LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Overview of the Monsoon Project

Computation Structures Group Memo 338
January, 1991

Kenneth R. Traub
Gregory M. Papadopoulos
Michael J. Beckerle
James E. Hicks
Jonathan Young

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

Overview of the Monsoon Project

Kenneth R. Traub
Motorola Cambridge Research Center

Gregory M. Papadopoulos
Massachusetts Institute of Technology

Michael J. Beckerle
Motorola Cambridge Research Center

James E. Hicks
Massachusetts Institute of Technology

Jonathan Young
ICAD, Inc.

Abstract

Monsoon is an experimental multi-threaded multiprocessor targeted to large-scale, *general purpose* scientific and symbolic computations. In particular, Monsoon is designed for the efficient execution of code compiled from Id, a high level, implicitly parallel declarative language [6]. Monsoon is a product of a multiyear cooperative research and development program between the Massachusetts Institute of Technology and Motorola, Inc., which, in turn, is an outgrowth of over ten years of research in dynamic dataflow architectures and languages conducted at MIT. The intent of this paper, and of its sequels [5, 4], is to provide an overall view of the Monsoon hardware and software system architecture.

1 Hardware Architecture

As shown in Figure 1, a Monsoon machine includes a collection of pipelined processing elements (PE's), connected via a multistage packet switch network to each other and to a set of interleaved I-structure memory modules (IS's). The PE nodes implement hardware primitives for direct support of efficient multithreading, including zero-cycle context switching, single cycle fork and join, and split-phase memory references with arbitrary reordering. The IS nodes implement event-driven synchronization through the atomic manipulation of presence bits which augment each location of memory. Each PE references the set of nodes as a uniform global address space.

The basic run-time execution state of a Monsoon program comprises a tree of activation records, or *frames,* which correspond to the invocation state of many simultaneous procedures. Unlike the traditional von Neumann stack model where only the top-most frame is active, any subset of Monsoon program frames may be executing at the same time. Moreover, there may be many active *threads* of computation within each frame. By using split-phase transactions, threads may freely reference shared objects on a global heap. The heap may be distributed across both PE and IS nodes, but a given activation frame is mapped at invocation-time entirely onto a single node. Please refer to Figure 2.

The state of a thread is contained in a *computation descriptor*, or CD. A CD, also called a *token*, has a value register V, and a continuation register C that defines the context in which the CD's thread executes. The C register contains a pair of pointers: the *instruction pointer*, IP, that indicates the next instruction to be executed, and a *frame pointer*, FP, indicating its associated frame in data memory. Each register is 72 bits: 64 bits of value and eight bits of (optional) run-time type information.

1.1 Processing Elements

A Monsoon PE is a 64-bit, highly pipelined processor. A PE interleaves execution of up to eight threads, drawn from a potentially very large set of ready threads whose CD's are kept in a pair of hardware queues local to each PE. The ready threads, in effect, represent available parallelism that the processor can draw upon to fill its pipeline and to mask the latency of remote memory references.

Each trip through the pipeline, a thread fetches an instruction in the first stage, performs a read or write to its activation frame in the next three stages, performs an ALU/FPU operation in the following three stages, and in the last stage produces zero, one, or two CD's which emerge at the bottom. Normally, one of these CD's is immediately recirculated to the top of the pipeline; typically this CD has the same FP and an incremented IP compared to its parent, and so a chain of CD's like this may be conveniently viewed as a sequential thread [7]. A CD produced at the bottom of the pipeline may also enter one of the two thread queues, for later execution when no thread is

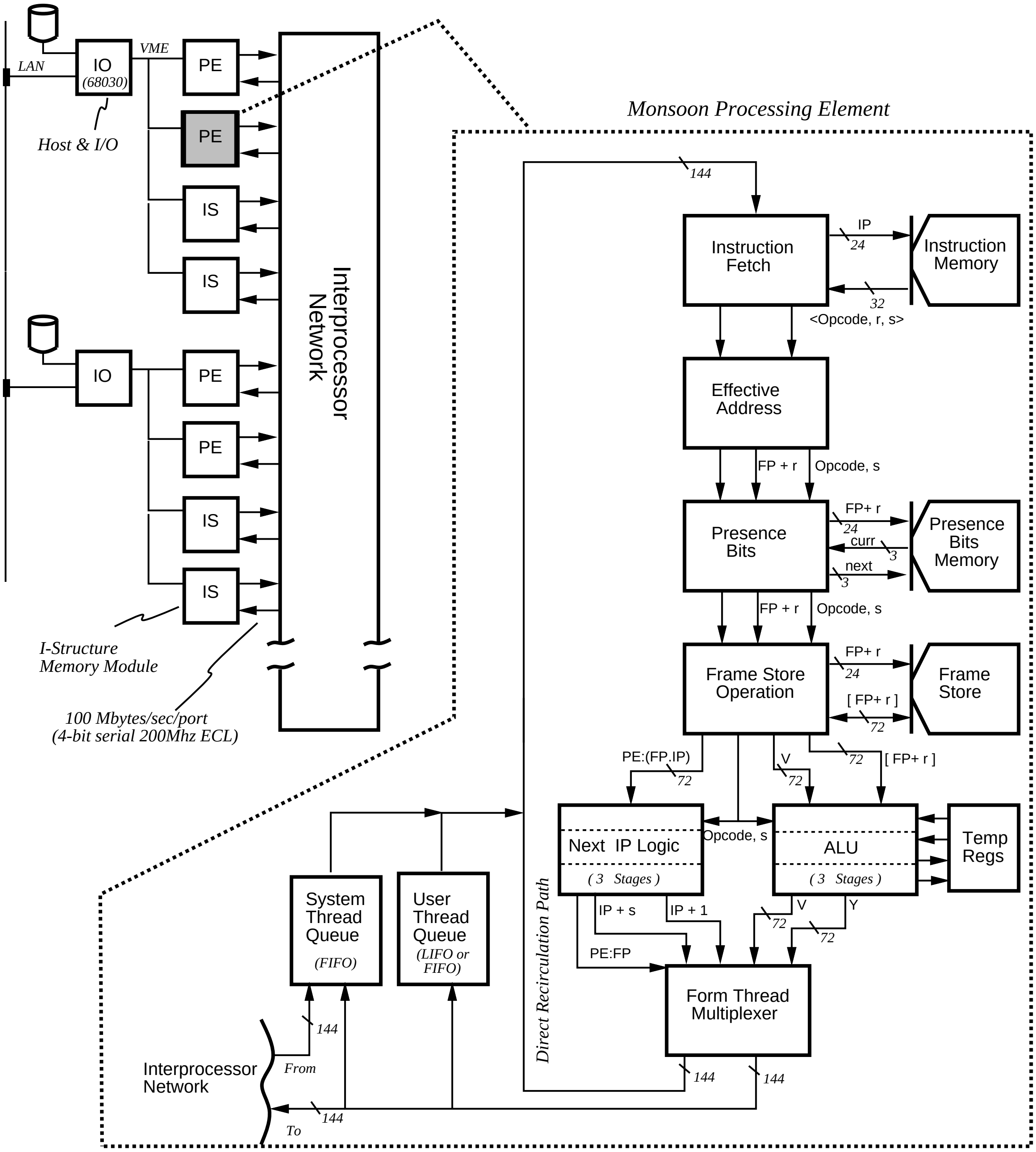

Figure 1: Monsoon Hardware Architecture

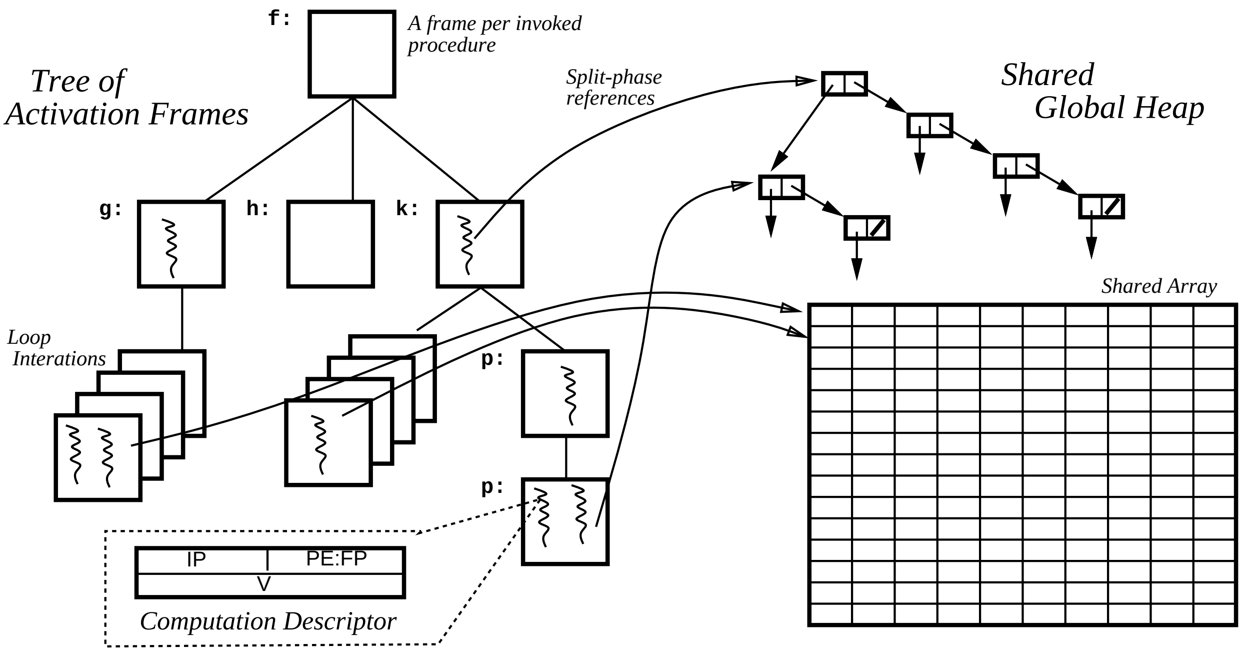

Figure 2: Monsoon Program Execution State

recirculated. In particular, when two CD's are emitted from the pipe one is often enqueued: this is a single-cycle *fork*. The other possibility for a new CD is that it enters the interprocessor network where it is automatically routed to another PE or IS. The most common case here is a split-phase transaction, where the CD sent to the network is a memory request, and the response will arrive sometime later in the form of a new CD that enters the System Thread Queue.

Two independent CD's may synchronize with each other by using the presence bits found on every word of frame memory in the following manner. The first thread, finding the presence bits of a frame location set to `empty`, sets them to `full`, records its V register in the location, and produces zero tokens at the bottom of the pipe (*i.e.*, the thread dies). The second thread finds the presence bits `full`, and so continues execution, with both its own V register and the V of the other thread, recorded earlier in the frame slot, as operands.

A PE is implemented on a single 9U×400mm surface-mounted printed circuit card which supports a VME port for diagnostics and input/output, and two unidirectional 100 Mbytes/sec network links (see [5]). The processor core is byte sliced into eight 10,000 gate 1.5 micron CMOS arrays. The floating point unit is fully pipelined, yielding up to 10 million double precision floating point operations per second for a 100 ns. processor cycle time. The current generation implements a 256 KWord (32 bit) instruction memory, 256 Kword (72 bit) frame store, and two 64 KWord (144 bit) thread queues.

1.2 I-Structure Modules

An I-Structure module [8] has a two stage pipeline, a memory stage and an output stage. In the memory stage, an incoming request is decoded into an operation code, memory address on the IS, and a value or return continuation. A memory operation (read, write, or exchange) is performed on the location, and, for some operations, a response generated in the form of a new CD to be sent to the requesting PE. A typical store request stores the incoming value and generates no response, while a typical fetch request forms a new CD from the fetched value and the return continuation from the request. If a response is generated, it is injected into the interprocessor network in the output stage. Like the PE, each word of IS memory is equipped with presence bits. These are used to implement a variety of synchronizing memory operations, including *I-structure* operations for producer/consumer synchronization [2], and *M-structure* operations for mutual exclusion [3].

Like the PE, and IS is implemented on a single 9U×400mm surface-mounted printed circuit card which supports a VME port for diagnostics and input/output, and two unidirectional network links. In the current configurations, each IS supports four million words (72 bits) and processes four million requests per second.

1.3 Two Node and Sixteen Node Configurations

We have constructed Monsoon configurations comprising sixteen nodes (eight PE's and eight IS's) and a two-stage packet switching network. Associated with each group of four nodes (two PE's and two IS's) is a 68030-based UNIX front end (the "host") for bootstrap, code loading, and I/O. We have also constructed a number of two node configurations comprising a single PE, an IS, and 68030 front end. The sixteen node systems are intended for scalability studies, while the two node systems should be useful for code development.

2 Software Architecture

The software system for Monsoon has two goals. The first goal is to provide a powerful environment for developing programs for Monsoon, principally in the Id programming language. Rapid program development is supported by a tightly integrated edit-compile-run cycle, including the facility for separate compilation and dynamic linking of procedures directly into the execution vehicle. The second goal is to provide a flexible, instrumented environment for experimentation with the Monsoon architecture. This is supported through the integration of statistics gathering and display tools into the programming environment; these tools provide access both to hardware statistics registers and to the more detailed statistics available from the software emulator. Of course, these goals overlap to a large extent: for example, statistics may be used to help tune the performance of an application. It should be noted that Monsoon hardware provides no facilities for multi-tasking or address space protection, and so the software system takes the view that at any given time, the whole machine (or static subdivision thereof) is devoted to a single program.

The architecture of the software system is depicted in Figure 3. At the top of the figure are compilers for the languages supported on Monsoon, currently assembly language and Id [6]. The Id compiler supports separate compilation of user procedures, but is still able to do interprocedural optimization via a database of separate compilation information. A trio of programs running on the host interact with Monsoon itself. The Loader dynamically links and loads separately compiled procedures directly into the machine, and records inter-module dependences so that consistency may be checked prior to execution. A Debugger provides the ability to examine and change the contents of memory, for debugging of programs or post-mortem analysis. The Execution Manager is the program running on the host when Monsoon is actually executing a user program. Normally, its only role is to start the program and await its termination, but it can also collect run-time statistics during execution. If the user has not requested run-time statistics collection, the Execution Manager does not interfere with or slow down Monsoon at all. An off-line Statistics Viewer analyzes and presents statistics collected during execution, with a variety of features for interactive display via X-windows or hardcopy via PostScript. Finally, the large box labeled "Id World" in the figure integrates the software components into a seamless interactive program development environment.

3 Emulator

The Loader, Debugger, and Execution Manager interact with Monsoon through the Monsoon Machine Interface, or MMI, which provides access to the machine via a client/server network communications model. The Monsoon hardware provides a server, and the Loader, Debugger, and Execution Manager are clients of that server.

The Monsoon features visible via the MMI are carefully chosen to avoid revealing too many implementation details of Monsoon. In particular, details about the scan rings and decode memories are suppressed, leaving only the instruction set architecture, or macro-architecture, of the machine visible. As a consequence client programs such as the loader or program debugger cannot tell whether the server to which they are connected is really Monsoon hardware, or an instruction-level emulator. In fact, the software system includes such an emulator, called MINT ("Monsoon INTerpreter"), which avoids the overhead of modeling internal operational details of the hardware. To the software system, the emulator and Monsoon hardware are indistinguishable, and for that reason we refer to the emulator as *plug compatible* with the hardware. Of course, hardware-specific debugging tools, such as the interactive scan-path debugger, do not interface via the MMI, but rather at a lower level (not pictured in Figure 3).

The MINT emulator serves a number of purposes within the software system. First, MINT has provided a vehicle for early debugging of the software system before hardware was available. This aspect of MINT is discussed further in another paper in this proceedings. Second, MINT provides an execution vehicle when Monsoon hardware is in use or otherwise unavailable. Third, MINT is a useful experimental vehicle, as it can be easily instrumented to allow gathering of execution statistics beyond those that the hardware can gather. For example, MINT has been modified to gather address traces for studying the effect that caches might have on the machine architecture.

One of MINT's most powerful features for exper-

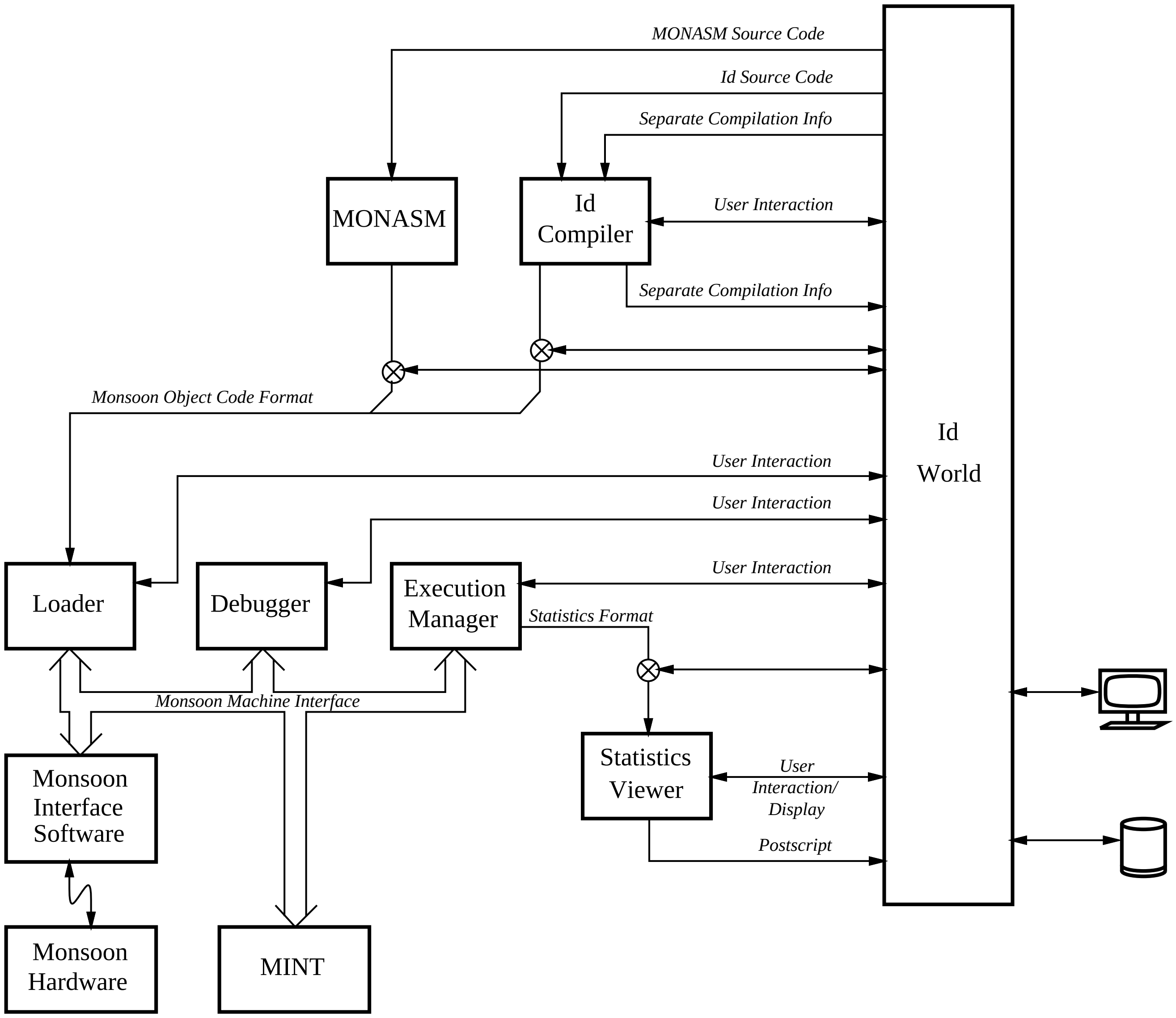

Figure 3: Monsoon Software Architecture

imentation is the ability to select among different *queueing systems* [9]. Queueing systems exploit the property of compiled Id code which allows CD's to be processed in any order without changing the answer computed by a program. By altering the order in which CD's are processed and the grouping of CD executions into artificial "timesteps," a wide variety of measurements about program behavior may be taken. For example, one queueing system employs one queue from which CD's are dequeued for execution and a second into which new CD's are enqueued. When the first queue is empty, the timestep is declared finished and the queues are exchanged to begin the next timestep. This models an ideal machine in which there are an unbounded number of processors, zero communication latency, and perfect load distribution (*i.e.*, all CD's available in one timestep are processed in that timestep by separate processors—such a distribution is not actually realizable because all threads within a given activation frame are constrained to execute on a single PE). A graph of the instructions executed in each timestep under ideal execution is called a *parallelism profile*, and is quite valuable in studying applications to see where the parallelism lies and how algorithms might be improved [1].

The interchangability of MINT and the hardware permits the same code for which idealized statistics were gathered to be run on the hardware, to see whether the system is able to exploit parallelism as predicted by idealized execution.

4 Run Time System

The Id Run Time System (RTS) is a collection of software procedures linked into every Id program, and which execute on Monsoon as part of the user's program. The Run Time System provides four basic services: frame management for procedure activations, heap management for aggregate data, error and exception handling, and I/O. The first two of these services are critical to multi-processor performance, as they are responsible for balancing of computation and distribution of data, respectively, among the nodes of the machine. Because these procedures are dynamically linked into compiled code, it is possible to experiment with different policies for balancing and distribution without recompiling user applications.

Activation frame management consists of the routines `get-context` and `return-context`, which allocate and deallocate the temporary storage needed to call a procedure. When an activation frame for a procedure is allocated on a different PE than that of the caller, work migrates from one PE to the other as part of the calling convention. Thus, the Run Time System's choice of where to allocate a new frame controls how load is distributed. Notice that work distribution is confined to procedure boundaries, but the compiler is free to use the procedure calling mechanism at places other than source code procedure boundaries. In particular, different iterations of large loops are routinely distributed across the machine.

The heap manager consists of the routines `get-aggregate` and `return-aggregate`, which allocate and deallocate storage on the I-structure modules. The heap manager must balance storage usage across I-structure modules so that individual I-structure modules do not become contention points. Hardware support for interleaving adjacent words across nodes is provided to assist in this.

Error and exception handling by the RTS is fairly straightforward. The RTS simply records the error in a vector of error descriptors and optionally requests that the host bring the whole program to a halt. If the user defers errors, parts of the program continue in parallel even though some of the threads have been stopped by exceptions or errors.

Finally, the RTS provides input/output services to the program. I/O routines in the RTS pass requests via a DMA interface to the Monsoon Interface Software running on the host (as Monsoon has no direct hardware interface to peripherals), whereupon the requests are handled and any data returned to the RTS for further processing by the user program.

The dynamic linking of RTS procedures into the user program is through software trap instructions. This provides a much more efficient transfer of control than conventional procedure linkage. More importantly, it provides an interesting level of flexibility to the instruction-level emulator, MINT. While MINT is fully capable of emulating the execution of RTS procedures, just as they would execute on the hardware, it is also possible for MINT to recognize the software trap instructions and branch to special code within MINT to simulate the RTS. This has the effect of compressing the execution of an RTS procedure, for statistical purposes, into a single instruction time. Typically this is used in conjunction with the idealized queueing system; because each RTS call is counted as a single instruction, the parallelism profile is not skewed by serialization and other implementation details of a particular RTS.

[1] Arvind, D. E. Culler, and G. K. Maa. Assessing the benefits of fine-grained parallelism in dataflow programs. *Int. J. of Supercomputer Applications*, 2(3):10–36, 1988.

[2] Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: Data structures for parallel computing. In *Graph Reduction*, volume 279 of *LNCS*, pages 336–369. Springer-Verlag, Oct 1986.

[3] P. S. Barth, R. S. Nikhil, and Arvind. M-structures: Extending a parallel, non-strict, functional language with state. In *FPCA '91*, 1991. (To Appear).

[4] M. J. Beckerle and G. M. Papadopoulos. Test and validation for Monsoon processing elements. In *ICCD'91*. IEEE, Oct 1991. (To Appear).

[5] C. F. Joerg and G. A. Boughton. The Monsoon interconnection network. In *ICCD'91*. IEEE, Oct 1991. (To Appear).

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

[7] G. M. Papadopoulos and K. R. Traub. Multithreading: A revisionist view of dataflow architectures. In *Proc. 18th Ann. Int. Symp. on Comp. Arch.*, pages 342–351. IEEE, May 1991.

[8] K. M. Steele. Implementation of an I-structure memory controller. Master's thesis, MIT, Cambridge MA, 1990.

[9] K. R. Traub. MINT white paper. Technical Report MCRC-TR-2, Motorola CRC, Cambridge MA, Oct 1989.