LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Performance Studies of Id on the Monsoon Dataflow System

CSG Memo 345-3
(This supersedes CSG Memo 345-2)
May 17, 1993

James Hicks
Derek Chiou
Boon Seong Ang
Arvind

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-89-J-1988.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Performance Studies of Id on the Monsoon Dataflow System

James Hicks, Derek Chiou, Boon Seong Ang, and Arvind

May 17, 1993

Abstract

In this paper, we examine the performance of Id, an implicitly parallel language, on Monsoon, an experimental dataflow machine. One of the precepts of our work is that the Id run-time system and compiled Id programs should run on any number of Monsoon processors without change. Our experiments running Id programs on Monsoon show that speedups of more than seven are easily achieved on eight processors for most of the applications that we studied. We explain the sources of overhead that limit the speedup of each of our benchmark programs.

We also compare the performance of Id on a single Monsoon processor with C/Fortran on a DEC Station 5000 (MIPS R3000 processor), to establish a baseline for the efficiency of Id execution on Monsoon. We find that the execution of Id programs on one Monsoon processor takes up to three times as many cycles as the corresponding C or Fortran programs executing on a MIPS R3000 processor. We identify the sources of inefficiency on Monsoon and suggest improvements, where possible. In many cases, however, improving single processor performance will reduce parallel processor performance.

1 Introduction

The Monsoon Dataflow Multiprocessor, built jointly by MIT and Motorola, is a small shared memory multiprocessor. The motivation behind building Monsoon research prototypes was to demonstrate the feasibility of general purpose parallel computers, suitable for both numerical and symbolic applications. Since writing parallel programs is itself a research issue, an inseparable part of this project was the demonstration of the implicitly parallel programming language Id, which allows the user to write parallel programs without worrying about the details of parallelism. In this paper, we evaluate the performance of Id programs running on Monsoon.

Our goal was to show that one could easily write parallel programs in Id and that these programs would run efficiently on Monsoon. The first part of this goal was fairly straightforward. We took programs written in Id by various people, and showed that they had parallelism on Monsoon. It was easy to determine how much Id programs sped up on one to eight processors, which was the largest Monsoon configuration constructed. Since the Monsoon processor, like the Denelcor HEP [36], consists of an eight-stage interleaved pipeline, an 8-processor Monsoon requires at least 64-fold parallelism to achieve 100% utilization. Thus, even experiments on a small 8-processor configuration reveal important issues, such as the amount of parallelism that programs are able to exploit, contention for network and processor resources, and to a lesser degree, the issues of work distribution, load balancing, and data structure distribution. We have had considerable experience with speedup studies from our experimentation with TTDA [2, 3]. Not surprisingly, the results in this area turned out to be very encouraging — we easily achieved speedups of more than 7 on 8 processors for most of the benchmark programs (see Table II).

The second part of the goal — showing that Id programs run efficiently on Monsoon — was rather daunting. We wanted to understand *why programs did not speedup perfectly, i.e.*, by a factor of 8 on 8 processors. Was the limit on speedup due to a lack of parallelism in the programs? We thought that this limitation was highly unlikely because of the overwhelming evidence to the contrary from the previous experiments on TTDA. Did a defect in the architecture or the run-time system prevent perfect speedups? This paper sheds some light on the aspects of the architecture and the run-time system that limited the achieved speedups.

Between November, 1990 and September, 1992, we spent a lot of time tuning the Id compiler and the Id run-time system (RTS) to improve absolute performance and speedups. This tuning resulted in approximately a factor of 3 speed improvement over our first successful execution of benchmarks on a single processor Monsoon system (see Table I). The effort deepened our understanding of some of the short comings of the Monsoon (or perhaps any pure dataflow) architecture, and also illustrated the inherent cost of asynchronous execution which underlies our compiling strategy. It also confirmed some of the well known problems of generating efficient code from "non-strict"[1] languages like Id. Again, the salient aspects of this effort are documented in this paper.

Regardless of the speedups we were able to achieve, however, a skeptic of dataflow architectures would want to know how the Monsoon architecture compares to commercial architectures, and how Id compares to C/Fortran. But there are many obvious difficulties in making such comparisons. The development cost of Monsoon, a research project, is minuscule in comparison with the development cost of commercial machines. Consequently it runs at 10 MHz, a clock speed 3 to 5 times slower than commercial machines from the same time period. Some respectable computer architects hold the view that any comparison of machines with such major differences is meaningless because implementation limitations determine the architecture of commercial machines. We think otherwise — we think it is possible to normalize over implementation technologies up to a point and learn about the intrinsically good and bad properties of architectures. The greatest difficulty in compar-

[1] Non-strict: The body of a procedure can start to execute before all of its arguments have arrive.

ing the Monsoon architecture with commercial machine architectures, however, is not these technology issues but software.

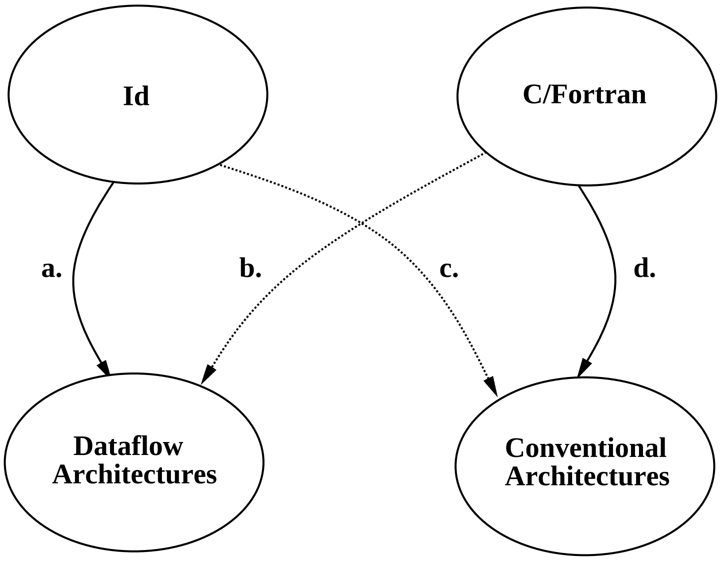

Figure 1: Compilation Paths

Consider the diagram in Figure 1 to understand the full dimension of our problem. One needs to run more or less the same programs on two different machines to compare performance. This implies that either Id should be run on some other parallel machine or some version of Fortran or C should be run on Monsoon. Both of these paths pose serious practical problems. To begin with, we are not aware of any compiler that takes standard Fortran or C programs and compiles them for commercial parallel machines with any success. (The results published in [9], for example, shows that poor speedup is achieved for most of the programs in the Perfect Benchmarks suit when a parallelizing compiler is used to parallelize the programs for the Alliant/FX80 system.) Instead, every vendor has its own version of parallel extensions to Fortran and C which allow a user to express his algorithm for a particular architecture or a specific configuration of the machine. In fact, the underlying architecture has such a profound influence on the way users write parallel programs that the language choice is of secondary importance.

The development of the data-parallel programming model offers a good example to illustrate the above point. The model was developed in the context of the Connection Machine [18], and Thinking Machines provides data-parallel extensions for Fortran and C. However, data-parallel programming in these extended sequential languages does not resemble sequential programming except in some superficial syntactic sense. It may be possible to develop a set of benchmark data-parallel programs to evaluate parallel machines from a data-parallel point of view. However, such an evaluation would not be suitable for Monsoon which has very different goals. The data-parallel model is not a general-purpose model for parallel programming, and data-parallel programming remains significantly harder than sequential programming. In fact, the difficulty of writing parallel programs is the *raison d'etre* for our approach.

Id, the language we use on Monsoon, is an *implicitly parallel* language — the compiler merely exposes the parallelism implicit in the program. The programmer is not required to write his program for a specific machine architecture or configuration. In fact, to a degree, the programmer does not even have to be aware of the parallelism in his program. Id programs on Monsoon are compiled to be executed in parallel — the same object code runs on any number of processors. Id is a high-level declarative (functional) language with extensions for single assignment data structures (called I-structures [7]) and for updateable data structures with fine-grain synchronization (called M-structures [8]) [25]. Using Id, it may one day be possible to treat high-level sequential programming as a special case of parallel programming!

Going back to Figure 1, another way to establish the performance of Monsoon was to develop Id compilers for commercial parallel machines. Compiling Id for commercial parallel machines, however, also had problems. Id-like languages offer expressivity but generally are difficult to compile efficiently for systems based on conventional processors. The non-strictness of Id makes it easy to generate a multitude of parallel threads[2] but makes it very hard to sequentialize these threads for efficient sequential execution (Strict implicitly parallel languages like Sisal [11], on the other hand, suffer from fewer compilation problems but generally exploit less parallelism.) Improper over-sequentialization of threads can lead to deadlocks, while under-sequentialization of threads leads to poor performance. The functional languages community has made very significant progress in the last few years in compiling non-strict languages for conventional architectures [20]. Compilation of Id on stock hardware has been pursued by Culler *et al.* [34] and Nikhil [26]. Results, however, are still preliminary. The initial performance figures of Id on CM-5 as reported by Culler [37], for example, suggest that Id on a 64-processor CM-5 may do just about as well as Id on an 8-processor Monsoon! Besides the fact that the Id compilers for stock hardware are still immature and lack many optimizations, such a comparison only points out that Monsoon has much better support for non-strictness and parallelism than stock processors; this comparison does not help us establish a base line performance of Monsoon.

Our final decision was to compare Id running on a single-processor Monsoon with C or Fortran running on a standard workstation to establish some base line performance. Barring hand-tuned assembly code, one cannot get code more efficient than C and Fortran programs compiled with good compilers for conventional uniprocessors. Though the comparison is unfair to us — our software is meant to run in parallel, and thus, has overheads that sequential C and Fortran implementations do not have — it gives us an idea of the amount of overhead we incur. We show that Id on Monsoon takes 2 to 3 times as many machine cycles as Fortran/C on MIPS R3000 (see Table VII), and we analyze the causes of this overhead.

Finally, the reader may wonder how well Fortran or C would run on a single processor Monsoon. Running sequential code efficiently was not a goal of the Monsoon project. For

[2] A thread is simply a sequential fragment of code, where each instruction executes after the one before it.

sequential programming, Monsoon instruction set may be viewed as a single-accumulator type of instruction set (reminiscent of the earliest von Neumann machines) [30]. We did not need any experimental evidence to show that Monsoon cannot compete with a modern RISC processor in executing Fortran or C!

1.1 Overview

In Section 2, we discuss our execution model for parallel programs, the Monsoon system, including its hardware and software, and our implementation of Id on Monsoon. In Section 3, we describe the benchmark programs that we ran to measure performance. The source code for these benchmark programs are available upon request to `jamey@lcs.mit.edu`. In Section 4, we discuss the ways in which the Id compiler and run-time system were improved in order to get better performance on a single processor Monsoon. In Section 5, we discuss the scalability and implicit parallelism studies, and in Section 6, we establish Monsoon's baseline performance by comparing the execution efficiency of Id running on Monsoon with that of Fortran/C running on a MIPS R3000. We also discuss our comparison methods and rationales. Finally, in Section 7, we present our conclusions and discuss some strategies for the future.

2 Id on Monsoon

To understand our results, it is necessary to understand our parallel execution model, dataflow and split-phase transactions, the Monsoon architecture, and the implementation of Id on Monsoon, including the resource management issues. Each of these topics is discussed in this section.

2.1 A Parallel Execution Model

One often thinks of the execution of a high-level language program on a sequential machine in terms of a *stack of activation frames* and a global *heap of objects*. An activation frame is allocated when a procedure is called and reclaimed when it terminates. A data structure resides in the heap if its life-time can be longer than the procedure activation that creates it. This model can easily be generalized for parallel execution by turning the stack of activation frames into a *tree of activation frames*. A procedure may spawn multiple procedures or loops in parallel, but the activation frames of child procedures may not be allocated on a stack because they may finish in an order unrelated to that in which they were invoked. When a procedure finishes, it is guaranteed to be a leaf in the activation tree because all of its children must already have completed, and thus, it is safe to remove its frame from the activation tree and reclaim it for reuse. This basic parallel execution model is shown in Figure 2.

All of the frames (not just the leaves) in the activation tree are potentially active. The code associated with each frame will generally have more than one "thread" of execution running simultaneously. There are three levels of parallelism to exploit in this model: processor level, consisting of procedure activations on separate processors, thread-level, consisting of multiple threads active on each processor, and instruction-level, consisting of instructions within a thread. It is this last type of parallelism that is exploited by super-scalar processors. Dataflow computation on the other hand tries to exploit all three types of parallelism.

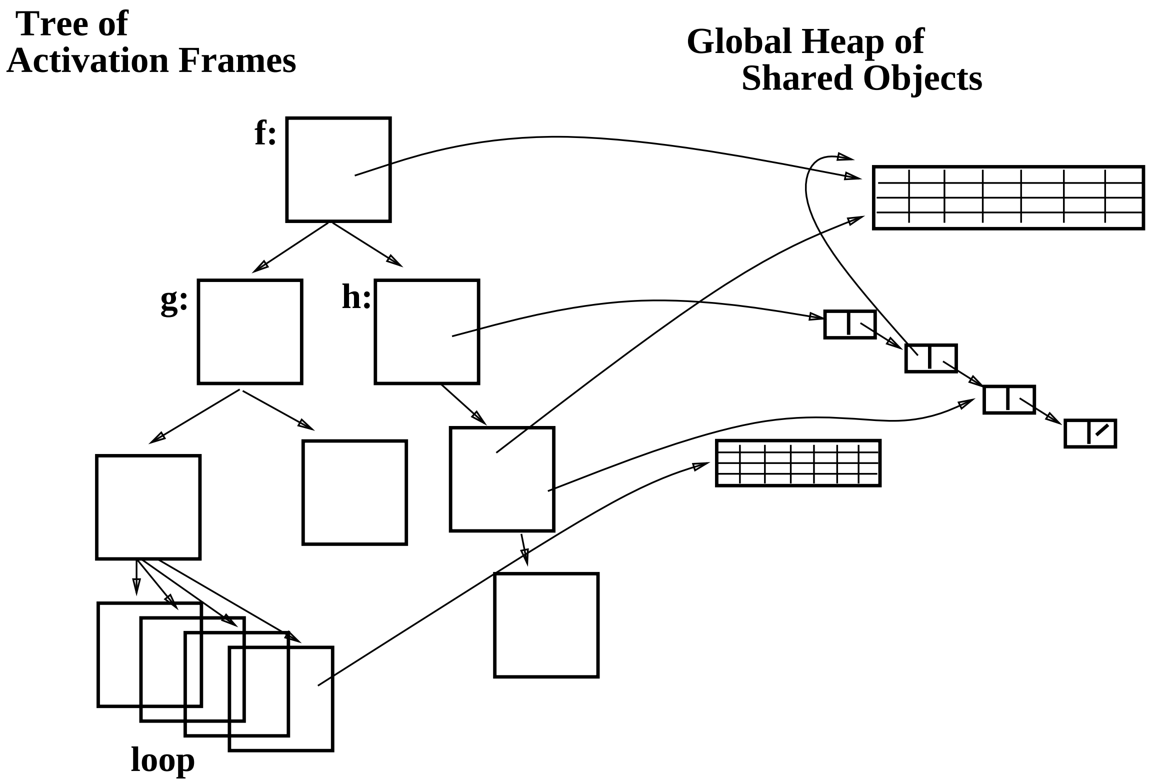

Figure 2: Fully Parallel Execution Model

The reader should note that this execution model has nothing in particular to do with Id; it should be applicable to parallel execution of just about any high-level language. However, Id exploits this model to the fullest. The non-strictness of Id permits the execution of a child procedure to begin even before all of its arguments have been computed. It also allows loops to unfold dynamically, constrained only by data dependencies. Id, via I-structures and M-structures, also allows very fine-grain producer-consumer parallelism between parent and children procedures and between sibling procedures. In fact, because of all these properties, Id tends to expose *too much* parallelism, often exhausting machine resources, especially frame memory, if not controlled. A simple way to limit the amount of resources required is *bounded loops*, where a bound is specified to indicate how many loop iterations can occur in parallel. Though currently the user must input the loop bounds either as parameters or by computing them at run time, we hope to greatly reduce if not eliminate that requirement using compiler analysis and run-time system support in future.

The execution model presented above has a natural mapping on shared memory MIMD machines: the frames reside in the local memory of the processor where they are assigned to execute, and the heap objects reside in the global shared memory, which can be accessed from any processor. Since a frame is accessed only locally, it is straightforward to cache frames. Caching of heap store, however, raises the usual multiprocessor cache coherence issues. To exploit parallelism effectively in this model, two architectural issues — memory latency and synchronization — have to be addressed [5, 27]. Dataflow architectures offer a solution to these problems as will be discussed in the next section.

Whenever a procedure is invoked, a frame or a set of frames (in case it is a loop procedure) needs to be allocated. Since the frame store is tied to processors, distribution of work depends upon how and where the frame manager allocates frames which cannot migrate. If the heap storage has non-uniform access time, then the heap manager can try to allocate storage to maximize locality. Thus, the issues of resource management and load balancing are intimately tied in such a model.

2.2 Split-Phased Transactions and Dataflow Execution

Interprocessor communication generally takes much longer than local communication. For example, fetching data from a frame usually is much faster than fetching data from remote memory. However, a processor may hide the latency of a remote request by doing other work while the request is in progress. In order for this latency toleration to be successful, the processor must be able to switch between activities very quickly and it must be able to deliver a response to the activity that made the request. Furthermore, the program must have sufficient parallelism for the processor always to have something to do while a request is in progress.

In the dataflow model, remote requests are structured as *split-phased* transactions so that multiple requests may be in progress at one time. An instruction issues a *request* to the processor or memory module containing the desired data, and then executes other instructions which do not depend upon the result of the request in progress. The request carries a tag, *continuation*, indicating the procedure activation and the instruction at which the computation should be continued when the response arrives. In our dataflow model a tag, or *context*, is composed of an instruction pointer (IP), a frame base pointer (FP), and a node number. Since responses to requests from a processor can be reordered due to the network and synchronization, tags are essential to match requests and responses correctly.

When processors communicate, they also need to *synchronize* to ensure that valid data is used and to avoid race conditions. Id I-structures and M-structures provide high-level semantics for fine grain synchronization, and can be implemented efficiently by providing a few *presence bits* for each word of memory [19, 36]. An I-structure has the property that a read-request that arrives before the write to that location is *deferred* until that element has been written. A deferred request can be memorized simply by saving its continuation (the return-tag). Once the value is present, it can be sent to the requesters using the saved return-tags [6]. Because of this synchronization, the latency of a request may be much longer than the actual network delay. However, this synchronization allows us to write deterministic programs, and the split-phased nature of remote requests allows us to hide the extra latency, given enough parallelism in the program.

In order for split-phased transactions to be effective in hiding latency, the overhead of issuing a remote request must be very low. Most commercial microprocessors have very poor network interfaces, and can take hundreds of cycles just to put the message, corresponding to the first-phase of a split-phased memory request, on the network. On the other hand, dataflow architectures, including Monsoon, Sigma-1 [19], and EM-4 [33], have network interfaces that blend seamlessly with the processor pipeline, and can produce a message for the network basically every cycle.

On a conventional processor, if some needed data is not available, the processor must either wait for the data to arrive or switch to another thread, occasionally polling for the data arrival or waiting for an interrupt announcing the arrival of the data. On a parallel computer, more than one thread may have to be executed while a request is in progress in order for the processor to stay busy. This adds the additional constraint that thread-switching must have low overhead, and that there must be a mechanism for resuming the execution of a thread when the required response arrives. Commercial microprocessors generally have very large context switching time and expensive interrupts due to large processor state. Dataflow architectures solve these problems by executing all instructions in a data-driven manner, and by making threads very light weight. The state of a dataflow thread is just a single *token*: a data value along with its continuation. Light-weight threads, together with hardware management of thread queue, allows threads to be switched in a single cycle! However, as can be expected, the small state of a thread in a pure dataflow machines is detrimental to good sequential performance.

For split-phased transactions to work efficiently, the overhead of synchronizing the response with the consumer must also be very low. High-performance sequential computers make some use of split-phased transactions by allowing a load instruction to issue a request, and continuing execution until an instruction that makes use of the register that was the target of the load is encountered. In this case, synchronization of the response and the consumer is done through the hardware register scoreboard and is very efficient. But these systems typically do not allow load requests to complete out of order. Furthermore, these techniques are of little use when the data to be read may be missing; the requesting task has to release the processor to avoid deadlocks in case of a synchronizing read [27].

Dataflow architectures employ a very general and efficient synchronization mechanism which is used in all instructions requiring two operands or instructions waiting for a remote fetch to complete. In a dataflow machine, in contrast to a von Neumann machine, an instruction is scheduled for execution when all of its operands are available. For example, the expression $(a + b)$ is evaluated when the values, carried by tokens, for both a and b become available. The tokens carrying a and b can arrive in arbitrary order, and therefore whenever a token arrives the processor must check the availability of its partner. If the partner has not arrived, the token must be saved until the arrival of the partner [6]. In modern dataflow machines this type of synchronization is performed using an *explicitly-addressed token store* [15, 32]. Tokens synchronize at a compiler ordained offset into an activation frame. Thus, when a token is processed, only the presence bits of the specified memory location are examined to see if the token's partner has already arrived. When the second token of an add instruction arrives, the value of the first token is fetched and the addition is performed. The result of $(a + b)$ is then packaged into a token and sent to the instruction or instructions that need it.

This style of execution allows dataflow architectures to exploit all the parallelism inherent in a particular program modulo system software and hardware constraints. In the next section, we discuss how all these mechanisms are implemented in Monsoon.

2.3 Monsoon Hardware

The largest Monsoon multiprocessor constructed is made up of eight processing elements (PE) and eight I-structure processors (IS) connected by a two-stage, packet-switched, butterfly network composed of 4×4 switches. Each PE in the current implementation runs at 10 MHz, and is capable of processing up to ten million tokens per second. It has 128 kilowords of instruction memory, where each instruction word is 32 bits; 256 kilowords of frame memory, where each data word is 64 bits, plus three presence bits and eight type bits, the last of which are currently not used. It also has two token queues which can hold up to 32K tokens each. (A token is roughly twice the size of a data word). The current implementation of Monsoon's processor has no caches, and thus is implemented using SRAMs exclusively.

Global memory on Monsoon is implemented with I-structure processors [38]. Access to global memory always occurs over the network, and every global read or write is a *split-phased* operation. A PE may also access the frame memory of another PE by sending split-phased requests to that processor. Global addresses are interleaved across the IS nodes. Each IS contains 4 megawords of 64-bit data memory with associated presence and type bits, and is implemented using DRAMs.

The network interface to each node (which is either a PE or an IS) has a bandwidth of 100 Mbytes per second. This translates to about four million tokens per second for each node. When Monsoon's network is unloaded, a token takes about 13 cycles to go from one node to another. Monsoon's network interface is capable of delivering a token to the network *every* cycle, though the network is not capable of delivering that many tokens at a sustained rate. Token formation, that is packaging up the destination and the value, is performed automatically in most cases. Though token formation and delivery to the network interface is extremely efficient, compiled code rarely, if ever, saturates the network.

2.3.1 Monsoon's Processing Element

The Monsoon processor has a 8-stage pipeline as shown in Figure 3. It consists of an instruction fetch stage, an effective-frame-address computation stage, a presence-bits operation stage, a frame operation stage, a three stage ALU, and a form token stage. The instruction fetch, effective-frame-address, and ALU stages perform the tasks specified by their names. The presence-bits operation stage can read, modify, and write presence bits in one cycle. The frame operation stage either reads a value from a frame, writes a value in a frame, exchanges a value (takes 2 cycles), or does nothing. The form-token stage generates zero, one, or two tokens each cycle.

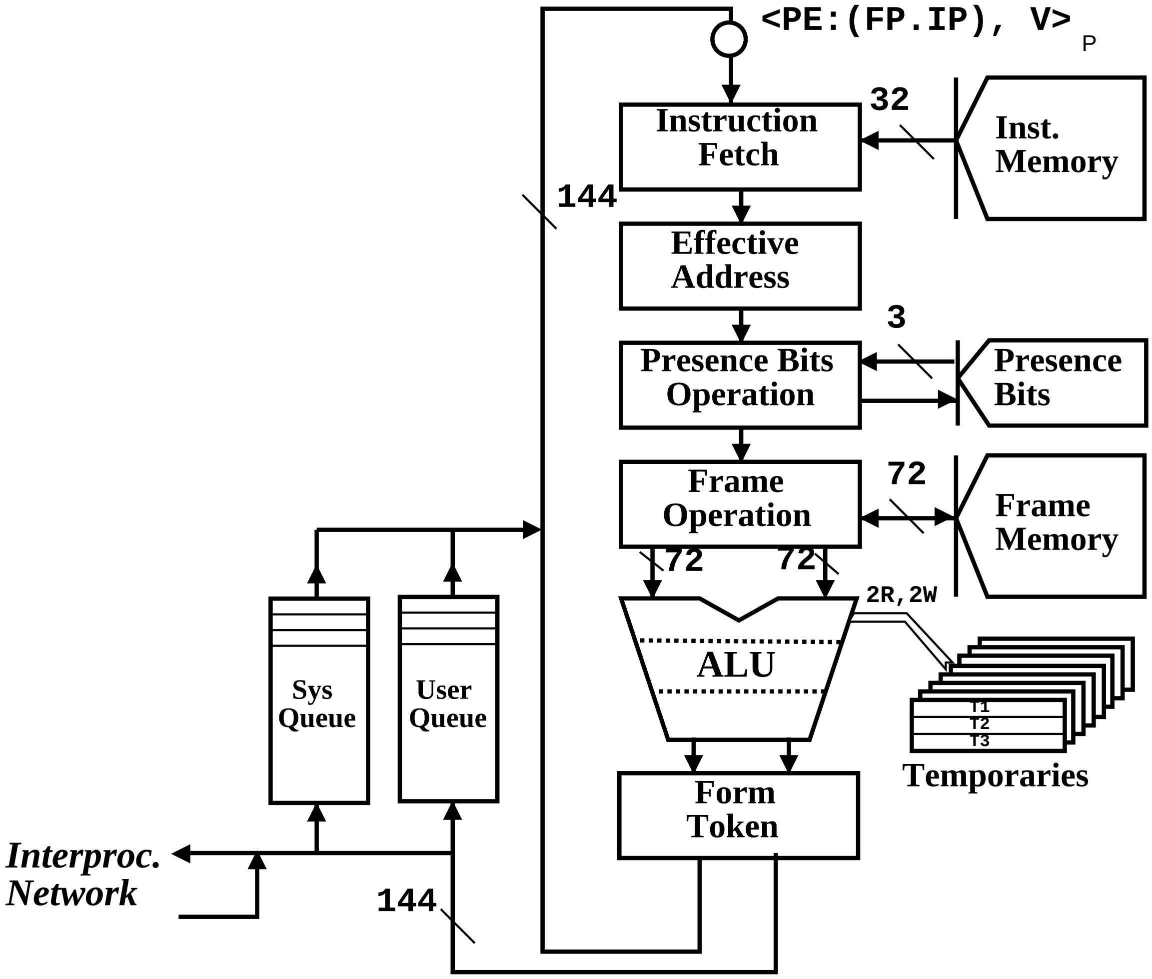

Figure 3: Conceptual View of Monsoon's Pipeline

Tokens are kept in *token queues* while waiting to be processed. Each PE has two token queues one for system tokens and the other for user tokens. The system token queue is needed to ensure that some critical tokens, such as a token to halt the system, can always be processed.

During each cycle, the processor tries to execute a recirculating token coming from the bottom of its pipeline. If no recirculating token was produced the last cycle, it pops a token from its own token queues to execute. Execution results in either zero, one, or two tokens at the end of the eighth cycle. These result tokens are either: (i) circulated to the head of the pipeline; (ii) sent to the network to be delivered to either another processor's system queue or an I-structure board; or (iii) enqueued in either the local processor's system or user token queue, depending on the operation. If no token is produced, then one is taken from the system or user queue to be executed next. If no token is available, then the processor *idles* for one cycle.

The eight tokens being processed at any given time are independent of each other — that is, no token in the pipeline could have created another token that exists in the pipeline at the same time. This is so because the "next" instruction is not computed until the 8th cycle of the pipeline, Thus, keeping an eight processor Monsoon busy (non-idle) requires at least 64-fold parallelism.

Since Monsoon accepts only one token at a time for processing, the first token of a 2-input instruction to arrive will cause the execution unit of the processor to idle. Since the second token could not have arrived yet, there is no work for the processor to do, other than to store away the value from the first token. This idling of the execution stages is called a *bubble* in the pipeline. Bubbles are unavoidable whenever synchronization is implemented as a stage of a pipelined machine that takes only one token per cycle.

Monsoon can execute a sequential thread of instructions which are scheduled at every 8th cycle. Except for the first instruction, each instruction in this sequence must take only one input token which comes from the previous instruction in the thread (via the recirculating path). Furthermore, each instruction except the last must be annotated as *critical*, which ensures that another thread cannot enter the original thread's pipeline slot. In the absence of the critical annotation on an instruction, a network token can displace the original thread from its pipeline beat. Such "critical" threads are broken by either synchronization points, trap instructions or split-phased transactions. An instruction in the middle of a critical thread can produce two output tokens, but only one of the output tokens can be critical. The other token must be pushed onto the token queue or sent to the network. Only a restricted subset of Monsoon's instruction set can be used in critical threads.

To enhance the performance of such critical sequential threads, each Monsoon processor has eight register sets of three temporary registers each. Each register set is associated with one of the eight interleaved threads. Registers are not saved or restored implicitly, and can only be used within an unbroken, *critical* thread of instructions. Currently only RTS code makes use of these registers. Notice, if a single, sequential thread of computation is executed on Monsoon, the pipeline utilization would be 12.5%, because seven of the eight pipeline stages would be idle. Again this $1/n$ utilization will be true for any processor pipeline design with fixed interleaving and n-stages.

It is possible to simulate I-structure operations in the Monsoon processor's frame memory because it also has presence bits. This allows the compiler and the run-time system to allocate some heap objects locally. Reading and writing to I-structures located in processor memory requires the second phase of the split-phased transaction to be executed on the processor containing the structure.

In addition to join, I-structure and M-structure types of synchronization, Monsoon supports synchronization through *spin-locks*. If an instruction attempts to acquire a spin-lock on a location which is already locked, a token to reexecute the same instruction is *recirculated* in the pipeline until the lock is freed. Only the frame and heap managers spin-wait, because, if used improperly, spin-waiting can cause deadlocks. Eight threads simultaneously spin-waiting on one processor will cause a live-lock in the pipeline, because no thread will ever be able to unlock any of the locations on which they are spinning. However, spin-waiting in some instances is much more efficient than M-structure synchronization.

A structural hazard: Because of limited board space and resources, Monsoon has a structural hardware hazard that affects token movement to and from its token queues and network interface (please refer to Figure 4). Essentially, the path from the token queues to the top of the processor pipeline and the path from the bottom of the processor pipeline to the network queues are the same path — the implementation uses a single bus for both paths. If an instruction generates two tokens, one that will recirculate and one that goes to the network, everything is fine. On the other hand, if an instruction generates a single token destined for the network, it must get a token from the token queues in order to keep the processor busy. Since the path to the network is the same as the path from the token queues, the processor cannot send a message to the network while pulling-off a token from the token queue. The network token always has higher priority for the shared bus, since results from the pipeline will be lost if the network token is delayed and a token from the queue is fetched first. Thus, when an instruction produces a single token which goes to the network, the pipeline must idle for one cycle in which the token is delivered to the network interface. A token is then read from the token queue in the next cycle, unless the next instruction in the pipeline also excites the hardware hazard.

The maximum number of idles that can be caused by the hardware hazard is 50% of the total cycles, but we have never seen any thing close to this amount. A similar hazard exists when the network tries to put a token into a token queue.

Most reads destined for the I-structure processors generated by the Id compiler create a network token, but no local token, forcing the structural hazard to insert an idle cycle in the processor pipeline. We will see the effect of this structural hazard in the statistics presented later.

2.3.2 Monsoon's Instrumentation

Monsoon has hardware instrumentation for performance measurement. Each processor has 64 statistics counters and each cycle of execution on Monsoon is accounted for by incrementing one statistics counter. The decoded instruction specifies the statistics register to be

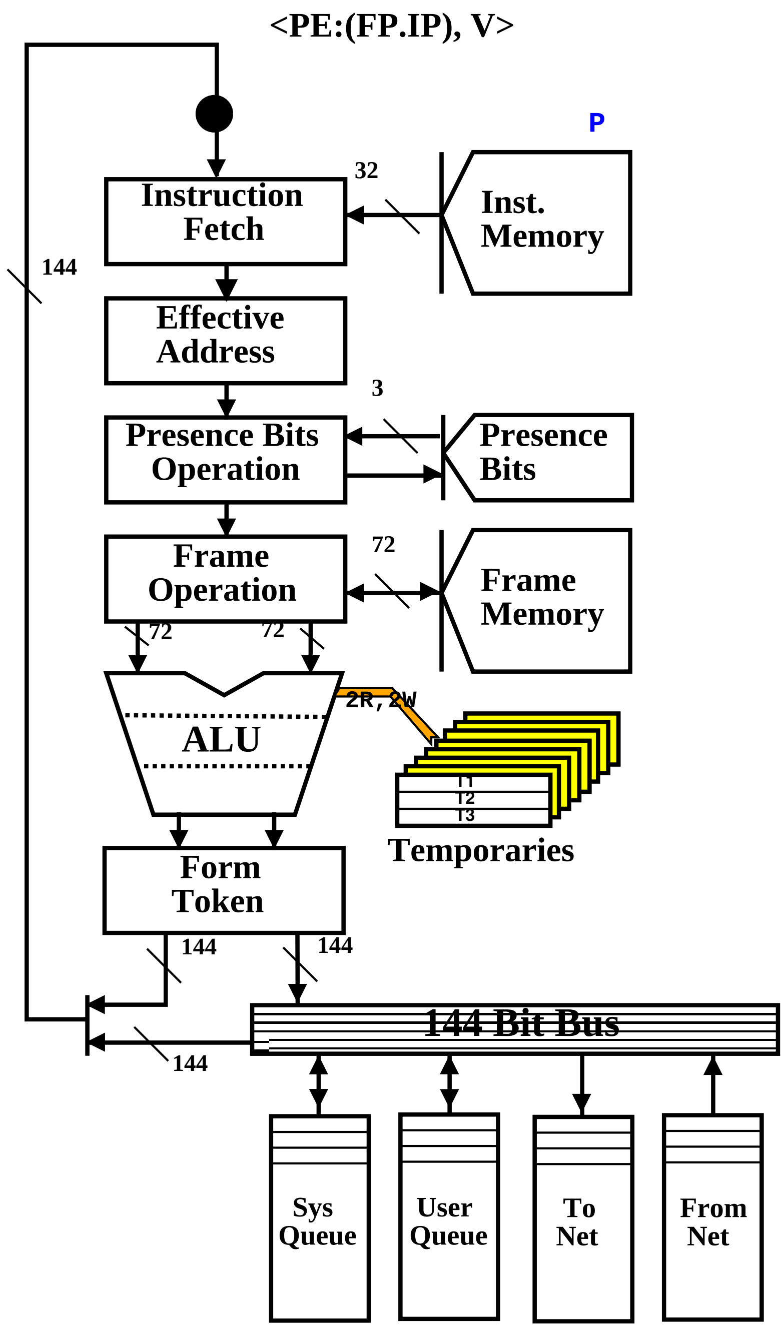

Figure 4: Monsoon's Real Pipeline

incremented. In order to account for every cycle of execution time, idle cycles also increment a statistics counter.

Monsoon's instrumentation also allows us to define between 1 and 64 groups of procedures, where each group is called a *color*. Procedures can either be specified to be in a specific color or to inherit the color of the procedure that called it. Statistics for each color are collected independently; thus, we can generate statistics divided into separate categories for each procedure group. More details on Monsoon's instrumentation are available in [24, 29].

For statistics collection, Monsoon's instructions, or rather cycles, are divided into a small number of categories. We explain these next.

Monsoon Operation Categories: The *integer* and *floating-point* operation categories are self-explanatory. The *fetch* and *store* categories refer only to the cycle in which an instruction issues remote memory requests. *Tag* operations are instructions that manipulates continuations, such as for sending arguments to and results from a called procedure. *Identity* operations include moves (frame fetches and stores), jumps, forks and joins. *Miscops* instructions consist of control flow, data conversion and control register operations. The remaining cycles on Monsoon may be classified into 4 categories: *idle cycles*, *bubbles*, *second-phase operations* and *recirculate cycles*.

Idle cycles occur when the first stage of the pipeline does not get a token to execute. They are caused either by empty token queues or by the hardware hazard that prevents a token from entering the pipeline.

Bubbles, as described earlier in Section 2.3.1, are caused by the arrival of the first token of a dyadic operation. The instruction cannot execute until the second token arrives, so the first value is stored into the frame and the rest of the pipeline does nothing for this particular token.

Second-phase cycles are due to the handling of read and write requests for the I-structures stored in a Monsoon processor's frame memory. Accesses to these objects cause the second-phase of the instruction also to be executed on the Monsoon processor. Normally the second phase executes on the I-structure board.

Recirculate cycles are due to the spin-lock instructions. The cycle in which the instruction acquires the lock is counted as a *fetch* instruction, but all the other cycles are counted as *recirculate* cycles. These cycles, which are devoted to "busy waiting", are like idle cycles.

2.4 Compiling Id for Monsoon

The Id compiler for Monsoon was created by retargetting the TTDA Id compiler [39, 6] to generate Monsoon object code. Like the TTDA code, each procedure and loop is compiled into a separate dataflow graph; instances of procedures are connected together at run-time by *dynamic* dataflow arcs according to some calling convention. The extra step in the Monsoon compiler is that of assigning a frame for each procedure. A frame is a contiguous block of memory and has the matching locations used by tokens of a particular procedure invocation. (For people familiar with TTDA, a frame serves the same function as a private waiting-matching store for a procedure invocation.) A frame is also used for storing constants and loop invariants.[3]

Code executing on Monsoon is divided into code-blocks corresponding to procedure bodies or loop iterations. The compiler may also split a procedure into several smaller code-blocks to reduce code-block size or "inline" procedures to create larger code-blocks. All tokens belonging to the same code-block invocation have the same frame pointer, which is the base address of the block. A frame always resides in the local memory of a single processor, and thus, all instructions belonging to a code-block invocation are executed on the PE where the frame is allocated.

The code generator for Monsoon has to satisfy a number of constraints imposed by the hardware. For example, instruction fanout is limited to two in most cases and one in the other cases. The compiler must add *fanout* or *fork* instructions to explicitly distribute a value if too many instructions require the output of an instruction. The instruction encoding also constrains the location in instruction memory of a destination instruction relative to its source instruction. A destination instruction can be at most 512 instructions away from its source instruction. For instructions that allow two destinations, one of the destinations is further constrained to reside at the instruction address immediately following the source instruction. In some cases, the compiler must insert *jump* instructions to generate correct code for large procedures or loop bodies. The compiler also splits large code-blocks in order to satisfy this constraint.

Although the Id compiler generates dataflow-style code for Monsoon, it is also possible to write threaded-style code for Monsoon. This can cut down the synchronization cost by eliminating some bubbles. The frame manager, for example, makes heavy use of threaded-style coding. Threaded-code generation requires more analysis and can also make use of Monsoon's temporary registers. However, Monsoon's instruction set was optimized for TTDA style code and is not optimal for the threaded style. For a discussion of threaded execution model on Monsoon, see [31].

2.5 Id Run-Time System and Resource Management for Monsoon

The run-time system (RTS) for Id consists of a frame manager and a heap manager. The frame manager allocates and deallocates the activation frames in which code-blocks are executed, and the heap manager allocates and deallocates dynamic storage for tuples, arrays,

[3]The compiler does not *allocate* frames, it only decides on the structure of the frame template for a procedure. The frames are allocated at runtime.

and other aggregate objects. The Id RTS for Monsoon was written from scratch, and involved considerable research and experimentation. On the TTDA emulator, GITA, the Id RTS was faked by passing all the resource management calls to the underlying Lisp system. The issues of parallelism, efficiency, contention, the length of the critical section, etc. were not addressed in the design of the Id RTS for TTDA. In the following we describe the current resource managers on Monsoon. Some more discussion of these may be found in Section 4.2.

2.5.1 Frame Management

When a procedure P calls a child procedure C, the parent procedure first executes a *get-context* instruction. The get-context instruction traps to an exception handler which allocates a frame, possibly on another processor and returns a continuation consisting of the instruction pointer (IP) of procedure C's entry point, and the node and base pointer of the newly allocated frame [12]. We sometimes refer to this continuation as a *context*. The frame manager chooses the processor on which to allocate the frame, so that the distribution of the work load does not have to be specified in the compiled code. The frame manager also colors the child's context for statistics gathering before returning it to the parent procedure. Procedure invocation then continues with the parent procedure sending the required arguments to its child procedure using the context. Each argument is directed to a specific instruction by some preset procedure calling convention.

In order to minimize external fragmentation, only a limited number of frame sizes are provided. For each frame requested, the smallest available frame large enough to satisfy the request is returned. The current frame manager has sixteen different frame sizes, each a multiple of 64, and employs a version of the quick-fit algorithm [40]. Each quick-list contains free frames of a specific size, linked together through the first word of each frame.

Besides the storage allocation algorithm, a crucial unknown parameter in the early stages of the frame manager development was the number of managers needed for good performance. Should we have one frame manager for the whole machine, or one for each processor, or have eight (one for each pipeline beat) per processor? More resource managers reduce the contention between frame requests but may suffer from fragmentation of resources. It is possible for a resource manager to request resources from another when it runs out of resources, but this may increase the overhead of processing each request.

After some experimentation, we settled on a single set of quick-lists per processor. Since Monsoon has eight-way interleaved pipeline, up to eight independent frame requests per processor can be active at the same time, and thus contention is possible. Chiou investigated frame managers that have eight sets of resources per processor, and found that, although these frame managers rarely waited for a critical resource, they had difficulty sharing resources between the different sets [12].

For maximum efficiency, the frame allocator is written in threaded-style in Monsoon assembly language, and uses some instructions written specifically for the frame manager.

Work Distribution: The frame manager is also responsible for partitioning work among the processors on a code-block granularity. Currently, the frame manager distributes work in a round-robin fashion. Each processor has a set of round-robin counters, one for each frame size, that it uses to distribute work to other processors. Since each processor has its own round-robin counters, work distribution decisions can be made locally. Such a distributed approach to load balancing does not ensure a globally even distribution of work. However, because many frames are allocated during the execution of most programs and the amount of work done in each frame is relatively small, we feel that a round-robin scheme will balance the load reasonably well. Our results so far have supported this intuition.

2.5.2 Heap Management

Requests for heap memory, like requests for frame memory, are made by special instructions which trap to run-time system procedures. These run-time system procedures are written in Id. The same heap manager code runs on the various configurations of Monsoon.

The Id run-time system places all aggregate object storage (such as arrays and tuples) on the I-structure processors. This storage is interleaved so that adjacent logical addresses are actually on different I-structure processors. Interleaving is used to reduce the contention on individual I-structure processors and hence the average latency of access. As pointed out in Section 2.3.1, I-structures can also be allocated in a Monsoon processor's local memory. These local I-structures, however, are not interleaved.

Each processor manages its own partition of I-structure storage in order to reduce RTS contention and to improve throughput. The current heap allocator uses the quick-fit algorithm to manage each partition, and keeps 15 quick-lists for object sizes of 2 through 16 words. Larger objects are managed by the first-fit algorithm. Detailed analyses of various heap allocation algorithms are presented in Iyengar's thesis [21].

I-structure objects have presence-bits associated with each word. These presence bits are initially *empty* on all words of an object, and are set to *full* by I-Store operations. In our implementation, the presence-bits of a piece of storage are cleared when that storage is first cut off of the *tail* (pool of unused memory), and then cleared again whenever the storage is returned to the heap manager for reuse. The heap manager keeps around storage that has been cut from the tail rather than returning it to the tail. Presence bits are cleared upon deallocation in order to minimize the latency of subsequent allocation requests.

Even though each processor has a heap manager, if too many requests to allocate or release heap objects come bunched-up together in time, the heap manager can stil become a temporary bottleneck. We will see some evidence of this in the statistics presented later.

2.5.3 Parallelism Control and Resource Management

Our eventual goal is to have both parallelism and storage management be implicit in Id. Such an Id program will contain no user annotations to direct where to exploit parallelism or how to manage storage. Unfortunately, our current system has not reached this goal yet, and the Id compiler and RTS are not able to run a program efficiently without some user help. To deal with the current situation, we have added two kinds of *annotations* to programs to control the amount of parallelism and the amount of resources used by them. The first annotation, *loop bounds*, which were briefly introduced in Section 2.1, controls how many iterations of each loop are executed in parallel. The second annotation causes heap storage to be reclaimed when it is no longer needed. This storage annotation is not discussed in this paper; the interested reader is referred to [17].

Currently, the user must specify loops as either *sequential, bounded, or unbounded.* The compiler generates specific code for each kind of loop. A sequential loop executes one iteration at a time and uses only one frame. A bounded loop executes a specified number (k) of iterations in parallel, and uses k frames. k can be computed at run time but must be greater than or equal to 2. An unbounded loop is compiled into a recursive call so all loop iterations can potentially execute in parallel. The sequential loop schema incurs less overhead than the bounded loop, which has to allocate and initialize more frames. However, it also exploits less parallelism than a bounded loop. It is usually used for the inner-most loops where it is more important to achieve smaller instruction counts than parallelism. Parallelism is, instead, provided by the outer loops.

Each iteration of a loop that is executing in parallel consumes a frame and, perhaps, some heap storage. By controlling the number of iterations executing in parallel, one can control how much storage is used. Generally, the number of iterations executing in parallel must be large enough to keep the machine busy but small enough not to exceed the storage available on the machine. Culler [14] presents some heuristics for selecting loop bounds.

Categorizing loops as sequential, bounded or unbounded and determining how many bounded loop iterations to execute in parallel is currently perhaps the only tedious part of achieving good performance on Monsoon. It is partly due to the relatively small size of Monsoon's frame memory. Some prior research has been done in this area, notably by Culler [14]. We had not automated this process as we needed more experience to see what impacts such choices have on an actual system (Culler's work was done on GITA, a simulator for TTDA). We plan to take the results of our studies and of Culler [14], and automate this process through a combination of compiler analysis and run-time mechanisms. We believe that this should get us good performance in most cases without user intervention.

3 The Benchmarks

We studied four benchmarks on Monsoon: Matrix-Multiply, Gamteb, Simple, and Paraffins. These benchmarks are described in this section. Members of the Computation Structures Group (CSG) and collaborators of CSG have written other applications as well. 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. Another is the Monte

Carlo Neutron Photon (MCNP) application being written by our collaborators at Los Alamos National Laboratory.

All Id programs described in this document were written in Id90.1 with some annotations for storage deallocation [25]. These programs are available from the authors upon request. Our Matrix-Multiply benchmark was written in Id, C, and Fortran by experts in each of these languages. Our two large benchmark programs, Simple and Gamteb, were originally written in Fortran. These programs were ported to Id by programmers familiar with both Id and the applications. Paraffins was originally written in Id and was then ported to C by an expert.

The following sections describe our benchmarks in great detail. Readers who are not interested in the details can skip to Section 4.

3.1 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 contain double-precision floating point numbers. The Id version of Matrix-Multiply is written as a straightforward triply-nested loop. In early versions of this benchmark, run before September 1991, the innermost loops of the matrix creation, multiplication, and summation routines were all unfolded ten times by the compiler to ameliorate the overhead of loop iteration. Compiler improvements since then have reduced the overhead of loop iteration by a large amount. We scaled unfolding down to four times as further unfolding increases code and frame sizes without providing much improvement in performance.

The Matrix-Multiply program is invoked by supplying a matrix size n and several loop bounds. Our Matrix-Multiply runs are of size 500 by 500. 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 to compare the performance of Id with the performance of C. The Id code implementing Matrix-Multiply is about 130 lines long, including comments. The corresponding C code is 152 lines long.

We also have a version of this benchmark written in both C and Id in a 4×4 blocked style. This blocked version simultaneously computes the value of 16 elements that form a 4×4 block in the final matrix. The net effect is to reduce the number of fetches that we perform on the two source matrices. Thus, each time we execute the innermost loop, we fetch 8 matrix elements, 4 from each source matrix, and use them to update 16 elements of the product matrix. The unblocked version needs 32 fetches to support the same computation. Blocking in this manner reduces the number of fetches by a factor of four. Although the numbers reported here for Id are obtained by changes to the source code, work is underway to have the compiler automatically perform this transformation as an optimization.

3.2 Gamteb

Gamteb [10] 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 the right-most cells. Each particle can be simulated in parallel. Gamteb was written by researchers from Los Alamos National Laboratories as a standard supercomputer benchmark derived from MCNP, a real application.

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.

The simulation considers n particles independently, where typical problem sizes are forty thousand to several million particles. Particles enter the simulation through the front surface and may exit the simulation in one of four ways: *escaping* through the cylindrical surface, *backscattering* 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, plus counts of particles that underwent various processes.

This program is storage intensive. 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 storage reclamation pragmas which direct the compiler to insert heap deallocation calls.

Our Gamteb runs are of forty thousand particles, which is a small but standard benchmark size. Real program runs would be in the millions or tens of millions of particles. The Id code implementing Gamteb is about 750 lines long, including comments. The Fortran code is 720 lines long.

3.3 Simple

This application is a hydrodynamics and heat conduction simulation program known as the Simple code [13]. The Simple document, along with the associated Fortran program, was developed as a benchmark to evaluate various high performance machines and compilers. Although 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 semi-circular 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 are called *nodes*. In the Lagrangian formulation, the nodes are mapped onto a two-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, $(k_{max} - k_{min} + 1) \times (l_{max} - l_{min} + 1)$, 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 following nine quantities based on the values of these quantities in the previous time step: the velocity and position of each node; and the area, volume, density, pressure, artificial viscosity, energy and temperature of each zone. 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 100×100. The Id and Fortran codes implementing Simple are about 1000 and 2400 lines long, respectively.

3.4 Paraffins

The Paraffins benchmark [4] 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 $n > 0$. A paraffin is essentially an unrooted 4-ary tree. Thus the problem of generating distinct paraffins is the same as the problem of detecting isomorphism in labeled free trees. The number of paraffins grows exponentially with n.

Paraffins is an example of a non-numeric program. 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. This came up to a total of 3807508 paraffins. The Id code implementing Paraffins is about 300 lines long. The C code is 370 lines long.

4 Monsoon Software Improvements

The software running on Monsoon has improved greatly over the lifetime of the project, as shown in Table I. The improvement in the benchmarks' run times is entirely due to improvements in the code generated by the compiler, and the implementation of the RTS (although RTS improvements mostly affected multiprocessor times.) The numbers were produced on a single processor Monsoon system.

The following sections describe compiler and run-time system improvements. The casual reader may skip to Section 5 for the speedup results.

Table I: Single Processor Monsoon Performance

Program	Feb, '91 (hr:min:sec)	Aug, '91 (hr:min:sec)	Mar, '92 (hr:min:sec)	Jun, '92 (hr:min:sec)
Matrix-Multiply (500 $\times$ 500)	4:04	3:58	3:55	2:57
4 $\times$ 4 Blocked-MM (500 $\times$ 500)	–	–	–	1:46
Gamteb-9c (40,000 particles)	17:13	10:42	5:36	–
(1,000,000 particles)	7:13:20	4:17:14	2:36:00	2:22:00
Simple (100 $\times$ 100, 1 iteration)	0:19	0:15	0:10	0:06
(100 $\times$ 100, 1000 iterations)	4:48:00	–	–	1:19:49
Paraffins (n=19)	0:50	0:31	–	0:02.4
Paraffins (n=22)	–	–	–	0:32.2

4.1 Compiler Improvements

Over the past year, we made a large effort to improve the Id compiler. In addition to general improvements to the quality of the back end, we worked on a few key areas.

Reducing Heap Allocations: Tuples are often used in Id to package multiple values to return to the calling procedure. Such tuples are allocated on the heap, though frequently, they are just initialized by the callee and immediately read and then discarded by the calling procedure. We can avoid allocating heap objects in such cases by passing each of the component values individually to the calling procedure. In this way, relatively expensive heap allocation and deallocation operations are avoided. While it may seem that passing many values explicitly is expensive, the original code which allocates and uses a new structure incurs even more overhead. This optimization, which is performed automatically, provided a large part of the improvement in the performance of Simple and Gamteb seen between the August, '91 and March, '92 columns of Table I.

Improving the Sequential Loop Schema: Sequential loop efficiency has a great impact on the overall performance of a program because most inner loops are sequential. Although the iterations of a sequential loop are serialized — each iteration must terminate before the following one may begin — the body of each iteration actually executes in parallel. Therefore, we must execute synchronization code at the boundary of each iteration to guarantee that all the threads executing in each iteration have terminated before any thread in the next iteration begins execution. This synchronization code is quite tricky to write. It can also be quite expensive in terms of execution time, if the compiler does not perform crucial analysis and optimization. Interested readers are referred to [1] for more details.

We discovered in early 1991 that our implementation of sequential loops had a relatively large overhead when the loop body was small and there was a fair number of "nextified" variables. Matrix-Multiply, for example, had an innermost loop that took 34 cycles per iteration, more than half of which were spent ensuring that the sequential loops executed sequentially. Out of the 34 cycles, 9 were bubbles, 5 were fanouts, 5 were gates and joins, 2 were switches. There were only 2 Floating point operations and two fetches from the heap! Our first fix was to unroll the loop ten times which increases the useful work done in each iteration of the unrolled loop while incurring the same overhead as one iteration of the original loop. The overhead is thus amortized over ten iterations. When unrolled this way, Matrix-Multiply took an amortized 19 cycles per iteration. This run-time statistic suggested that the original code incurred a 16 cycle overhead during each iteration.

Unrolling the loop reduced the overhead in terms of cycle counts, but also increased the frame and code sizes of the loop. We devised a much more efficient implementation of sequential loops in August, 1991, which reduces the overhead itself without loop unrolling, and hence without increasing the frame size. This new loop schema reduced the cycle counts for the generic Matrix-Multiply to 19 cycles per iteration *without* any loop unrolling. This one had only 3 bubbles, 2 fanouts, 1 join and 1 switch while the rest of the instructions remained the same. A modest loop unrolling of four times further reduced this to 16 cycles per iteration. The overhead of the new loop schema is 4 cycles compared to the original overhead of 16 cycles. The improved sequential loop implementation reduced the running time of both Paraffins and Simple by 6%.

The overhead of sequential loop execution is proportional to the number of variables that are updated each loop iteration. This overhead is significant only when the loop body is small relative to the number of nextified variables. Thus, the new sequential loop schema only shows appreciable improvement if there are small loops in the code. This however does not diminish the importance of having efficient sequential loops as loops with small loop bodies often form the innermost loops of programs.

Lifting Loop Initialization Code: We improved the code generated for loops by adding a compiler transformation module that lifts the initialization code of inner loops in a loop nest as far as possible into the outer loops. Loop initialization code includes the allocation of frames and the storing of loop-constants. This transformation has the greatest impact when there is a large number of loop-constants and, on each call of the inner loop, the number of iterations executed is not large.

Consider, for example, a triply-nested loop where the outer two loops are executed a large number of times and the innermost loop is executed only a few iterations each time it is called. Overall, the initialization of the inner loop becomes significant since it is amortized over very few iterations but is executed many times. Lifting the inner loop initialization out of the outer loops will reduce run time, especially if there are many loop constants. Simple is a program that benefited from this optimization. Lifting the initialization as described above reduced Simple's run time by almost 10%.

Work in Progress: Currently, we are augmenting the compiler to automatically perform *blocking*. The compiler would detect cases where we can reduce the number of fetches in the body of a loop through a combination of loop unrolling and loop merging. The interesting cases are those involving a loop nest, where the fetches are in the innermost loop, but the loops that are to be unfolded are the outer loops. The blocking transformation merges the inner loops after unfolding the outer loops of the loop nest. Matrix-Multiply and some relaxation codes are examples of programs that take advantage of blocking. Our goal is to have the compiler automatically transform the generic Matrix-Multiply code into the blocked version described in Section 3.1. This work has not been completed as of Mar 93.

We have also experimented with allocating I-structure objects with deterministic lifetimes in frames. Doing so requires that the heap object being allocated is small and that the lifetime of the heap object can be determined at compile time. Certain versions of the compiler already have this feature implemented.

4.2 Run-time System Improvements

Our early run-time systems processed memory requests with very low throughput. Since the frame manager is written in a threaded style, the low throughput was caused in part by the eight-way interleaving of the pipeline. As mentioned earlier, the interleaving multiplies the execution time of critical sections by eight since only one instruction in a particular thread is executed every eight cycles. In addition, initial RTS implementations used a single free-list which forced the sequentialization of critical sections. Increasing throughput requires either a reduction in the critical sections or an increase in the number of resource sets or both.

Reducing Critical Section Size: An example of the problems encountered when critical sections were long was discovered when writing this paper. Matrix-Multiply had a large percentage of idles. Most of them, but not all, were due to the hardware hazard. After some investigation, it was discovered that the heap manager had unusually long critical sections making it a bottleneck. We found that it was locking the heap tail pointer, allocating the heap object by incrementing the tail pointer, clearing the presence bits of the newly allocated heap object, *then* unlocking the heap tail pointer. The heap tail pointer could have been unlocked right after it was incremented. After the heap manager was changed to reduce the critical section, the number of idles decreased significantly

Paraffins is another benchmark which revealed the weakness of a low-throughput heap manager. Paraffins was analyzed after the dismal runs in February and August 1991. We discovered that the processor was 50% idle, and spent 30% of the time in heap management, 10% in frame management and only 10% running Paraffins itself. We further discovered by looking at the C code that it hardly generated any garbage, that is, all the allocated storage remained in use through out. Since the heap manager was not running at the necessary throughput, we wrote a faster heap manager that did not handle deallocation of memory. Furthermore we wrote it in assembly code. (We used the assembly code heap manager for Matrix-Multiply as well.)

The assembly-coded heap manager starts with a large contiguous chunk of memory, and cuts memory from one end to satisfy heap allocation requests. It takes only three instructions in its critical section. These three instructions are guaranteed to execute one right after another. In contrast, the critical section of the general heap manager written in Id takes at least tens of instructions. In the worst case, when the general heap manager has to coalesce the free storage into larger blocks, the critical section lasts thousands of instructions. Thus, the throughput of the Id heap manager is often not sufficient for codes that allocate many heap objects.

Splitting Resources among Managers: Splitting resources to create more resource sets can cause poor utilization of available resources. Thus, most of our effort has been directed at reducing critical sections to increase throughput. Nevertheless, some division of resources are made — for instance, there are several free-lists in each set of resources, and each processor has its own set of resources for both the frame and the heap managers. The frame and heap managers communicate with managers on other processors, allowing them to share resources.

5 Scalability and Implicit Parallelism Studies

In this section we study the scalability of Id programs on Monsoon. The same object code is run on all machine configurations from one processor to eight processors. We also use the same run-time system for one processor and multiple-processor systems, even though we could write a more efficient RTS for specific use with single-processor configurations. We estimate that such an RTS would save up to 50% of the frame and heap management overhead compared to the multiprocessor RTS. However, run-time system overhead on Monsoon is usually less than 30%, so total run times would only be reduced by 15%. Since our goal was to write an RTS that would run on any number of processors, we did not think the effort required to write a specialized uniprocessor RTS was worthwhile.

We saw in Section 2.5.3 that currently the programmer must add loop bound annotations to control how much parallelism is exploited in each loop. Optimal performance is tied to the choice of loops to execute in parallel and the loop bounds for those loops. Since our compiler and RTS do not automatically choose loop bounds yet, in order to do this study, we tried a number of loop bounds and chose the ones that performed the best.

For our speedup studies, we ran the four benchmarks on 1, 2, 4, and 8 processor Monsoon configurations. Our speedup results are shown in Table II, which contains the critical path for each program on each configuration. We define the critical path to be the total number of cycles executed by the processor which starts and ends the execution of the program. All other processors will always execute fewer instructions, because they must start after and end before the first processor.

Tables III to VI give the detailed breakdown of the dynamic instruction counts for each of the four benchmarks. Each table shows the total dynamic opcode mix for a single benchmark

Table II: Speedup Results: Id on Monsoon

Configuration / Program	1 PE		2 PE		4 PE		8 PE	
	$\frac{1PE}{1PE}$	$\times 10^6$ critical path	$\frac{1PE}{2PE}$	$\times 10^6$ critical path	$\frac{1PE}{4PE}$	$\times 10^6$ critical path	$\frac{1PE}{8PE}$	$\times 10^6$ critical path
4×4 Blocked-MM 500×500	1	1057	1.99	531	3.90	271	7.74	137
Gamteb-2c 40000 particles	1	590	1.95	303	3.81	155	7.35	80.3
Simple 100 iters, 100×100	1	4681	1.86	2518	3.45	1355	6.27	747
Paraffins n = 22	1	322	1.99	162	3.92	82.2	7.25	44.4

on all four machine configurations. The column for each configuration shows the sum of the statistics, including idles, from all of the processors in the configuration.

5.1 Why Linear Speedup was not Achieved

Ideally, a program exhibiting perfect speedup would execute the same number of total cycles (summed over all processors in the configuration) regardless of the number of processors. However, there is always some overhead associated with exposing more parallelism, so the total number of cycles executed always increases with the number of processors in the configuration. Most of the overhead on multiprocessor configurations of Monsoon shows up as additional idle cycles executed, but some shows up as additional instructions executed. In this section we discuss the causes of these overheads.

5.1.1 Additional Instructions in the Multiprocessor Case

Loop bounds: In order to keep more processors busy, a program needs to expose more parallelism. We increase the loop bounds in order to increase the parallelism exploited by a program. The overhead of a *k-bounded* loop consists of allocating and initializing the iteration frames, and is directly proportional to the loop bound k. Thus, a program on a larger machine configuration requires larger loop bounds and executes more overhead instructions than on a single processor system.

Resource management: As mentioned before, each processor in the system has a set of frame and heap resources and associated managers for these resources. Due to differences in scheduling and available resources, the resource managers on two different machine configurations may process requests at different rates. Furthermore, increasing loop bounds causes more frame requests to be made and so may increase contention for frame managers. This increase in frame manager contention shows up as the increase in the number of tokens recirculated. The number of tokens recirculated indicates the cycles spent waiting for a frame manager lock.

Deferred reads: An I-structure fetch request may arrive at an I-structure element before the corresponding I-structure store completes (see Section 2.2). In the case of an I-fetch before an I-store, a small sequence of instructions are executed to enqueue the deferred requests on a linked-list. When the I-store finally occurs, a few more instructions are executed respond to the deferrred reads by distributing the written value. Whether an I-fetch is deferred or not depends upon scheduling, and hence on machine configuration. We suspect that a multiprocessor system will defer more reads than a single-processor system executing the same program, because it will do more computation in parallel.

5.1.2 Causes of Idle Cycles

Idles tend to be the largest contributor to increased total cycle counts on Monsoon. Idles on Monsoon are caused either by its hardware hazard, or by a lack of work on one or more processors. Multiprocessor Monsoon systems are guaranteed to have more idles than single processor systems due to both of these reasons. Multiprocessor configurations require more network tokens, increasing the number of idles due to Monsoon's hardware hazard. Too little parallelism, which results in lack of work, is much more likely on a multiprocessor since more parallelism is required to keep it busy. Load imbalance, a problem only on multiprocessor configurations, causes idles because some processors will not have enough work to do. These causes are explained in greater detail below.

Hardware Hazard: As explained in Section 2.3.1, Monsoon has a hardware hazard caused by a shared bus between the token queues and the network interface. This hazard prevents any token from being enqueued or dequeued from the network input or output queues while a token is enqueued or dequeued from the user or system token queue, and *vice versa*. Any time there is contention on this shared bus, an idle cycle occurs.

Fetches and *Tag* operations can excite the hardware hazard in two ways. If an instruction produces exactly one token that goes to the network, an idle is forced because that token will be sent to the network while the token fetch from the token queue is delayed one cycle. The fetches executed by the frame manager have local destinations and, therefore, do not cause idles. But the benchmarks are all compiled and our compiler, which produces dataflow-style code, stops a thread after every split-phased instruction. Hence, all compiled fetches and tag instructions excite this hazard.

Once a token arrives at its destination processor, the token may cause an idle by interfering with an operation being performed on the token queue of the destination processor. The hardware implements a fair policy between processor-pipeline and network tokens. When a network token arrives, it is placed in the network input queue. If network input queue contains tokens, and the processor pipeline is kept busy by recirculating tokens, then every other cycle, a processor token is enqueued in the user queue, and a network token is popped from the network input queue. This causes an idle cycle to be placed in the processor pipeline. If the processors are always busy, then the fraction of network tokens that will cause an idle on the destination processor is one half. Thus, we estimate the number of idles due to the hardware hazard to be roughly 1.5 times the sum of the number of fetches and the number of remote send operations, in the worse case (assuming that the processor is always busy.) If the processor is idle due to lack of work, then this ratio will be somewhere between 1.0 and 1.5.

Lack of Work: A processor also executes an idle cycle when its token queues are empty due to lack of work. A processor can have a lack of work because of: (i) load imbalance, (ii) lack of parallelism due to RTS sequentialization, (iii) lack of parallelism during startup and termination of program, or (iv) lack of parallelism in the algorithm.

Load imbalance can cause a lack of work on one or more processors even though there is enough work in the machine to keep all processors busy. The RTS cannot balance load perfectly, because it cannot know which processors need work at a given moment, and which may need work in the future. However, by definition load imbalance cannot occur on single processor configurations.

The run-time system can artificially reduce parallelism by taking a long time to satisfy a frame or heap object request. A processor may need the frame or heap memory in order to continue computation and thus may idle if that object is not returned by a specific time. To see how long a typical RTS request takes, we describe the steps involved in allocating frame memory, giving an instruction count for each step.

Under the best scenario, allocating a frame requires executing 31 or 32 instructions sequentially. It takes 16 instructions to read a frame request, choose a processor, and forward a request to that processor. If a frame is available on that processor's free-lists, it then takes 7 instructions to pop a frame, and another 8 or 9 more instructions to form a context by combining the frame pointer, instruction pointer and statistics color of the called code-block. On Monsoon, the actual latency will be eight times as long because of processor pipeline interleaving.

If no frames are on a free-list, the cost of allocation is much higher, because the presence-bits of a new frame will have to be cleared. Allocating a new frame takes eight instructions plus about $\frac{n}{16}$ instructions to clear out the presence bits of an n word frame. We clear frame presence-bits sixteen words at a time using a block clear instruction. We only clear the presence bits the first time we allocate a frame because procedures automatically clean their own frames before returning them.

The execution of Id on Monsoon starts with the invocation of a top level procedure on one processor which spreads work to other processors. The computation ends in a similar way, but in reverse order. Thus during the *startup and termination phases*, the parallelism ramps up and then ramps down again and there is not enough parallelism to keep all the processors busy. The length of the startup and termination phases are affected by the rate at which the program exposes parallelism, the latency of RTS requests, the latency of interprocessor communication, and the number of processors. Notice this cost is non zero even on a single processor due to the 8-way interleaved pipeline. In our experiments, increasing the number of processors linearly increases the length of the startup and termination phases.

Finally, a lack of parallelism may occur because the *algorithms* used in a program are inherently serial in nature. In these cases, the program must be rewritten to use a more appropriate algorithm. However, we think that all of our benchmarks had sufficient parallelism for an eight processor Monsoon machine.

On Monsoon, it is difficult to differentiate idles caused by a lack of parallelism, load imbalance and the hardware hazard, because neither the sizes of the token queues, nor the number of cycles when they are empty can be measured while running a program. If we had this information, we could determine whether it was lack of work or the hardware hazard that was causing the idles in our programs. By periodically sampling this information on each of the processors, we may be able to distinguish between a temporary load imbalance and the lack of parallelism. For these statistic to make sense, we have the additional requirement that the statistics be collected from all the processors at exactly the same time, something that is impossible to do on Monsoon in particular, and difficult on any (large) asynchronous system in general.

In the following sections, we will try to explain the idle cycles for each of the benchmark runs; the precision of our explanation is limited by the ways in which we can gather statistics.

5.2 Analysis of Benchmarks

5.2.1 Blocked-MM

Our matrix multiplication benchmark running with a problem size of 500 by 500 achieves an excellent speedup of 7.74 on 8 processors. It achieves an *efficiency* of 97%, where we define efficiency to be the speedup divided by the number of processors. The dynamic opcode mix for Blocked-MM is given in Table III, which shows that the total cycle count increases very little from the 1 PE to 8 PE configuration. The more interesting issue here is why there are any idle cycles in the 1 PE case to start with.

On this configuration, about 6% of the total cycles executed are fetches and 9.4% are idle cycles. All of the fetches in the Blocked-MM occur in compiled code, and thus, excite the hardware hazard. We believe the additional 3.5% idles are caused by startup and end costs as well as incoming network tokens. This probably contributes to the slightly rising percentage of idles as we go from the 1PE to the 8PE configuration.

Table III: Blocked-MM Opcode Mix

Configuration Op category	1 PE		2 PE		4 PE		8 PE	
	cycles $\times 10^6$	fraction %	cycles $\times 10^6$	fraction %	cycles $\times 10^6$	fraction %	cycles $\times 10^6$	fraction %
Intop	77.0	7.28	77.0	7.25	77.1	7.11	77.2	7.06
Floatop	250.3	23.66	250.3	23.55	250.3	23.07	250.3	22.90
Fetch	62.8	5.93	62.8	5.91	62.8	5.79	62.9	5.76
Store	1.5	.14	1.5	.14	1.6	.14	1.6	.15
Identity	289.4	27.36	289.4	27.23	290.0	26.73	290.7	26.60
Tag	0.5	.05	0.5	.05	0.7	.06	0.9	.08
Miscop	7.5	.71	7.5	.71	7.7	.71	7.9	.72
Bubble	269.2	25.45	269.2	25.34	269.6	24.85	270.0	24.70
Second Phase	0.1	.01	0.1	.01	0.2	.02	0.4	.04
Recircs	0.0	.00	0.0	.00	0.0	.00	0.0	.00
Idles	99.5	9.41	104.2	9.81	125.1	11.53	131.1	12.00
Total	1057.7	100	1062.5	100	1084.9	100	1093.0	100
Speedup	1.00		1.99		3.90		7.74	
Efficiency	100.0%		99.5%		97.5%		96.8%	

5.2.2 Gamteb

Gamteb is actually an "embarrassingly parallel" program, so we expected it to show excellent speedups on Monsoon. It does so, achieving a 92% efficiency on 8 processors, with a speedup of 7.35.

Gamteb actually executes fewer idle cycles than fetch instructions. We have determined, through code-block coloring, that about two thirds of the fetches are performed by the frame manager and thus do not excite the hardware hazard. For a single processor, therefore, all idles are accounted for. The additional idles for multiprocessors come from the *tag* operations, most of which are used to send arguments to other procedures. The number of procedure calls in Gamteb is proportional to the total number of particles generated, and usually 6 to 7 times the number of initial particles in our data set. With runs that start with 40000 particles, the number of procedure calls, and consequently the number of tag operations, can be very substantial.

When there is only one processor, all of the tag operations send their arguments back to the same processor. When the number of processors p increases, however, the number of remote procedure calls go to approximately $\frac{(p-1)}{p}$ (assuming perfect distribution). Thus, about $\frac{(p-1)}{p}$ of the *Send* operations (a subset of the tag operations) send a token to the network but none back to the same processor, causing a hardware hazard. This explains

Table IV: Gamteb Opcode Mix

Configuration Op category	1 PE		2 PE		4 PE		8 PE	
	cycles $\times 10^6$	fraction %	cycles $\times 10^6$	fraction %	cycles $\times 10^6$	fraction %	cycles $\times 10^6$	fraction %
Intop	47.5	8.05	47.5	7.85	47.5	7.68	47.6	7.40
Floatop	40.4	6.84	40.4	6.67	40.4	6.52	40.4	6.28
Fetch	27.6	4.69	27.6	4.57	27.6	4.46	27.6	4.30
Store	13.3	2.26	13.3	2.20	13.3	2.15	13.4	2.08
Identity	171.6	29.07	171.6	28.36	171.7	27.72	171.8	26.72
Tag	43.6	7.40	43.7	7.21	43.7	7.05	43.7	6.80
Miscop	83.6	14.17	83.7	13.82	83.7	13.51	83.7	13.02
Bubble	119.2	20.21	119.3	19.71	119.3	19.27	119.4	18.57
Second Phase	25.7	4.35	25.7	4.24	25.6	4.14	25.6	3.98
Recircs	2.3	.39	1.8	.29	1.9	.30	1.9	.30
Idles	15.2	2.57	30.7	5.07	44.5	7.19	67.9	10.56
Total	590.1	100	605.3	100	619.2	100	642.9	100
Speedup	1.00		1.95		3.81		7.34	
Efficiency	100.0%		97.5%		95.3%		91.8%	

most of the increase in the number of idle cycles as we move to configurations with more processors.

For eight processors, Gamteb's idles are a little higher (around 6.7 million cycles or about 10.6%) than expected according to this formula. This excess is probably due to the start up cost not being amortized when the problem size is small. This hypothesis is partially confirmed by the fact that the fraction of idle cycles decreased dramatically when we increased the problem size from 40000 particles to 1,000,000 particles. On smaller runs (8 PE Monsoon took only 8 seconds for 40,000 particles), there is also a significant perturbation of the statistics by the skew in the timing of when each processor is started and halted by the the front end processors. Because of this skew, some of the processors may be idling while waiting for tokens from processors that are not yet running.

5.2.3 Simple

We ran Simple for 100 iterations of 100 by 100 grid. The best speedup we have achieved for Simple is a factor of 6.2 on an 8 processor configuration, giving an efficiency of 78%. We believe that this is mainly because of a throughput problem in our current general heap manager. It is also possible that data dependence in the user code (the time step calculation between iterations) contributes to the sequentialization of computation between iterations.

Table V: Simple Opcode Mix

Configuration / Op category	1 PE cycles $\times 10^6$	1 PE fraction %	2 PE cycles $\times 10^6$	2 PE fraction %	4 PE cycles $\times 10^6$	4 PE fraction %	8 PE cycles $\times 10^6$	8 PE fraction %
Intop	591.2	12.63	593.0	11.77	593.2	10.94	600.4	10.05
Floatop	334.8	7.15	334.8	6.65	334.8	6.18	334.8	5.60
Fetch	399.6	8.54	400.0	7.94	399.8	7.38	403.6	6.75
Store	79.2	1.69	80.0	1.59	80.3	1.48	82.6	1.38
Identity	1194.3	25.51	1201.0	23.84	1199.9	22.13	1235.6	20.68
Tag	265.2	5.67	267.9	5.32	268.1	4.95	280.7	4.70
Switch	270.9	5.79	273.4	5.43	272.8	5.03	287.6	4.81
Bubble	937.1	20.02	942.2	18.71	941.3	17.36	967.8	16.20
Second Phase	146.2	3.12	143.3	2.84	142.7	2.63	149.3	2.50
Recircs	12.2	.26	9.4	.19	9.7	.18	10.0	.17
Idles	451.0	9.63	791.8	15.72	1178.4	21.74	1622.2	27.15
Total	4681.7	100	5036.8	100	5421.0	100	5974.6	100
Speedup	1.00		1.86		3.45		6.27	
Efficiency	100.0%		92.9%		86.4%		78.4%	

Together, they result in a lack of parallelism which was observed by looking at the front-panel LEDs on Monsoon which indicate computation and network activities. They dim in a periodic fashion, the same number of times as the number of iterations executed. Table V shows the dynamic operation mixes for Simple running on the four Monsoon configurations.

The total cycles that Simple executes increases with the number of processors. We believe that this is due to the rising number of frame allocations performed on multiprocessor configurations.

5.2.4 Paraffins

Our current Paraffins runs had speedups of 7.25 for eight processors, giving an efficiency of 91%. One through four processor runs of Paraffins actually execute fewer idles than fetches. About half of the fetches were due to critical instructions in the frame manager and half were due to fetches in Paraffins itself, revealing that the idles were due to 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 front-panel LEDs on the Monsoon hardware that indicates processor and pipeline activities. Towards the end of the computation, the work lights on all processors get dimmer

Table VI: Paraffins Opcode Mix

Configuration Op category	1 PE cycles $\times 10^6$	1 PE fraction %	2 PE cycles $\times 10^6$	2 PE fraction %	4 PE cycles $\times 10^6$	4 PE fraction %	8 PE cycles $\times 10^6$	8 PE fraction %
Intop	77.4	24.01	77.4	23.94	77.4	23.54	77.4	21.82
Floatop	0.0	.00	0.0	.00	0.0	.00	0.0	.00
Fetch	15.6	4.85	15.7	4.84	15.7	4.77	15.7	4.42
Store	49.8	15.46	49.8	15.41	49.8	15.16	49.8	14.05
Identity	73.6	22.85	73.7	22.80	73.7	22.43	73.8	20.79
Switch	42.0	13.05	42.1	13.02	42.1	12.81	42.1	11.88
Tag	16.0	4.97	16.1	4.97	16.1	4.90	16.1	4.55
Bubble	33.4	10.36	33.4	10.34	33.4	10.17	33.5	9.43
Second Phase	0.5	.15	0.6	.18	0.6	.19	0.6	.18
Recircs	4.4	1.37	4.5	1.39	4.6	1.40	4.9	1.37
Idles	9.5	2.94	10.0	3.10	15.2	4.62	40.9	11.52
Total	322.3	100	323.3	100	328.8	100	354.8	100
Speedup	1.00		1.99		3.92		7.27	
Efficiency	100.0%		99.7%		98.0%		90.8%	

simultaneously, indicating a lack of work. A larger problem size will probably amortize the tail and increase efficiency. Another possibility would be to change some of the sequential loops to bounded to expose more parallelism.

6 Establishing the Baseline Performance

In this section, we present and discuss experiments comparing the execution efficiency of Id on Monsoon with that of Fortran and C on a MIPS R3000. Each Fortran program was compiled with the DEC Fortran (`f77`) compiler with full optimization (`-O4`), while the C programs were compiled using the stock MIPS `cc` compiler with full optimization (`-O3`).

Since commercial processors do not have hardware support to count the number of instructions a program executes, we used the *pixie* tool for gathering this information on the MIPS R3000 processor. The R3000 executables were preprocessed by the *pixie* program provided by MIPS. The programs were then run on a DEC Station 5000 which contains an R3000 processor, and the resulting data was processed by the program *pixstats*, which is also provided by MIPS.

Table VII include both cycle counts and run times for MIPS and Monsoon runs of each program. On the MIPS processor, cycle counts are produced by pixstats, and run times are the sum of user and system times as measured by the Unix *time* command. When using Unix

Table VII: MIPS R3000/Monsoon critical path comparison

Program	MIPS R3000		1PE Monsoon	
	($\times 10^6$ cycles)	seconds	($\times 10^6$ cycles)	seconds
Matrix-Multiply, 500×500				
double precision	1198	202.3	1768	176.8
single precision	915	153.1	–	–
4×4 Blocked-MM, 500×500				
double precision	954	61.4	1058	105.8
single precision	741	44.9	–	–
Gamteb-2c 40000 particles	265	11.1	590	59.0
Simple, 100 iters, 100×100				
double precision	1787	86.5	4682	468.2
single precision	1745	84.1	–	–
Paraffins n = 22	102	12.0	322	32.2

time, the original binary is run, not the pixie-processed one. On Monsoon, the total cycle count is measured by Monsoon's hardware statistics counters, and run times are calculated by dividing the total cycle count by 10^7, because Monsoon issues instructions at 10 MHz. Both sets of cycle counts are in millions of cycles, while the run times are in seconds.

Our comparison is complicated by the fact that we are comparing two different systems that include different source languages, compilers, RTS's and hardware. Aside from the algorithms, which stay the same, all the other system components are changed at the same time. The rationale of making this study despite the difficulties has been discussed in Section 1. In the next section, we will describe the architectural differences between Monsoon and the MIPS R3000 and motivate the use of cycle counts as the basis for comparison. Section 6.2 describes the differences in run-time system overheads between Id and C/Fortran. In Section 6.3, the remaining overhead is examined and is shown to come mostly from asynchronous fine grain parallel execution.

6.1 Dynamic Cycle Counts as the Basis of Comparison

We want to compare architectures, not implementations. Consequently, we need to factor out the differences in performance that are artifacts of the implementations. The biggest difference between the implementations is the clock speed. Monsoon, as mentioned before, runs at 10MHz, while the MIPS R3000 that we used runs at 25MHz. The two machines have different cycle times because they are implemented using different technologies — Monsoon is a board-level design and the MIPS R3000 is a VLSI chip set. We believe, however, that we could build a Monsoon with clock speeds comparable to the MIPS processor given the design and manufacturing resources used to build the MIPS processor. We account for the different cycle times by collecting statistics in terms of number of cycles executed in addition to absolute execution times.

Of course there are obvious and unadjustable differences between the respective instruction set architectures (ISA). Monsoon's ISA is good at synchronization but poor at threaded performance, while the MIPS's ISA is good at threaded performance but poor at synchronization. The MIPS, in executing imperative languages, uses long threads and almost never synchronizes. Monsoon, on the other hand, runs mostly short threads and synchronizes frequently in executing compiled Id code. Thus, the ISA of two processors is optimized for different purposes. We will discuss these differences in greater depth in later sections.

The next few paragraphs explain the architectural differences. We compare the pipeline stages of Monsoon and MIPS and suggest ways to implement the Monsoon processor pipeline stages so that each stage is no more complex than a MIPS pipeline stage. We also discuss memory system differences and word size differences.

Making a Monsoon with a fast clock: Comparing cycle counts is fair only if the complexity of a stage of the Monsoon pipeline is similar to that of a stage of the MIPS R3000's pipeline. Looking at Monsoon's pipeline, every stage save the presence bit stage is similar to a MIPS R3000 pipeline stage. By changing the presence bit stage into two stages, we can reduce its complexity to that of a normal RISC pipeline stage. Splitting the presence bit stage into two stages can be done using a dual-ported cache memory for the presence bits, one read port and one write port. The first stage of the split stage will read the presence bits while the second stage of the split stage will write back modified presence bits.

This two-ported cache memory is not that complex — the size of the memory will increase by no more than 4 times (2 times for the row lines, 2 times for the column lines) and should run at the same speed, especially if we take advantage of the fact that we are always writing to the location we read the cycle before. Also, since presence bits require only 3 bits per word, less than 100 kilobytes of this memory is necessary. It is important to remember that "unfamiliar" does not necessarily mean "more complex". Thus, our processor's pipeline stages are of reasonable complexity compared to the MIPS R3000. If compared to modern superscalar pipelines with extensive hazard detection, our processor is considerably simpler.

Memory System Differences: The memory systems on the two machines are very different — Monsoon has no virtual memory and uses static RAM for local memory. It, therefore, has an advantage in that it does not incur cache miss or page fault overheads. Monsoon's memory, however, is very small (2 megabytes) which is smaller than the first-level caches in some modern RISC processors. Thus, Monsoon's lack of a cache does not make it an unrealistic machine — in reality, Monsoon can be thought of as having a compiler-managed cache, the frame store, in which compiled code explicitly reads and writes its long latency main memory, the I-store. In fact, our fetches from I-structure memory take approximately 26 cycles in the best case, making fetches to main memory *worse* than most conventional RISC machines. This method of split-phase transactions and using parallelism to hide memory latency is actually extremely effective. For example, even though the cycle count of Matrix Multiply on the R3000 is less than the cycle count on Monsoon and the MIPS clock is 2.5 times faster, the MIPS run time is actually longer than the Monsoon run time.

Fortunately, pixie-generated statistics do not include cache miss or page fault overheads, allowing us to completely sidestep this issue. Memory system support for parallel processing is a very important issue but is not the subject of this comparison study. Adding caches to Monsoon, however, is straightforward since our frames are totally local and our heap objects are write-once.

Word Size Differences: The final architectural difference is that Monsoon is a 64-bit architecture and the MIPS R3000 is a 32-bit architecture. This difference shows up most clearly as the difference between the execution times for 32-bit and 64-bit floating point arithmetic. The MIPS R3000 floating point processor performs 64-bit arithmetic at almost the same speed as 32-bit arithmetic, but it requires twice as many cycles to load 64-bit floating point values. To quantify the effect of this architectural difference, we ran Matrix-Multiply, Blocked-MM and Simple on the MIPS R3000 with 32-bit floating point. We were unable to run Gamteb with 32-bit floating point numbers. The resulting execution times are reported in Table VII. Paraffins does not use floating-point values and is, therefore unaffected by this difference.

Having accounted for the discrepancies due to implementation differences, we can now start the more interesting part of the comparison. Overall, Table VII shows that Id programs on Monsoon take about three times as many cycles to execute as corresponding C or Fortran code on the MIPS R3000. While three times as long may seem very large, the same code supports parallel execution on multiple processors with almost perfect speedup, as reported in Section 5. Nevertheless, it is important that we understand the source of this overhead and identify means of improvement.

One immediately apparent source of overhead is the Id run-time system. Run-time systems supporting parallel computation are by necessity more complicated and more expensive to run than those supporting sequential computation. This overhead is quantified and compared in the next section. The remaining overhead comes from the style of parallel execution adopted in Monsoon. This is examined in Section 6.3.

6.2 Accounting for the Differences due to the Id/Fortran-C Run-time Systems

The fine grain parallel execution of Id programs demands a more sophisticated RTS than is found in the implementations of sequential languages such as C or Fortran running on

Table VIII: Frame and heap management overheads on 1 PE Monsoon and cost of similar operations on the R3000.

Program		Frame Management $\times 10^6$ cycles	Frame Management % of total cycles	Heap Management $\times 10^6$ cycles	Heap Management % of total cycles	Total RTS $\times 10^6$ cycles	Total RTS % of total cycles
4×4 Blocked-MM	Id	–	–	–	–	1.03	0.10%
	C	.006	0.00%	0.13	0.02%	0.136	0.02%
Gamteb-2c	Id	104.1	17.61%	78.0	13.20%	182.1	30.85%
	F77	1.63	0.62%	0.71	0.26%	2.34	0.88%
Simple	Id	557.1	11.90%	198.4	4.23%	755.5	16.32%
	F77	9.45	0.54%	0	0%	9.45	0.54%
Paraffins	Id	–	–	–	–	165.8	51.45%
	C	7.62	7.47%	49.5	48.5%	57.12	56.0%

uniprocessors. The RTS overheads incurred by Id on Monsoon will probably be incurred in other general purpose parallel processing systems as well.

Id activation frames cannot be stack-allocated because more than one child of a given procedure may be executing at once, and these children may terminate in an order unrelated to the order in which they were invoked. The storage management problem for activation frames of arbitrary size that may be allocated and deallocated in arbitrary order is essentially the same as the heap management problem and is much more expensive than the stack allocation of activation frames used for sequential languages.

Another difference between the Id and Fortran run-time systems is that Id programs dynamically allocate memory from the heap, while Fortran programs use statically allocated storage. The Fortran version of Gamteb explicitly stack-manages a large block of statically allocated storage. Part of the dynamic heap allocation and deallocation overhead incurred by Id programs is in clearing the presence-bits associated with each word of the storage. This cost is proportional to the size of the object.

Table VIII shows the percentage of run time spent on frame and heap management for each of the benchmarks. For Paraffins and Blocked-MM, the frame and heap management overheads are combined because we used a heap manager coded in Monsoon assembly code, which prevented us from gathering these statistics separately. Precise determination of the cost of frame management for C and Fortran is difficult because these are tightly integrated into user's code. We estimate this by noting the number of procedure calls, and assume that frame management for each call costs two cycles. (One to increment the stack pointer at the beginning of the call, and another to decrement it at the end of the call. We assume that stack overflow is taken care of by the underlying virtual memory system.) Heap management is virtually non existent in the Fortran version of Simple as heap objects are allocated on the stack. "Heap" space needs to be allocated for the particles in Gamteb. We estimate

Table IX: Dynamic Cycle Counts Excluding RTS code.

Program	Monsoon $\times 10^6$ cycles	MIPS R3000 $\times 10^6$ cycles
Blocked-MM 500×500	1057	741
Gamteb-2c (40,000)	408	263
Simple (100×100, 100)	3926	1735
Paraffins (n = 22)	156	45

the heap management overhead for the Fortran Gamteb by multiplying the total number of particles generated by 4 (load pointer, increment pointer, test pointer for overflow, store back new pointer.)

The figures in Table VIII show that RTS overhead is insignificant for Blocked-MM for both the Id and C code. This is to be expected as the program allocates only three matrices, and makes a small number of procedure calls. The RTS overhead for Gamteb-2c and Simple are however significant for the Id code, but negligible for the Fortran code. In particular, the Id version of Gamteb-2c spends about 31% of its cycles in the RTS.

Paraffins, on the other hand, is intensive in its usage of heap memory. This is reflected in the statistics. The C code performs heap storage management by calling an optimized routine written by the programmer. Both the Id and C allocators use the same heap management algorithm: start with a large chunk of memory and allocate storage by incrementing a pointer indicating start of the free region. The only difference is that the Id allocator must clear the presence-bits on the allocated storage, at a cost proportional to the amount of storage allocated, in addition to performing standard heap management tasks. Because of this and Monsoon's accumulator style instruction set, the Id code ends up executing about 3 times as many cycles as the C code to perform all RTS functions.

While RTS differences can be significant at times, they do not account for all of the cycle differences. To continue our investigation, we factor out the RTS cost from the run time statistics. The result is shown in Table IX. It shows that even if we discount the frame and heap management overheads, Id code running on Monsoon still executes more cycles than C or Fortran code on the MIPS system. In the next section, we explain where the remaining overhead is coming from.

6.3 The Cost of Asynchronous Parallel Execution

Asynchronous parallel execution is generally less efficient than sequential execution in terms of the total amount of work that has to be done. There is a considerable amount of overhead involved with running in an asynchronous fashion on a synchronous machine. It is important, however, to understand the overhead of our execution model, what overhead is avoidable and what is not. If the overhead is a small constant and if we can get good speedups from the programs we write, executing asynchronously would be worth it.

We examine the specific costs caused by asynchronous parallel execution[2]. These overheads are incurred by forks and joins used to spawn and synchronize tasks, tags used to send arguments to threads, bubbles which are a synchronization cost, running loops in parallel, termination detection, conditionals, and structure handling. These overheads, though reducible, are not completely removable without change to the execution style. We have not worked to reduce these overheads because of manpower reasons. In the following we describe some of these costs.

Forks and Joins: These instructions explicitly split and combine computation and are the overhead of spawning tasks and rejoining tasks. *Fork* instructions spawn new threads within the same frame and consequently, on the same processor, while *join* instructions can join data arriving from any processor. When code executes in parallel towards a common goal, work must be spawned to keep processors busy and results must be joined to share the computation[4]. Spawning and joining threads is necessary to run a program in parallel but completely unnecessary to run one sequentially since there is only one thread for the whole computation and operations produce results in the order in which they are specified in the instruction stream[5].

Tag: The cost of spawning threads depends upon whether we are spawning to another processor or to the local processor. In the former case, send instructions are needed for sending arguments and results between the cooperating processors and are not required in conventional sequential execution. Though one may argue that sequential procedure calling conventions may require some data movement akin to tag operations, tag operations involve movement of data across the network and are thus more expensive.

Bubbles: These represent the cost of synchronization in collecting arguments to start a thread. Our compiler performs most dyadic operations in a "dataflow" style which produce a bubble for each useful operation. Dyadic instructions are heavily used throughout our compiled code and include arithmetic instructions, switches to control dataflow through a conditional, loop protocols, and producer/consumer parallelism using I-structures and deferred reads. The code our compiler generates allows computation to progress as much as possible given the available data, but requires very frequent synchronization, incurring bubbles in the process. Using threaded code will reduce the number of bubbles, but not eliminate them since there will still be synchronization when entering a thread.

[4]Though SIMD or data parallel machines do not do "spawn" tasks they must perform barrier synchronizations which are also expensive.

[5]Modern RISC processors can produce results out of order, but use complicated hardware mechanisms to insure that it *seems* like everything is executing in order.

Parallel loops: Such loops can execute each iteration of the loop on a potentially different processor. This requires sending data from one iteration of the loop to the next over the network. Like sending arguments to a thread spawned on another processor, this is expensive not only in terms of having to send the arguments and results, but also in synchronizing the data. We also use sequential loops in our programs, particularly in the innermost loops. As mentioned in Section 4.1, asynchronous execution within each loop iteration incurs the overhead of synchronization at the end of each iteration.

Termination signals: Termination signals are needed in an Id implementation to indicate that all side-effecting operations in a basic block have completed, so that frame or heap objects may be reclaimed. Termination detection for n threads is achieved through an n way join, at a cost proportional to n. Termination detection costs are not incurred in normal sequential languages because we know that a side-effect is completed by the execution of the instruction sequence. Termination signals are different from joins in that only the presence of the tokens are used, not the values carried by the tokens.

Conditionals: Conditionals in data-driven execution must direct data to the code of the correct arm of the conditional, and require a *switch* instruction for every free variable that is needed in the conditional. A *switch* instruction takes two inputs, the value to be switched and a predicate, and has two possible output destinations corresponding to the "then" and "else" parts of the conditional. The number of switch instructions in a conditional is equal to the number of free values used in the conditional. A conventional implementation of a conditional, on the other hand, executes a single branch instruction regardless of the number of free variables in the conditional. All the free variables are carried either on the stack or in registers. The overhead of using *switch* instructions is significant in the heap manager written in Id. In addition, part of the overhead of loops in Id on Monsoon is the *switch* instructions used to circulate values from one iteration to the next.

Split-phase heap requests: These also cost more than normal memory reads. At least two instructions are executed to make a I-structure request — one instruction to make the request and another to actually do the read/write. Even more instructions are executed if the read is deferred.

Fine Grain Synchronization with Presence bits: While the use of presence bits help fine grain synchronization, particularly between producer and consumer, it incurs the cost of clearing the presence bits before a heap object is allocated. This cost can be significant as it is proportional to the size of the object.

Our compiler produces code to expose maximum parallelism. It would be more efficient to expand the parallelism just enough so that the cost of exploiting the exposed parallelism is no more than the gain obtainable from running in parallel — however, achieving this precise balance is very difficult. We only allow dynamic control over parallelism via loop bounds, and but otherwise pay the parallelism overhead throughout the compiled code.

By paying the price of asynchronous parallel execution, however, we are able to obtain parallelism from a wide range of programs, particularly programs with complicated control structures. In addition, we can fairly easily simulate code written in a synchronous style with good efficiency[35]. Due to the overheads involved with general asynchronous code, however, we expect Id code, even without RTS cycles, to require more run time than C/Fortran code running the same program. This fact is obvious from Table IX which gives execution cycle times with the RTS cycles removed. Our higher run times are caused by the non-strictness of Id, the parallel code our compiler generates as well as architectural constraints that make certain operations harder to perform on Monsoon.

Next we attempt to quantitatively estimate the cost of parallel execution. We do so by examining the dynamic opcode mixes and analyzing some samples of compiled codes. Though such analyses can be carried out for each benchmark with some degree of accuracy, it is difficult to generalize across benchmarks.

6.3.1 Dynamic Opcode-Distributions

Tables X and XI show the dynamic distribution of instructions on a 1-PE Monsoon and the MIPS R3000 respectively. We show statistics from single-precision runs for the MIPS R3000 where possible. These opcode categories were chosen for a reason but in retrospect, we wish we had categorized the operations differently.

Intop and *Floatop* are the same on both machines. *Fetch* and *Store* instructions on Monsoon are only used for accesses to I-structures, while on the MIPS architecture *Load* and *Store* instructions are used to access both the frame and the heap stores. The instructions to move data in and out of frame memory on Monsoon are included in the *Identity* instructions. *Identity* instructions on Monsoon are however used in other ways too: as *forks* and *joins*, and *jumps* instructions. The *Others* category contains *switches*, data conversion and the SVC instructions used to invoke the RTS. Finally, there are instructions which are "penalties" when we are not able to exploit as much parallelism as the hardware is meant to handle. *Idle* cycles on Monsoon correspond roughly to *Nop* and *Interlock* cycles on the R3000. Interestingly, the R3000 incurs a substantial number of *Nop* and *Interlock* cycles when floating point operations are involved. In 3 out of 4 of our benchmarks 25% of cycles on R3000 are used up in these penalties. We will use the R3000 numbers after subtracting *Nop*s and interlocks as an estimate of base line work on a sequential machine.

Bubbles, *Tag*, *Others* are all operations that are identified as the overheads of asynchronous execution in the previous section. In addition, all of the *identity* instructions except those used as conventional *move* instructions are additional costs of asynchronous execution. The total number of cycles spent in these categories gives us a quantitative indication of the cost of asynchronous execution. Unfortunately, the Monsoon instruction set normally

Table X: Dynamic Opmix of Execution on Monsoon

Program	4 × 4 Blocked-MM		Gamteb-2c		Simple		Paraffins	
Op category	$\times 10^6$ cycles	% of total	$\times 10^6$ cycles	% of total	$\times 10^6$ cycles	% of total	$\times 10^6$ cycles	% of total
Intop	77	7.28%	48	8.05%	591	12.63%	77	24.01%
Floatop	250	23.66%	40	6.84%	335	7.15%	0	0.00%
Fetch	63	5.93%	28	4.68%	400	8.54%	16	4.85%
Store	2	0.14%	13	2.26%	79	1.69%	50	15.46%
Identity	289	27.36%	172	29.06%	1194	25.51%	74	22.85%
Tag	1	0.05%	44	7.39%	265	5.67%	16	4.97%
Others	8	0.72%	112	18.91%	429	9.17%	47	14.56%
Bubble	269	25.45%	119	20.19%	937	20.02%	33	10.36%
Idles	100	9.41%	15	2.62%	451	9.63%	9	2.94%
Total	1058	100%	590	100%	4682	100%	322	100%

Table XI: Dynamic Opmix of Execution on MIPS R3000

Program	4 × 4 Blocked-MM		Gamteb-2c		Simple		Paraffins	
Op category	$\times 10^6$ cycles	% of total	$\times 10^6$ cycles	% of total	$\times 10^6$ cycles	% of total	$\times 10^6$ cycles	% of total
Intop	57	7.65%	25	9.51%	316	18.09%	37	36.33%
Floatop	250	33.77%	60	22.74%	426	24.43%	0	0.00%
Load	150	20.19%	65	24.39%	291	16.70%	21	20.58%
Store	79	10.66%	35	13.21%	131	7.52%	29	27.97%
Branch	9	1.21%	15	5.54%	73	4.16%	8	7.54%
Jump	0	0.00%	0	0.06%	10	0.55%	8	7.44%
Nop	1	0.15%	17	6.59%	118	6.75%	0	0.14%
Interlock	195	26.36%	47	17.95%	381	21.80%	0	0.00%
Total	741	100%	265	100%	1745	100%	102	100%

collects the number of all types of *Identity* instructions, including *moves*, in a single statistics category. Also, as can be seen in Table X, *Identity* instructions account for the largest numbers of cycles executed in almost every program. Thus, in order to accurately measure the costs of asynchronous execution, we must be able to measure the *moves* separately from the other *identities*.

So far, we have done that only for Gamteb-2c. The initial analysis shows that the 29% of total processor cycles spent executing *identities* is subdivided into: (i) 5% dataflow synchronization (*Joins*) overhead; (ii) 15% dataflow-style *Fanouts*, (iii) 3% architecturally-imposed *Jump* instructions, and (iv) 11% conventional *Move* instructions. If we assume that this subdivision of the *identity* cycles in Gamteb-2c is representative, *i.e.*, that approximately 1/3 of all identities are conventional *moves* as opposed to asynchronous execution overhead, the cost of asynchronous execution works out to be 44% for Blocked-MM, 66% for Gamteb-2c, 52% for Simple and 45% for Paraffins. Overall, it looks like about 1/2 of the cycles executed on Monsoon are overheads of asynchronous execution.

The overhead incurred by parallel execution is reasonably high, increasing total run time by up to a factor of 2. The speedups seen, however, seem to indicate that we can about eight times faster for many programs. Thus, the overhead cost is dwarfed by the observed and potential speedups and makes viable the asynchronous parallel execution model which we call dataflow.

One last question that we studied is whether there are ways of lowering the overhead of asynchronous parallel execution while preserving most of its benefits. In particular, we studied the effects of threading on Paraffins and Blocked-MM. Each of these benchmarks has an inner loop which dominates the run time. Our study involves hand-coding the inner loop in a threaded style.

In the case of Paraffins, the compiled Id code for the inner loop took 35 cycles per iteration to execute on Monsoon. The corresponding loop from the C program executed in 14 cycles per iteration on the MIPS processor. The hand-coded Monsoon version, written in threaded style, took 21 cycles per iteration. A threading Id compiler should be able to produce the same code for the inner loop. Comparison of this hand-coded inner loop with our current compiled code shows that the 14 cycles overhead in the compiled code comes from: fanout and join instructions, and sequential loop overhead, all of which are the costs of highly asynchronous execution.

Blocked-MM with hand-coded inner loop ran about 20% faster faster than the compiled version, executing in 814 million cycles. This execution time compares very favorably to that of the C code executing on MIPS listed in Table VII, which is 954 million cycles. Again, we see that threading can decrease the overhead of asynchronous parallel execution. However, the effect of threading on parallelism has not been fully explored. We will be exploring this issue in the future.

7 Discussion and Conclusions

7.1 Monsoon performance

Monsoon has proven itself to be an effective parallel architecture, achieving close to perfect speedups on the applications we ran. Even though the largest Monsoon configuration consists of 8 processors, this configuration actually executes 64 threads in parallel because of the eight-way pipeline interleaving on each PE. Few other machines have had this kind of success in exploiting 64-fold parallelism.

Id also proved to be effective at exposing parallelism inherent in the ordinary algorithms we used. The dataflow style code our compiler generated is pretty efficient, even when compared to MIPS compilers. Generating better code for Monsoon would require generating threaded code which would give us at most 15% to 20% improvements in speed. Since the effort to add threading to our compiler would be substantial and the benefits small, we decided to wait until our next machine to implement threading. When we do generate threaded code, we must consider the impact of long sequential threads on the parallelism available in a program — the scheduling of these long threads, for example, may have adverse effects on the ease of performing load balancing.

Monsoon gave us our first real chance to explore run-time systems. We found that naive frame management and work distribution strategies that our RTS uses to be adequate to achieve seven-fold speedups on eight processors. We also found that oblivious work distribution schemes balanced the load well. Although work distribution is, in general, a difficult issue, a round-robin scheme works well for us because work distribution is done at a very fine granularity, resulting in many distribution decisions.

On the other hand, Monsoon's default instruction set is not geared towards a threaded style of execution, making the writing of the run-time system difficult. Run-time systems have critical sections that are most efficiently written in threaded code. Though there are enough threaded instructions in Monsoon to write a run-time system, a more suitable instruction set would make the system coding much easier. Monsoon also suffers from rather weak addressability. Three-address instructions will greatly improve code efficiency, particularly when executing threaded code. Finally, eight way interleaving of the pipeline has the disadvantage of making critical paths and critical sections longer than they need to be.

Our experiments have led us to the conclusion that instruction level parallelism achieved by explicit fanout and joining is expensive, and would be better exploited by hardware mechanisms. We want to emphasis, however, that superscalar architecture alone will not expose enough parallelism. Monica Lam's work [23] shows this to be the case for a large number of benchmark programs.

On the positive side, our experiments with Monsoon have shown that fine-grained parallelism, split-phased memory accesses, and fast context switching can hide memory latency and synchronization waits when there is sufficient parallelism. The experiments also verify that there is sufficient parallelism in most programs. We reach this conclusion from the fact that all of the benchmarks ran with very few idle cycles aside from *idles* caused by hardware hazards.

The adoption of split-phased operations for global memory access allows computation to overlap memory access latency. This technique, which has been used in Id implementations for many years, achieves the memory-latency toleration goal confirmed to be important by researchers at Stanford [16]. The same paper expressed doubt that hardware techniques for hiding memory-latency are practical because these techniques dynamically reorder instructions within a large instruction window. The use of split-phased transactions, implemented by a combination of software and hardware, shows that memory-latency toleration is achievable without a large investment in hardware.

High network bandwidth and low network latency are crucial to fine grain parallel execution and must be supported by hardware. Overlapping computation with communication and synchronization would be impossible if the processor/network interface was not tightly integrated. Monsoon's tight processor-network integration allows programs to perform frequent communications with very low overhead. Furthermore, Monsoon's pipeline has a stage for formatting messages (tokens), so even at the instruction set level Monsoon overlaps communication and computation.

A single processor Monsoon running Id is still not competitive with a single RISC processor running C or Fortran code. Once we move to multiprocessor systems, however, the overhead we see on a single processor is spread out over many processors. Also, more conventional parallel processors will see much of the overhead that we see since there is always overhead in generating parallelism and sychronizing that parallelism. Monsoon should be very competitive with other parallel systems in terms of performance and ease of use.

7.2 Future Work

One can look at future work in two lights: what we would do if there were no commercial processor market and what we would do since there is a commercial market. The reason for splitting in this fashion is simple. Though we have ideas about what should be done to execute our paradigm efficiently, there is great momentum pushing conventional processor design forward. The amount of resources at our disposal is a very small fraction of what the industry is using to develop their systems. Thus, though we are capable of designing systems from scratch, we would be further ahead in terms of performance and price if we modified conventional architectures.

We will first discuss some possibilities for a machine unconstrained by the forces of the commercial world. We then focus our attention on a more feasible system that leverages conventional hardware by adding novel modifications to a commercial processor.

7.2.1 No commercial market: Dream machines

Our experience with Monsoon showed us that fast sychronization and a fast network are extremely useful to our execution paradigm. Monsoon's interleaving allowed us to exploit parallelism found in single procedures and to tolerate branch latencies — we do not need delay slots after branches. Monsoon's multithreading together with dataflow's ability to expose parallelism allowed us to tolerate latencies incurred when accessing data through the network.

Monsoon's poor single threaded performance, however, hampered us greatly when executing code with critical sections (mostly in the run-time system). The techniques for combining the benefits of multi-threading without sacrificing good single thread performance are already emerging. Two architectures that further advance dataflow processsor design are a dynamically interleaved pipeline[22] or two or more pipelines fused together[28].

Thus, if there were no commercial market, or if we were able to muster as much resources as a commercial project, we would have gone on to build a successor to Monsoon that is faster and larger, and which retains Monsoon's positive features of fast access to the network, while incorporating both multithreading and good single thread performance.

7.2.2 Living in a real world

Modifying an existing high-speed processor to support another paradigm is a way to use existing technology to further our own research. Our next project does just that. Our next machine, called *T [27] (pronounced "Start") and being built in collaboration with Motorola, will use 88110 processing nodes that have a fast network interface incorporated as a functional unit on the chip. While this is not optimal, it is still very fast. *T will also have hardware support for handling continuations. These are similar to token queues on Monsoon but are an improvement in that it allows a certain degree of software control over scheduling.

The architecture of *T was driven in part by observations of Monsoon's architectural drawbacks. *T will have good single thread performance because threads are not interleaved in the processor pipeline. This will, among other things, cut down RTS latencies and RTS critical section bottlenecks. Furthermore, the *T processor will be based on a RISC core so threads will be able to make better use of registers than they could on Monsoon. Plans have been made for a 512 processor *T machine to be build by the end of 1994.

The research potential for *T is enormous. The available of a large machine coupled with the fact that software has control over scheduling allows us to explore scheduling at a level which we have not been able to before. *T will have virtual memory and caches, allowing us to experiment with sophisticated memory systems for parallel processing. With a large system, we will be able to develop and run large real-world applications.

7.2.3 A Challenge:

Although the statistics we have gathered only cover a small number of benchmarks, we believe that the results are extremely promising. We have seen good speedups for most of our programs, and have shown that the amount of overhead to run Id on Monsoon is reasonable. We have also demonstrated that the execution times, measured in cycles, of Id programs are within a small constant factor of those of corresponding C or Fortran programs. Finally, our experiments have shown that a tightly coupled processor and network can support very fine-grained parallelism and hide communication and synchronization latency. We would like to see similar analyses of these applications running on other parallel machines, so that we can compare different approaches to achieving parallelism.

We are continuing our work on both the compiler and the run-time system. More applications, including several SPEC benchmarks have been written and are currently being debugged. We hope to have more performance numbers in the near future for these programs.

Credits: The Monsoon project is the result of a large team effort composed of MIT and Motorola people. Greg Papadopoulos designed Monsoon and managed much of its implementation. Chris Joerg and Andy Boughton designed the network, while Ken Steele designed the I-structure board. Motorola built the hardware. Ken Traub wrote the TTDA Id compiler while at MIT and then became the Monsoon software architect at Motorola. Jamey Hicks became the compiler guru after Ken Traub left and lead the Monsoon software effort at MIT. Andy Shaw wrote the original Monsoon back-end for the Id compiler. Boon Ang improved the performance of compiled code by hacking the middle and back ends of the compiler. Derek Chiou and Arun Iyengar designed and wrote the frame and the heap managers, respectively. Mike Beckerle wrote the MINT simulator and defined the IO software system for Monsoon. R. Paul Johnson implemented the I/O substrate, while Christine Flood implemented the I/O libraries. Christine also wrote the transcendental libraries. Peter DeWolf wrote most of the Id World interface and execution manager. Venkat Natarajan and Maria Carlon worked on statistics and visualization. Maria Carlon wrote the loader. Gamteb was written by Olaf Ludbeck of Los Alamos National Laboratory.

Acknowledgements: We would like to thank Christine Flood, Shail Aditya, Alejandro Caro, Paul Barth, Arun Iyengar and Kyoo-Chan Cho for their invaluable assistance in tuning various programs, in adding storage reclamation annotations, for run-time system and compiler hacking, and for running these programs on GITA and Monsoon.

References

[1] B. S. Ang. Efficient Implementation of Sequential Loops in Dataflow Computation. In *Proceedings of the 6th Conference on Functional Programming and Computer Architecture, Copenhagen, Denmark*, June 1993. (to appear).

[2] Arvind, D. E. Culler, and K. Ekanadham. The Price of Asynchronous Parallelism: An Analysis of Dataflow Architectures. In *Proceedings of CONPAR, Manchester, England*, Sept. 1988.

[3] Arvind, D. E. Culler, and G. K. Maa. Assessing the Benefits of Fine-Grained Parallelism in Dataflow Programs. *The International Journal of Supercomputer Applications*, 2(3):10–36, Nov. 1988.

[4] Arvind, S. K. Heller, and R. S. Nikhil. Programming Generality and Parallel Computers. In *Proceedings of the Fourth International Symposium on Biological and Artificial Intelligence Systems*, Sept. 1988.

[5] Arvind and R. A. Iannucci. Two Fundamental Issues in Multiprocessing. In *Proceedings of 4th International DFVLR Seminar on, "Parallel Processing in Science and Engineering", Bonn, FRG*, volume 295 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1987.

[6] Arvind and R. S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transactions on Computers*, 39(3):300–318, Mar. 1990.

[7] Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: Data Structures for Parallel Computing. *ACM Transaction on Programming Languages and Systems*, 11(4):598–632, Oct. 1989.

[8] P. S. Barth, R. S. Nikhil, and Arvind. M-Structures: Extending a Parallel, Non-strict, Functional Language with State. In *Proceedings of the 5th Conference on Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts*, volume 523 of *Lecture Notes in Computer Science*. Springer-Verlag, Aug. 1991.

[9] W. Blume and R. Eigenmann. Performance Analysis of Parallelizing Compilers on the Perfect Benchmarks Programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(6):643–656, Nov. 92.

[10] 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 of Supercomputing '89, Reno, Nevada*, pages 10–20. ACM Press, Nov. 1989.

[11] D. Cann. Retire Fortran? A Debate Rekindled. *CACM*, 35(8):81–89, Aug. 1992.

[12] D. T. Chiou. Frame Memory Management for the Monsoon Dataflow Processor. Master's thesis, MIT, EECS, Laboratory for Computer Science, Cambridge, MA, Sept. 1992.

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

[14] D. E. Culler. *Managing Parallelism and Resources in Scientific Dataflow Programs.* PhD thesis, MIT, EECS, Laboratory for Computer Science, Cambridge, MA, 1989.

[15] D. E. Culler and G. M. Papadopoulos. The Explicit Token Store. *Journal Of Parallel and Distributed Computing*, 10(4):289–308, 1990.

[16] K. Gharachorloo, A. Gupta, and J. Hennessy. Hiding Memory Latency using Dynamic Scheduling in Shared-Memory Multiprocessors. In *Proceedings of the 19th Annual International Symposium on Computer Architecture, Gold Coast, Australia*, pages 22–33. ACM Press, 1992.

[17] J. E. Hicks. Experiences with Compiler-Directed Storage Reclamation. In *Proceedings of the 6th Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark*, June 1993. (to appear).

[18] W. D. Hillis and G. L. Steele, Jr. Data parallel algorithms. *CACM*, 29(12):1170–1183, Dec. 1986.

[19] K. Hiraki, K. Nishida, S. Sekiguchi, T. Shimada, and T. Yuba. The SIGMA-I Dataflow Supercomputer: A Challenge for New Generation Supercomputing Systems. *Journal on Information Processing*, 10(4):219–226, 1987.

[20] J. Hughes, editor. *Proceedings of the 5th Conference on Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts*, volume 523 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[21] A. Iyengar. *Dynamic Storage Allocation on a Multiprocessor*. PhD thesis, MIT, EECS, Laboratory for Computer Science, Cambridge, MA, 1992.

[22] H. Koike and H. Tanaka. Overview of the Parallel Inference Engine: PIE64. In *Annual Report of the Engineering Research Institute of the University of Tokyo*, June 1990.

[23] M. S. Lam and R. P. Wilson. Limits of Control Flow on Parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture, Gold Coast, Australia*, pages 46–57. ACM Press, 1992.

[24] V. Natarajan, D. Chiou, and B. S. Ang. Performance Visualization on Monsoon. *Journal of Parallel and Distributed Computing*, 1993. (to appear).

[25] R. S. Nikhil. Id Reference Manual, version 90.1. Computation Structures Group Memo 284-2, Massachusetts Institute of Technology Laboratory for Computer Science, Cambridge, MA, Sept. 1990.

[26] 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*. Academic Press, 1990.

[27] R. S. Nikhil, G. M. Papadopoulos, and Arvind. *T: A Multithreaded Massively Parallel Architecture. In *Proceedings of the 19th Annual International Symposium on Computer Architecture, Gold Coast, Australia*. ACM Press, May 1992.

[28] K. Nishida, K. Toda, and T. Shimada. The Hardware Architecture of the CODA Real-Time Parallel Processor. Technical Report TR-92-46, Electrotechnical Laboratory, Tsukuba, Japan, Dec. 1992.

[29] G. M. Papadopoulos. Program Development and Performance Monitoring on the Monsoon Dataflow Multiprocessor. In *Proceedings of the Workshop on Instrumentation for Future Parallel Computing Systems*. ACM Press, 1989.

[30] G. M. Papadopoulos. *Implementation of a General-Purpose Dataflow Multiprocessor*. Research Monograph in Parallel and Distributed Computing. MIT Press, 1992.

[31] G. M. Papadopoulos and K. R. Traub. Multithreading: A Revisionist View of Dataflow Architectures. In *Proceedings of the 18th Annual International Symposium on Computer Architecture, Toronto, Canada*. ACM Press, 1991.

[32] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba. An Architecture of a Dataflow Single Chip Processor. *Proceedings of the 16th Annual International Symposium on Computer Architecture, Jerusalem, Israel*, pages 46–53, 1989.

[33] M. Sato, Y. Kodama, S. Sakai, Y. Yamaguchi, and Y. Koumura. Thread-based Programming for the EM-4 Hybrid Dataflow Machine. In *Proceedings of the 19th Annual International Symposium on Computer Architecture, Gold Coast, Australia*, pages 146–155. ACM Press, 1992.

[34] K. E. Schauser, D. E. Culler, and T. von Eicken. Compiler-Controlled Multithreading for Lenient Parallel Languages. In *Proceedings of the 5th Conference on Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts*, volume 523 of *Lecture Notes in Computer Science*, pages 50–72. Springer-Verlag, 1991.

[35] A. Shaw. Implementing Data-Parallel Software on Dataflow Hardware. Master's thesis, MIT, EECS, Laboratory for Computer Science, Cambridge, MA, Jan. 1993.

[36] B. J. Smith. Architecture and Applications of the HEP Multiprocessor System. In *Real-time Signal Processing IV*, volume 298, pages 241–248, Aug. 1981.

[37] E. Spertus, S. Goldstein, K. Schauser, T. von Eicken, and D. Culler. Evaluation of Mechanisms for Fine-Grained Parallel Programs in the J-Machine and the CM-5. In *Proceedings of the 20th International Symposium on Computer Architecture*, May 1993. (to appear).

[38] K. M. Steele. Implementation Specification for a Minimal I-Structure Controller. Technical Report MIT/LCS TR-471, Massachusetts Institute of Technology Laboratory for Computer Science, Cambridge, MA, Sept. 1989.

[39] K. R. Traub. A compiler for the MIT Tagged-Token Dataflow Architecture. Technical Report MIT/LCS TR-370, Massachusetts Institute of Technology Laboratory for Computer Science, Cambridge, Massachusetts, Aug. 1986.

[40] C. B. Weinstock and W. A. Wulf. Quick Fit: An Efficient Algorithm for Heap Storage Allocation. *SIGPLAN Notices*, 23(10):141–148, 1988.

Contents