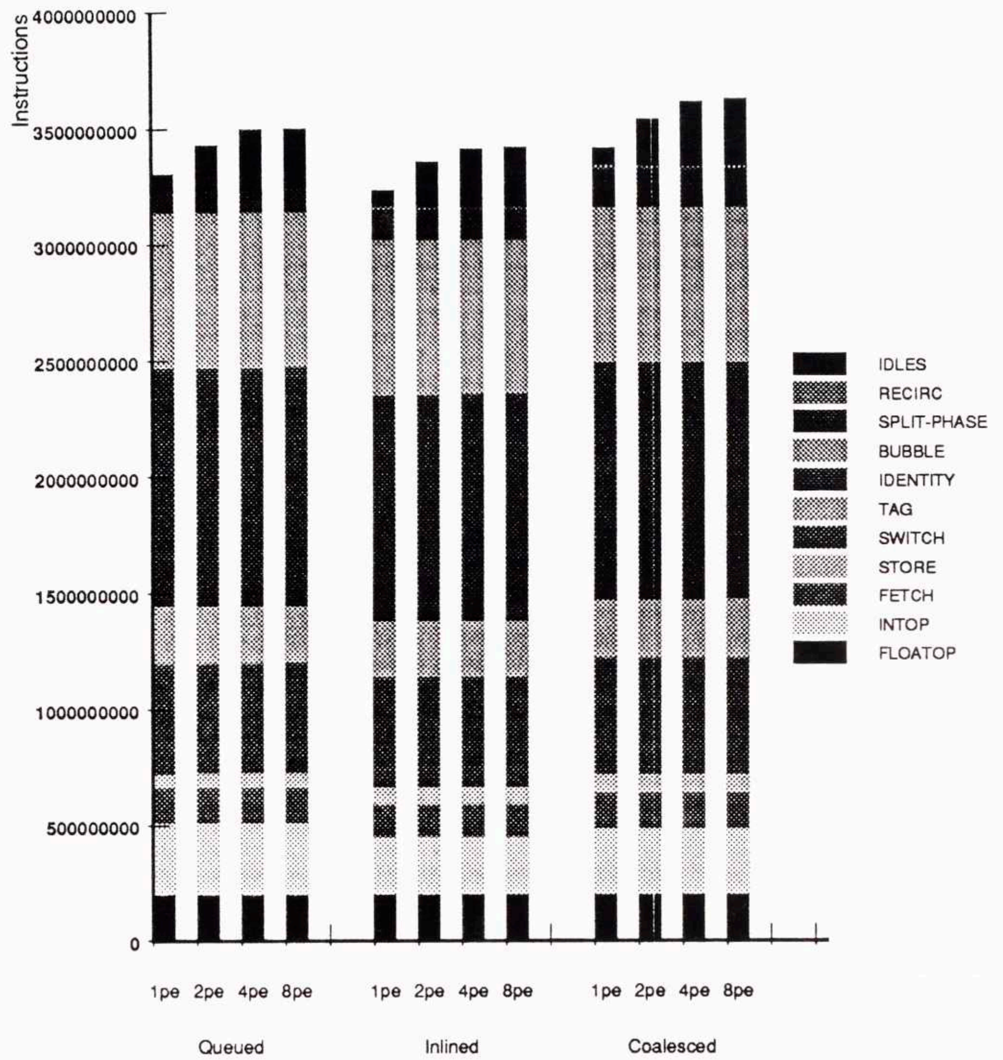

Figure 5.4: GAMTEB-9 opmix, 40000 particles

idle when a fetch without a recirculating token is executed or about 50% of the time that a network token enters the processor. Thus, the idles in a single processor run were unavoidable. We did not understand, however, where the the extra idles in the two, four, and eight processor runs were coming from.

After some puzzlement, we noticed that tag operations, when going to another processor, act like a fetch in that they could cause an idle. Since all tag operations on a single processor go to the same processor, no idles were caused by them. Once we go to n processors, approximately $(n-1)/n$ of the tag operations go off the processor, causing a hazard idle. From Figure 5.4 we can see that the number of idles for two processors is approximately the number of idles from one processor plus half the tag operations. For four processors, the number of idles is about the number of idles from one processor plus three-quarters the number of tag operations. The same progression goes for eight processors as well. With this observation, we were able to explain all of the idles produced during Gamteb runs with $rts_{inlined}$ and $rts_{coalesce}$.

5.8.1 Frame Manager Analysis

All the frame managers performed as expected. Speedups are perfect, after taking the hardware hazard into account. The run-times differed because of different latencies in the frame managers.

5.9 Matrix-Multiply

Most of the idles found in Matrix-Multiply are explainable. The opcode mix graph for Matrix-Multiply is found in Figure 5.6. Looking at the one processor bar, one can see that the fetches take up about 6% of the total cycles and the idle count is around 9.4% of the total cycles. The fetches used in the Matrix-Multiply inner loop excite a hardware hazard (see Section 3) which forces an idle for every fetch. Also tokens returning from

the network into a processor can cause an idle about 50% of the time if the pipeline is full. This idle is caused by the same hardware hazard. Thus it is possible to account for between 8% and 9% of the idles found in a one processor run. The extra 1% or 2% of the idles can be explained by startup and ending costs as well as some of the frame and heap management latencies.

For multiprocessors, accumulated idles increase very slowly. We believe that these idles are due in part to multiplied startup and end costs but mostly to the hardware hazard. Currently, the run-time system allocates the initial two matrices on the same processor and this can be a bottleneck. Even if this problem was fixed, however, only one processor can allocate any specific heap object. Thus, since only three matrices are created at most three processors can work during their allocation. There is a lack of parallelism until the three matrices are allocated. It is possible to distribute part (clearing presence bits) of the allocation procedure, but we have not yet done so.

5.9.1 Frame Manager Analysis

Matrix-Multiply was run with an assembly code heap manager. Heap deallocation is not supported, making heap allocation a lot faster. Specializing the heap manager is a common technique used by C and Fortran programmers (actually, Fortran programmers often statically allocate heap area), and thus is not unreasonable.

Our Matrix-Multiply numbers show linear speedup for rts_{queue} and $rts_{inlined}$ after taking the hardware hazard into account. $rts_{coalesce}$ is probably a little too slow to get great performance for eight processors. Most likely, $rts_{coalesce}$ would do much better on a longer run. The run times were less than fifteen seconds for eight processors. Such short runs magnify the startup and end costs dramatically.

Matrix-Multiply does very few frame allocations and even fewer heap allocations. The structure of our Matrix-Multiply with the two outermost loops bounded and the innermost loop sequential will tend to allocate frames in the beginning of the run. Thus,

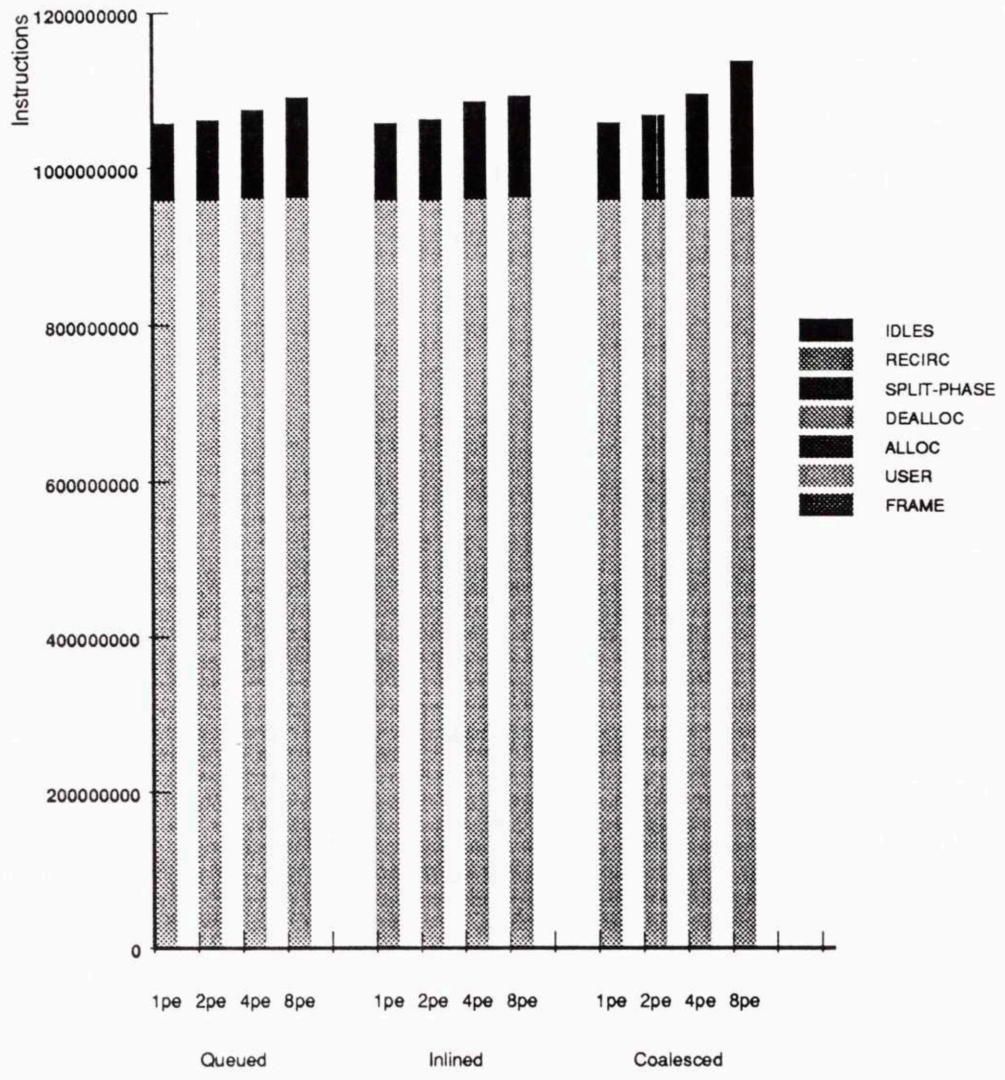

Figure 5.5: Matrix-Multiply colors, 500x500

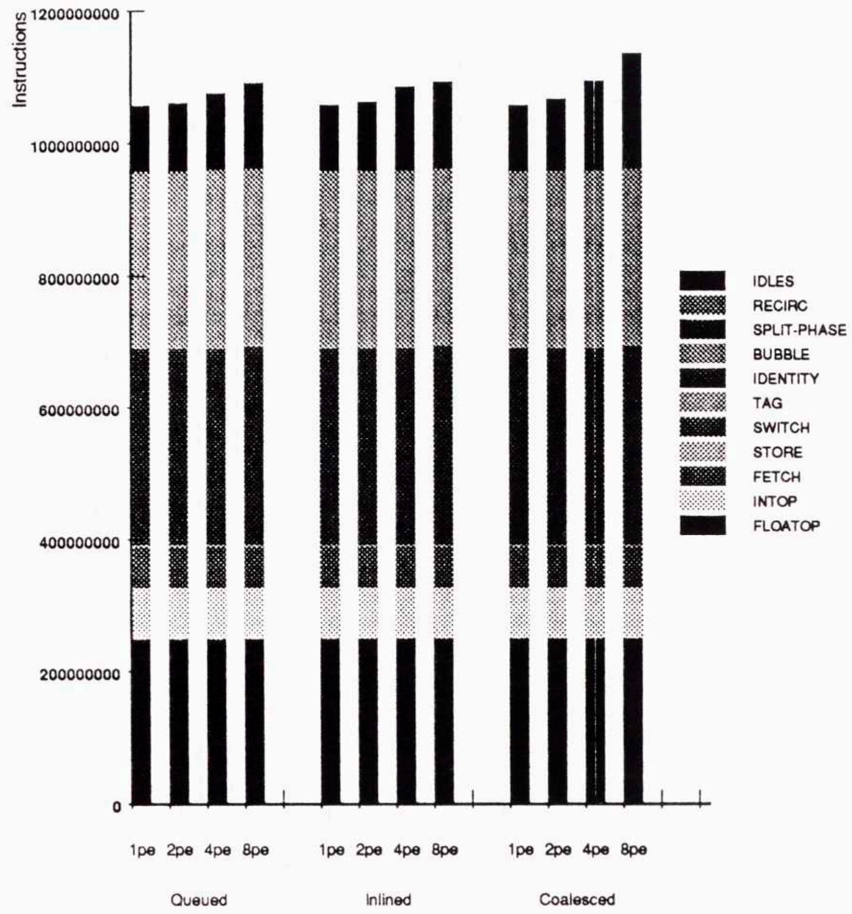

Figure 5.6: Matrix-Multiply opmix, 500x500

load balancing is relatively static. We believe that in these kinds of programs, a simple round-robin scheme may introduce less than optimal load balancing. Watching the work-load lights on the Monsoon hardware supports this theory. We can actually see poor load balancing. Using a single round-robin counter for each frame size on each processor can prevent perfect load balancing. For example, let us say that two loops are being unfolded simultaneously on the same processor. One loop simply traverses a list to find its length and thus does very little work inside each frame. The other loop, on the other hand, does a lot of work inside each frame. If the frame sizes used by both loops are the same, it is possible for all of the little-work frames to be allocated on one processor and all of the big-work frames allocated on the other processor. Load balancing will be poor in this case.

Another scheme would have each code-block have a round-robin counter. Although this scheme will prevent the problem posed in the previous chapter, it is still possible to get uneven load balancing from recursive calls to a procedure. In a two-processor example, if the round-robin counters for a specific procedure become synchronized across all processors, it is possible for each processor to execute an instance of that procedure and its recursive child. In the unfortunate circumstance that all active procedures' counters are synchronized, it is possible that only one processor is active at any one time. This possibility is very low. We should, however, try to implement this scheme sometime in the future. It will require changes outside the frame manager (adding counters to all code-block descriptors) as well and thus is difficult to do.

5.10 Paraffins

When we first started running Paraffins in parallel, we did not achieve the speedups that we expected. After modifying heap management extensively, Paraffins ran much better but still did not perform as we expected. Since Paraffins' complexity is exponential in space, we ran out of memory very quickly on a one processor one I-structure board configuration. The runs we were making were taking about .4 seconds on eight processors

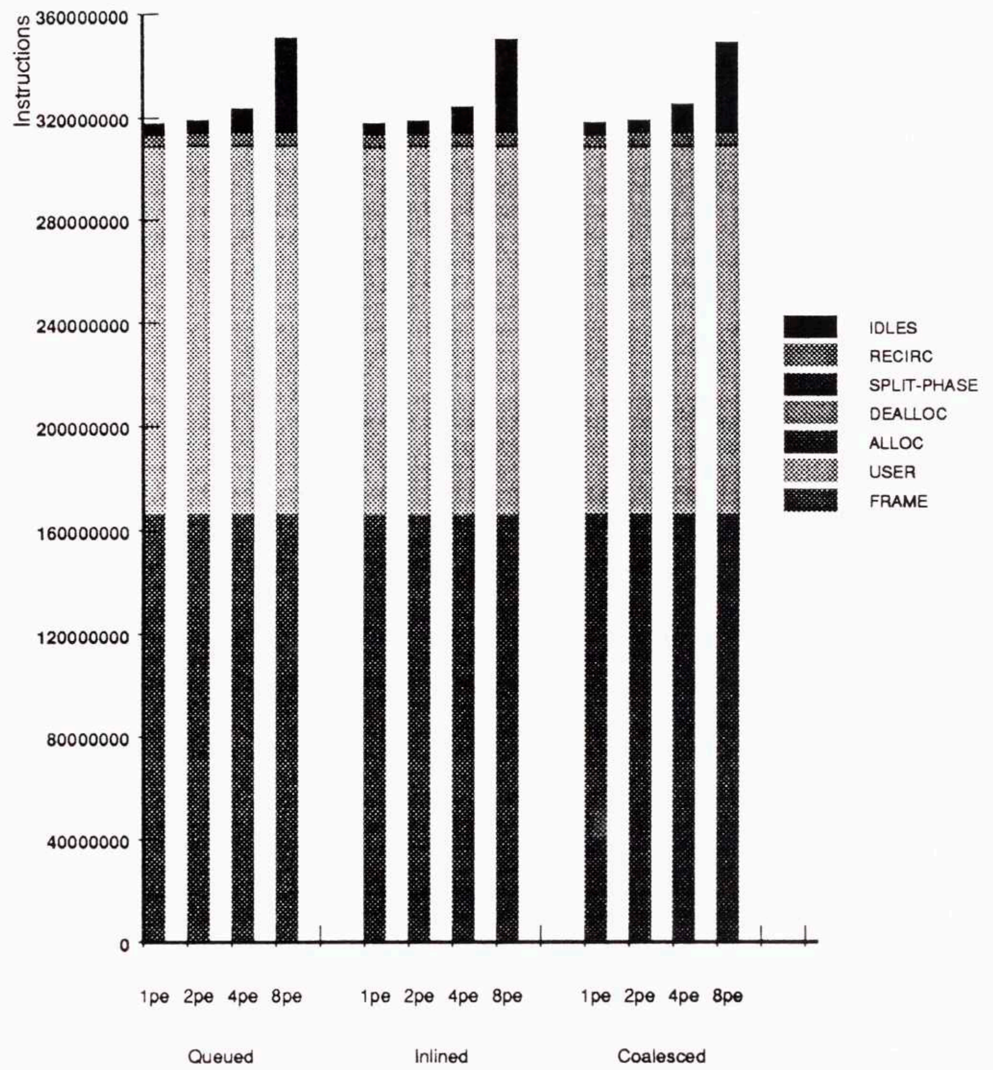

Figure 5.7: Paraffins colors, n = 22

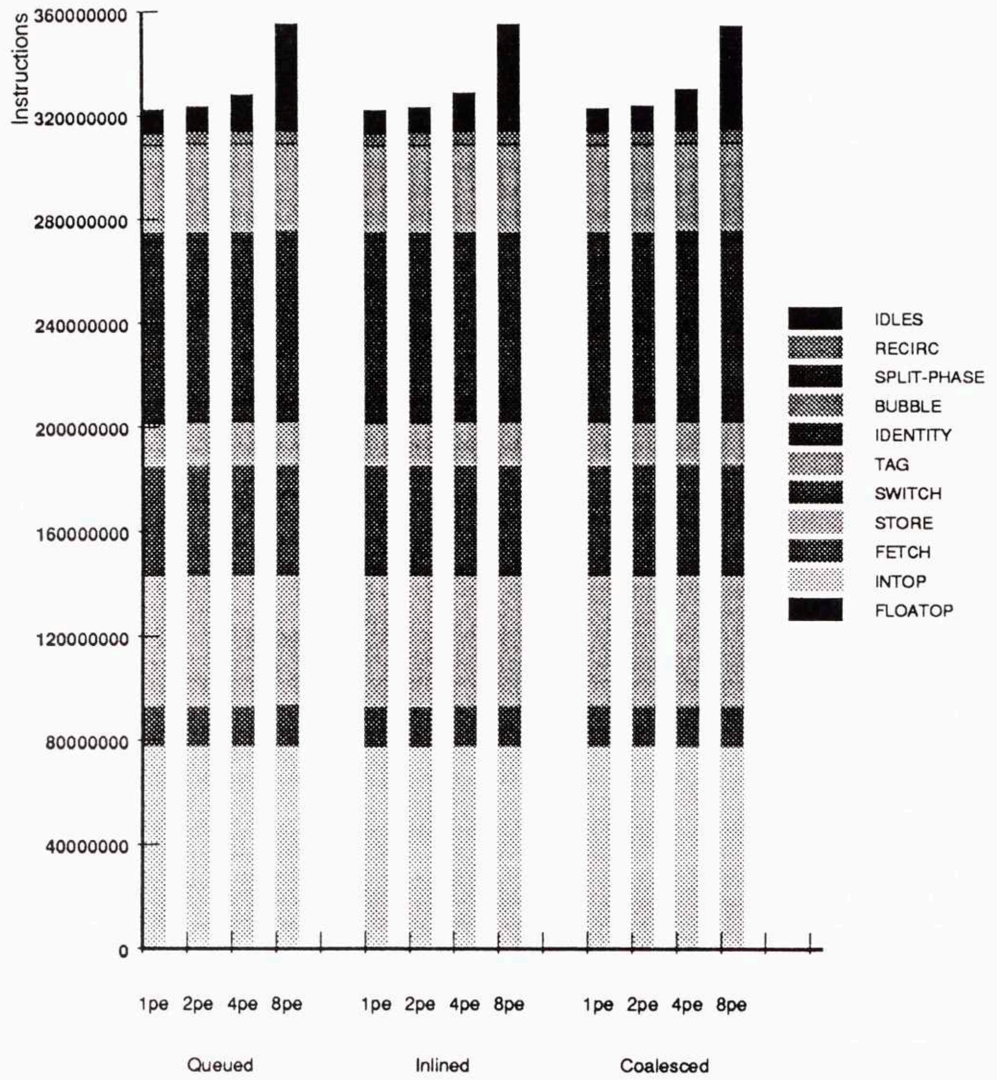

Figure 5.8: Paraffins opmix, n = 22

and we were only seeing about 5.5 times speedup. Recently we ran all of the different configurations with eight I-structure boards, allowing larger runs. Now we see about 7.3 times speedup. On very short program runs, however, the accuracy of our statistics gathering is poor since it is currently impossible to start and stop all the processors at the same time.

We believe that the speedups seen for two and four processors are completely explainable by the hardware hazard. We believe a lack of parallelism in the tail of the computation reduces the efficiency of each processor in the eight processor case. This theory is supported by manually observing the work lights found on the Monsoon hardware. Toward the tail end of the computation, the work lights on all processors get dimmer simultaneously, showing a lack of work. A larger problem size will probably amortize the tail and increase efficiency. The final results are shown in Figure 5.7 and Figure 5.8. Note that heap management is grouped with frame management and is not separated out into alloc and dealloc, even though the graph still lists the alloc and dealloc keys.

5.10.1 Frame Manager Analysis

The heap management used in the Paraffin runs, like the Matrix-Multiply runs, were hand-coded in assembly code for speed. All of the frame managers to perform the same, implying that frame management does not take much time. Most likely, the heap management is consuming the run-time system cycles shown as the *frame* key. Because we hand-coded the heap manager, it is counted as part of the frame management. The heap management's large percentage of the run-time is not unreasonable — we see the same in an equivalent C program.

5.11 Simple

We run Simple for one hundred iterations of one hundred by one hundred matrices. Loop bounds were arbitrarily placed within the code — there are too many loops in Simple. Because the loops to bound were not well chosen, the performance is very poor. We get at most 6.2 speedup on eight processors.

Part of the poor speedup problem is the `while` main loop. The `while` conditional depends on all of the elements in the computed matrix and thus creates basically a barrier between iterations. This barrier is tolerable on a single processor configuration but destroys performance on eight processors. Lam[18] showed that this control dependence can severely reduce parallelism and urges speculative execution to reduce the problem. We can alleviate the problem by simply unrolling the `while` loop a small number of times.

5.11.1 Frame Manager Analysis

Because loops selected for bounding were arbitrary, we were not able to get much speedup performance. We only ran with rts_{queue} and $rts_{inlined}$ to save time. It is clear that our one processor performance is still excellent. The idle count for the one processor case can be completely explained by the hardware hazard. Poor performance for multiprocessors can be easily explained by the loop bounds and the control dependence. We did examine the Monsoon load lights during several program runs and noticed that there were periodic dimmings of the lights, corresponding to the end of an iteration.

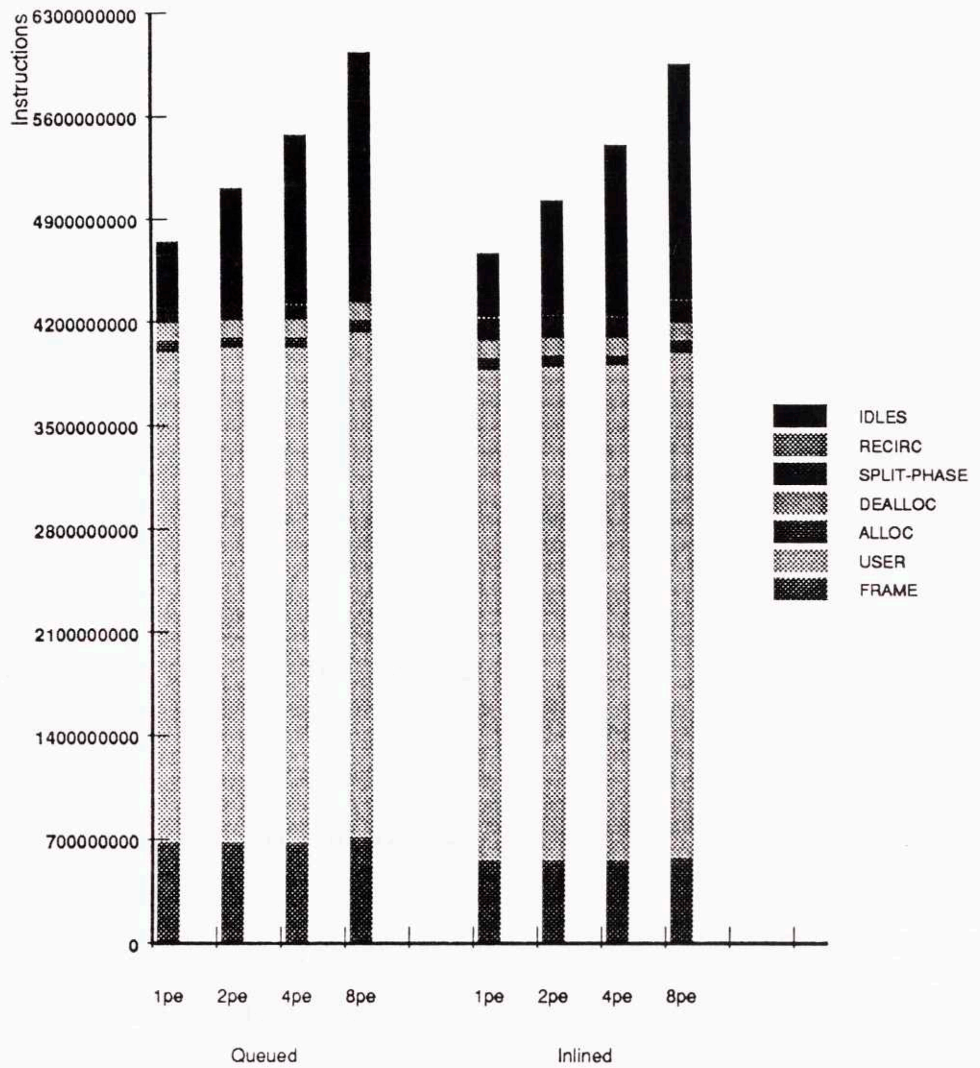

Figure 5.9: SIMPLE 100 iterations, 100 x 100

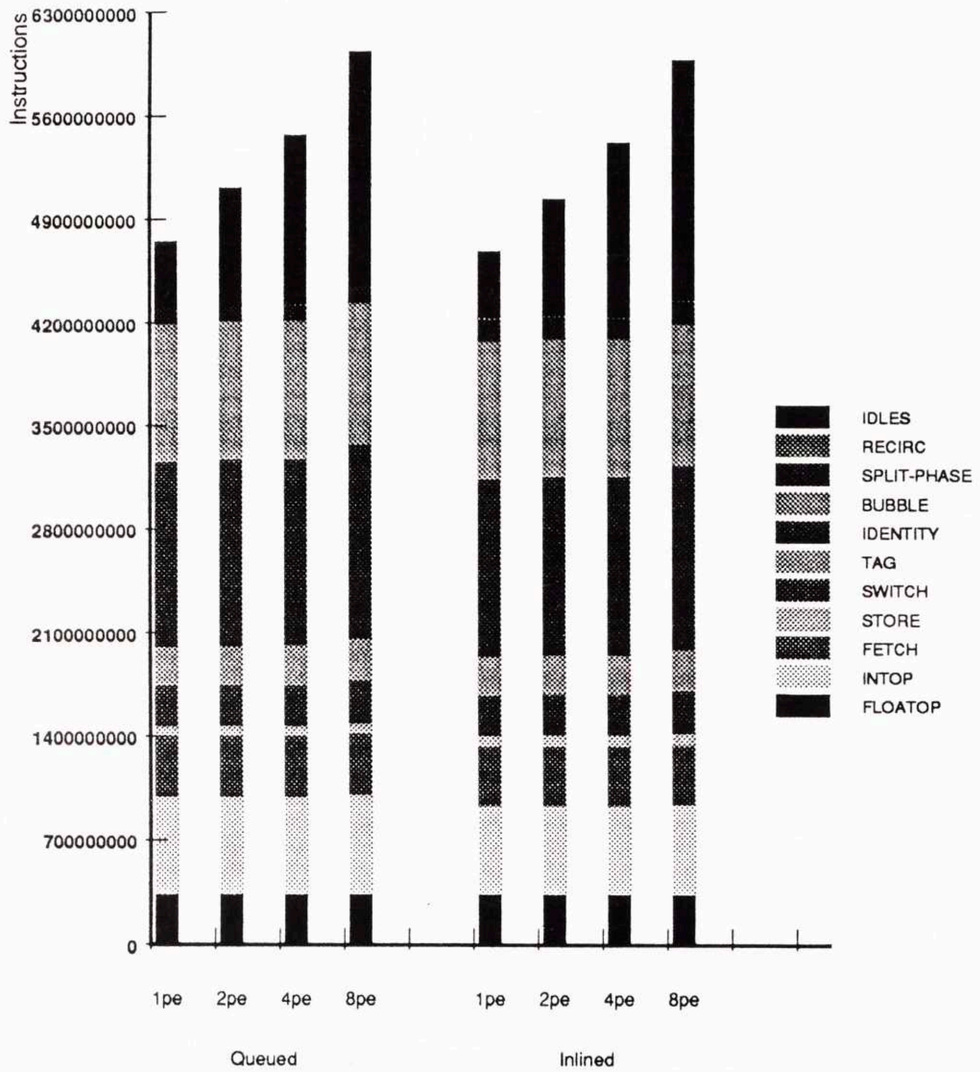

Figure 5.10: SIMPLE 100 iterations, 100 x 100

Chapter 6

Discussion and Future Directions

From the Results chapter, it is easy to see that rts_{queue} produces the best runs. Its critical path is the shortest and it uses more randomized round-robin counters to avoid bunching. Clearly, minimizing frame management latencies does affect application speed when many frames are allocated.

6.1 Future Directions

rts_{queue} and $rts_{coalesce}$ are as fully optimized as $rts_{inlined}$. Some work should be done to equalize the optimization level. Many more points need to be investigated. The first one is testing exactly how much frame management latency affects application run-time. We can do this experiment perfectly in MINT by modifying the fictitious run-time system to consume a constant number of pipeline cycles for every frame allocation/deallocation. Another option is to modify rts_{queue} to execute in some fixed number of cycles. We can do this by adding a counter that will count to a specific number before continuing with the frame allocation/deallocation.

We can also investigate better load balancing techniques. For an accurate study, experiments must be run in MINT since idealized load balancing can be simulated in MINT. We can try other heuristical load balancers such as the one found in rts_{queue},

using a random number generator, and one that queries the load on other processors every so often.

Frame management itself can be further investigated. We should reexamine the scenario where each processor is statically allocated a set of frames that it can allocate. If it runs out of frames to allocate, it either crashes or asks another processor for a frame. A frame manager similar to that could cache a few frames, but have each processor still manage all of its own frames. We could also examine dynamically deciding frame sizes. Currently, we only allocate a predetermined set of frame sizes.

It is possible that the compiler can help us allocate frames more efficiently. For example, if it is known that a procedure will definitely call another, it is possible for the calling procedure to allocate both frames at once by requesting a frame large enough for both called procedures. This will reduce the number of frame allocations at the possible cost of poor load balancing since the super-frame will exist only on one processor. It is also possible for the compiler to inline the run-time system code, thus saving at least three instructions that are required when executing a trap.

6.2 Related Work

Very little has been written up about work related to ours. Sakai and company at ETL in Japan have implemented a series of frame managers, but make no direct reference to the frame managers in their papers. They do discuss load balancing, however, and have performed similar experiments to ours[17].

Parallel Prolog researchers have also probed the frame management question. Prolog must often examine the values stored in ancestor frames. The required values can be copied[7, 14], as we do on Monsoon, or references can be made to ancestor frames[19] in a shared memory scheme. We have not been able to find any literature on their actual frame management techniques when their frame management is similar to our distributed frame memory. Most likely, our solutions are similar to theirs.

Chapter 7

Conclusion

In conclusion, frame management is a tractable problem for eight processors. We discovered that simple systems work best, since they were easy to write, ran quickly, and used the same load balancing scheme as the more complicated systems. Our speedups are quite respectable, since each Monsoon processor requires at least eight fold parallelism to keep busy. Frame manager latencies are important — in fact, frame managers with shorter latencies often run faster than frame managers with slightly longer latencies and shorter critical sections.

A simple round-robin load balancing scheme, though not perfect, works very well. Round-robin works best when the work done in a certain size frame is similar to the amount of work done in other frames of the same size. Although round-robin still performs reasonably when frames of the same size contain different amounts of work, things would work even better if each round-robin counter was associated with code-blocks containing equal quanta of work. Since round-robin depends on the "circular sprinkler" distribution of work, it is important to continuously distribute the work. If frame allocations are lifted, much care needs to be taken to keep load balancing reasonable. Loop bounds must be chosen to provide the right amount of parallelism so that all processors can receive an equal amount of work and care must be taken so that the work really is evenly distributed. The dataflow intuition that given sufficient parallelism, load balancing is relatively easy, is true.

We can explain the idles seen in Gamteb runs as artifacts of Monsoon's hardware hazard. Matrix-Multiply's idles come mostly from the hardware hazard but also from the startup costs of allocating heap objects. Paraffin's idles, at least for an eight processor run, are probably due to the short run-time. The short run-time will make startup and end costs take a larger percentage of the total run-time. Simple's loop bounds need to be improved in order to get the necessary parallelism for good speedups.

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading MA, 1986.

[2] Arvind, S. K. Heller, and R. S. Nikhil. Programming generality and parallel computers. *Fourth International Symposium on Biological and Artificial Intelligence Systems*, Sep 1988.

[3] Arvind and R. S. Nikhil. Executing a program on the Massachusetts Institute of Technology tagged-token dataflow architecture. CSG Memo 271, MIT Lab. for Comp. Sci., Cambridge MA, Mar 1987.

[4] A. P. W. Bohm and J. Sargeant. Efficient dataflow code generation for sisal. , 1985.

[5] P. J. Burns, M. Christon, R. Schweitzer, O. M. Lubeck, H. J. Wasserman, M. L. Simmons, and D. V. Pryor. Vectorization of Monte Carlo particle transport: An architectural study using the LANL benchmark "Gamteb". In *Proceedings Supercomputing '89*, pages 10–20, New York, NY, Nov 1989. IEEE Computer Society and ACM SIGARCH, ACM Press.

[6] D. T. Chiou. A reverse compiler: Monsoon dataflow microcode to common lisp. B. S. Thesis, Massachusetts Institute of Technology, Jun 1989.

[7] J. S. Conery. Binding environments for parallel logic programs in nonshared memory multiprocessors. In *Proceedings of the 1987 IEEE Symposium on Logic Programming*, pages 457–467, San Francisco, Sep 1987.

[8] W. P. Crowley, C. P. Hendrickson, and T. E. Rudy. The SIMPLE code. UCID 17715, Lawrence Livermore Laboratory, Feb 1978.

[9] D. E. Culler and G. M. Papadopoulos. The explicit token store. *Journal Of Parallel and Distributed Computing*, 10(4):289–308, 1990.

[10] J. B. Dennis and D.P. Misunas. A preliminary architecture for a basic data-flow processor. In *Proceedings of the Second Annual Symp. Computer Architecture*, pages 126–132, Jan 1975.

[11] E. G. Coffman Jr. (Ed). *Computer and Job-Shop Scheduling Theory.* John Wiley & Sons, New York, 1976.

[12] J. R. Gurd, C. C. Kirkham, and I. Watson. The manchester prototype dataflow computer. *CACM*, 28(1):34–52, Jan 1985.

[13] J. Hicks, D. T. Chiou, B. S. Ang, and Arvind. Performance studies of the monsoon dataflow processor. Technical Report 345, MIT Lab. for Comp. Sci., Cambridge MA, Aug 1992.

[14] L. V. Kale. The reduce-or process model for parallel evaluation of logic programs. In *Proceedings of the Fourth International Conference on Logic Programming*, Melbourne, Australia, Aug 1987.

[15] D. E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley Publishing Company, Reading, Ma, 1973.

[16] Y. Kodama, S. Sakai, and Y. Yamaguchi. A prototype of a highly parallel dataflow machine em-4 and its preliminary evaluation. In *Proceedings of InfoJapan*, pages 291–298, 1990.

[17] Y. Kodama, S. Sakai, and Y. Yamaguchi. Load balancing by function distribution on the em-4 prototype. In *Supercomputing 91*, pages 522–531. Electrotechnical Laboratory, Jul 1991.

[18] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. In *The 19th Annual International Symposium on Computer Architecture*, May 1992.

[19] T. M. Nguyen. *Hybrid Memory Management for Parallel Execution of Prolog on Shared Memory Multiprocessors.* PhD thesis, University of California, Berkeley, Berkeley, CA, Jun 1990.

[20] R. S. Nikhil. The parallel programming language id and its compilation for parallel machines. In *Proceedings of the Workshop on Massive Parallelism: Hardware, Programming and Applications*, Oct 1989.

[21] R. S. Nikhil. Id version 90.0 reference manual. CSG Memo 284-1, MIT Lab. for Comp. Sci., Cambridge MA, Sep 1990.

[22] R. S. Nikhil and Arvind. Id: a language with implicit parallelism. CSG Memo 305, MIT Lab. for Comp. Sci., Cambridge MA, Feb 1990.

[23] G. M. Papadopoulos. *Implementation of a General Purpose Dataflow Multiprocessor.* PhD thesis, MIT, Cambridge MA, Aug 1988.

[24] G. M. Papadopoulos. Program development and performance monitoring on the monsoon dataflow multiprocessor. CSG Memo 303, MIT Lab. for Comp. Sci., Cambridge MA, Oct 1989.

[25] G. M. Papadopoulos and D. E. Culler. Monsoon: An explicit token store architecture. In *Proceedings of the 17th International Symposium on Computer Architecture*, May 1990.

[26] G. M. Papadopoulos and K. R. Traub. Multithreading: A revisionist view of dataflow architectures. In *The 18th Annual International Symposium on Computer Architecture*, May 1991.

[27] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba. An architecture of a dataflow single chip processor. In *Proceedings of the International Symposium on Computer Architecture*, pages 1155–1160, 1989.

[28] T. Shimada, K. Hiraki, K. Hishida, and S. Sekiguchi. Evaluation of a prototype data flow processor of the Sigma-1 for scientific computations. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 226–234. IEEE, Jun 1986.

[29] B. J. Smith. Architecture and applications of the HEP multiprocessor system. In *Real-time Signal Processing IV*, volume 298, pages 241–248, Aug 1981.

[30] K. R. Traub, G. M. Papadopoulos, M. J. Beckerle, J. E. Hicks, and J. Young. Overview of the monsoon project. In *Proceedings of the 1991 IEEE International Conference on Computer Design*, Oct 1991.

[31] Y. Wang and R. J. T. Morris. Load sharing in distributed systems. *IEEE Trans. on Computers*, C-34(3):204–217, 1985.

[32] J. Young and D. T. Chiou. Context mangement in the id run time system. CSG Memo 319, MIT Lab. for Comp. Sci., Cambridge MA, Sep 1990.

4435-22