LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

StarT the Next Generation: Integrating Global Caches and Dataflow Architecture

CSG Memo 354
August 5, 1994

Boon Seong Ang, Arvind, and Derek Chiou

Proceedings of the ISCA 1992 Dataflow Workshop, Hamilton Island, Australia (to be published), May 1992

This paper 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 N00014-92-J-1310.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

StarT the Next Generation: Integrating Global Caches and Dataflow Architecture

Boon Seong Ang, Arvind, and Derek Chiou
Laboratory for Computer Science
Massachusetts Institute of Technology

August 5, 1994

Abstract

The implicitly parallel programming model provides an attractive approach to deal with the complexity of parallel programming. Implementing this model efficiently, especially on stock processors, remains a big challenge, partly because of the fine granularity of the parallelism exploited. The Monsoon[27] project was designed to address and investigate support for fine-grain parallelism, and has yielded very encouraging results[13].

Our experience with Monsoon and *T[24, 28], a followup project after Monsoon, suggests that provision for global shared memory is an area where both the Monsoon and *T architectures can be improved. Starting with the split-phase approach used in Monsoon and *T, we propose to augment global memory access by including coherent global caches.

The rapid improvements in stock microprocessors, and the high cost and effort required to develop a competitive microprocessor, presents practical constraints on what can be built in any experimental architecture project. We propose a machine that attempts to include as many of the desired features as possible within the constraint of using a stock microprocessor. This machine will allow us to continue research into the dataflow approach to parallel computing as well as provide a prototype of a commercial product.

1 Introduction

The dataflow approach to parallel computing has been very successful in at least two important aspects: (i) exposing parallelism and (ii) using fine-grain parallelism to tolerate long latency operations and synchronization waits. These features have been demonstrated in several studies on real machines[13, 9]. There are, however, still several obstacles in making the dataflow approach a truly viable solution to parallel computing.

The biggest challenge is to find a way of executing dataflow style code efficiently on a wide variety of platforms. One approach is to improve the compilation of dataflow style languages to reduce synchronization requirements, instruction counts, non-local communication, and so on. Research efforts are underway to try to achieve this goal without affecting the ability to expose parallelism and tolerate latency. Another approach is to identify the basic architectural mechanisms that will efficiently support the dataflow style of fine-grain parallel execution. For pragmatic reasons the resulting architecture should be as close to conventional architectures as possible to take advantage of the decades of experience in building and programming conventional processors, as well as to make maximum use of existing hardware and software subsystems.

This paper will concentrate on architectural features that are needed to support fine-grain parallel execution. We discuss the lessons we have learned from the Monsoon project[27, 13] and the early phases of implementing the *T Multithreaded architecture[24]. Drawing on these experiences, and the basic requirements of any parallel computation, we suggest a desirable set of features to be incorporated in our current machine called *T-NG (StarT, the Next Generation). Essentially *T-NG has better provision for global shared memory than either Monsoon or 88110MP [28], a Motorola 88110 based implementation of the abstract *T model presented in [24]. Our proposal adds caching of global memory in a way that allows it to coexist with the split-phase transactions and the multithreaded approach used in Monsoon and the 88110MP.

The rapid improvements in stock microprocessors and the high cost and complexity of developing a modern microprocessor imposes practical constraints on what can be built in any experimental project that wishes to have competitive performance. *T-NG has to work within these constraints, and thus represents many compromises. Nevertheless, it incorporates many of the desired features. We hope that the importance of the missing features can be investigated through the effects of their absence. The machine will also provide the platform to continue research into compilation technology.

The rest of the paper is organized as follows: Section 2 describes some of the models for parallel programming. Section 3 describes the architectural concerns of running fine-grain parallel programs, especially on stock hardware. It also describes architectural modifications that would allow dataflow programs to run much more efficiently and how those modifications may help other programming paradigms as well. Section 4 examines Monsoon, an implementation of a processor designed to be able to exploit fine-grain parallelism, and how it performed. That section ends with a discussion of the deficiencies of Monsoon, and how they may be fixed. Section 5 describes *T, an architecture that addresses some of Monsoon's weaknesses while moving towards a conventional microprocessor based machine. The initial plan was to build *T around Motorola's MC88110 microprocessor. However, for reasons beyond the control of the project, these plans had to be abandoned in late 1993. While drawing up new implementation plans centered around the Power PC architecture family, we took the opportunity to explore incorporating global caches into our new machine. Section 6 gives a high-level view of this machine. *T-NG is an on-going project; thus, the discussed implementation should not be assumed to be final. Finally, we conclude this paper with a brief look at related work.

2 Parallel Programming

Writing parallel programs is currently much more difficult than writing sequential programs. The difficulty arises from three problems: (1) managing the control flow along multiple threads of execution, (2) managing the placement and access of global data, (3) timing events in such a way that parallel speedup is achieved. While it is possible for a programmer to explicitly choreograph a parallel program by indicating which tasks execute when, how tasks communicate and synchronize, and where data should be placed and how it can be accessed, such an undertaking is extremely complex. Several parallel programming models have thus emerged in an attempt to cope with these difficulties. We discuss some of these below.

2.1 Data Parallel Model

The data parallel programming model, exemplified by languages such as CM Fortran, Fortran 90, Fortran D, and High Performance Fortran (HPF), is a very popular model of parallel programming. The data parallel programming model maintains the single thread of execution of sequential programming. That single thread of execution, however, can perform the same operation in parallel on a selected subset of array elements (hence the name, *data parallel*). Conceptually each data element resides on a specific virtual processor. Periodically, data elements have to be moved to a virtual processor that needs them for computation. Since the cost of data movement is high, the initial data placement, and the data movement operations must be planned very carefully to achieve high performance. Currently, the programmer has to help with the placement of global data by providing directives (automating the data placement process is currently an active area of research). Once the placement is specified, the compiler takes care of the tedious task of generating the actual memory access code.

SIMD machines, like the CM-2, provide a direct hardware implementation of this programming model. Processors in the machine operate in lock-step, with a single instruction stream controling the operation of all the processors. General communication, that is an arbitrary permutation of array elements, is very expensive. For example, general communication on the CM-2 takes about 80 times as long as a primitive arithmetic operation; nearest neighbor communication is an order of magnitude faster, but still takes about 4 times that of a primitive arithmetic operation. A program must keep general communication of data to a minimum in order to achieve reasonable performance. Unfortunately, some programs, are not amenable to this type of tuning.

The data parallel programming model has been implemented on various types of MIMD machines such as distributed memory, message passing machines (*e.g.,* CM-5, Paragon), cache-coherent shared memory machines, and dataflow machines (*e.g.,* EM-4[34]). Although each processor has its own thread of control during execution, the program still follows a single thread of control logically. In order to preserve the lock-step execution semantics of data parallel programs, implementations on MIMD machines have to perform explicit barrier synchronization periodically. Barriers can be implemented in software on top of the communication facilities of the machine[34]. Many global communication primitives have the implicit effect of barrier synchronization, making the removal of redundant explicit barriers a compiler optimization. For machines that do not have very high performance networks, specialized global synchronization hardware as found on the CM-5 is sometimes provided. On the other hand, the ability of each processor to have its own thread of control makes MIMD machines much more efficient at executing conditionals than SIMD machines.

Some distributed memory machines favor large messages over small ones due to poor network interfaces. On such machines, the compiler or the programmer has the added burden of grouping small messages together (*i.e., message vectorization*). Managing the global movement of data remains the biggest issue in the implementation of data parallel model. Split-C is a language that provides a convenient way of expressing global data movement in the data parallel model[8].

Placement of data is significantly easier on shared memory machines than on distributed memory machines. Shared memory machines are somewhat more tolerant of dynamic communication requirements, but still require good temporal and spatial locality to perform well. Careful orchestration of data access pattern is still important on these machines.

The data parallel programming model, because of its closeness to sequential languages, and because of its good performance for a certain class of programs (mostly dense matrix computations) on a fairly wide range of hardware platforms, has gained wide acceptance. The data parallel model is, however, restricted in the forms of parallelism that it can exploit due to its single thread of control. It is also restricted in the class of programs that it can efficiently execute due to data placement requirements. Any sort of irregularity rapidly decreases the efficiency of data parallel programs, even though there may be considerable parallelism in the underlying algorithms. As far as we know no one has even attempted to express a symbolic code, such as a compiler, in the data parallel model. These types of programs need both a more general model of parallel computing as well as a machine that can support general communication efficiently.

2.2 Control Parallel Model

Programming models that try to exploit task level parallelism represent a large departure from the data parallel programming model. These models do not restrict parallelism to concurrent operation on arrays. Instead, they attempt to make full use of multiple threads of control. “Data parallelism” can still be exploited since it can be expressed as multiple threads of execution on different array elements. These models, however, are more complicated than the data parallel model, since they must manage multiple threads of control. In the next few paragraphs, we discuss several programming models that fall into this general category, but differ from one another in some significant aspects.

Communicating Sequential Processes (CSP) based programming languages, such as Ada and Occam, offer one approach to control parallel programming. The multiple threads of control are exposed at the source language level. These threads are called tasks in Ada and processes in Occam. Communication between processes is through fixed channels, and synchronization is achieved through the use of blocking sends and receives. Globally shared memory is usually not supported in this model. Any shared data has to be communicated through messages managed by the programmer. The network of processes and channels corresponding to a program are statically mapped onto physical processors and channels. A very simple scheduler on each processor schedules processes and manages communication. While this programming model puts the power of writing parallel programs in the programmer's hands, it does not provide very much support in managing the complexity of parallel programming.

PVM[12] represents another approach to parallel programming which is closely related to the CSP model. This model is implemented via libraries on top of conventional sequential languages like C or Fortran. PVM consists mostly of send and receive functions. Messages have to be explicitly received. Receives can be blocking or non-blocking and can specify a message sender processor and/or a message type to receive, but are not required to do so. Shared memory is not directly supported by PVM. There are other parallel programming "languages"/packages that are similarly implemented via libraries on top of C that do support the abstraction of shared memory. An example is the Argonne National Laboratory's (ANL) parallel processing macro package, parmacs, used in the SPLASH benchmarks[36]. The management of multiple threads of control is still the programmer's responsibility in all these packages.

In an effort to exploit finer grain parallelism, several programming models take a much more dynamic view of thread creation and synchronization. Each processor keeps a pool of threads and does "multithreaded execution". This allows the idle cycles in one thread, due to such things as waiting for synchronization or remote memory operations, to be used by other threads to perform useful work. A very dynamic form of scheduling is required by this model. In general, management of threads can be done in user code, in the operating system, or by a system supplied run-time system (RTS). Having the user manage threads is impractical, tedious, and error prone. Having the operating system manage the threads imposes extremely high overheads. Both methods are prone to deadlocks and race conditions. A better solution is to hide this complexity from the user and have the system software, either compiler, run time system, or both handle this task. When the compiler generates threads, it is able to adhere to certain restrictions that allow the RTS to ensure that deadlocks and races do not occur. The TAM model, which we will discuss later, embodies many of these features[9].

2.3 Implicitly Parallel Model

Id, a parallel language based on functional languages, represents an effort at exploiting fine-grain parallelism while keeping the task of parallel programming manageable [21]. The implicitly parallel programming model offered by Id insulates the programmer from the concerns of managing parallel threads of control and global data. Instead of a data parallel, or task-level model, the programmer is provided with a data-driven model. In writing code with the functional subset of Id, the programmer only has to think about expressing data dependencies, a task that is inherent in any form of programming. The functional nature of the language allows the compiler to extract fine-grain parallelism and generate code that takes advantage of multiple threads of execution via multithreading. In Id, synchronization is managed automatically by the compiler, allowing fine-grain producer-consumer synchronization. The programming model provides global shared memory by default, and thus, the programmer does not have to manage data movement explicitly.

The functional nature of Id is the key that provides automatic extraction of parallelism. Other functional languages, notably SISAL[7], are similarly able to exploit implicit parallelism. Id goes one step further by providing non-strict procedure calls, which allow a call to begin execution before all the arguments have arrived, thus pushing the idea of an eager, data-driven execution to its limits.

The implicitly parallel approach provides a model that takes full advantage of all forms of available parallelism, while hiding the complexity of multiple threads of control. It is more general than the data parallel model in the forms of parallelism it can exploit, while avoiding the coding complexity of the control parallel models. Having such positive traits, the implicitly parallel approach will be a clear winner if the model could be implemented efficiently. Here, however, lies the difficulty encountered in this approach. The fine granularity of the parallelism makes efficient implementation on commercial parallel machines difficult. Since synchronization is pervasive, threads are switched frequently. Compile time information for managing locality is simply not available under such dynamic conditions, and this makes it difficult to generate efficient code.

The implicitly parallel model also requires a global shared address space and word-level memory accesses. Commercially available parallel machines either do not support these abstractions or do so poorly, thus increasing the difficulty in efficient implementation. Despite these problems, considerable progress has been made. The Monsoon dataflow machine, which we will discuss in greater detail in Section 4, is a demonstration that hardware can be built to support such a programming model. Much work still has to be done to deliver the promises of implicit parallel programming. The potential payback of taking advantage of all forms of useful parallelism implicitly, without the complexity for the programmer, is a great incentive to continue research in this area. This paper will deal with some of these difficulties and propose means of overcoming them.

3 Fine-Grain Parallel Programs on Stock Hardware

A natural question to ask dataflow architecture researchers is "why not use stock hardware?" Given the availability of very high performance microprocessors, it is a very relevant question. The problem is that the demands of fine-grain parallel execution are different than the demands made by sequential code. Current microprocessors are optimized for the latter. It is a challenge to determine whether processors features for fine-grain codes can coexist with existing features for sequential codes.

3.1 What is fine-grain parallel execution?

Dataflow graphs offer perhaps the most mature model for fine-grain parallelism. These graphs are usually generated by a compiler from a functional language (see [4] for an example). We briefly explain the execution of a dataflow graph.

The nodes of a dataflow graph specify the operations to be performed on the values that flow on *tokens* along arcs between the nodes. Each token also carries a *continuation* consisting of a node number, an instruction pointer and a frame pointer. A continuation specifies the computation to be done with the value on the token. Execution starts with a pool of initial tokens. When a token is processed, a waiting/matching store is checked to see if the token's partner, if any, is already present. If not, the token is inserted into the waiting/matching store, and the next token is processed. If the synchronization is successful (the correct partner token is found), the node is fired (*executed*) and the resulting values are put back into the pool. The order in which tokens are processed does not affect the result of the computation, but may greatly affect the running time and the resources required. To avoid deadlocks, it is essential that a value that fails to synchronize (*i.e.,* one or more of its partners has not yet been processed) should not block the processing of other tokens.

Our dataflow graphs allow global memory in the form of I-structures which are write-once, read-many-times data structures. Accessing an I-structure location is done by splitting the read operation into two separate phases (see [4] for details). In the first phase, the read (fetch) is initiated and the processor is then relinquished to another thread. The read request consists of a continuation and the address to be read. In the second phase, the read is performed and the result is returned (actually forwarded) to the waiting node, specified in the continuation supplied on the request tokens.

In such a *split-phase* operation, the actual reading of the value may occur on some processor other than the one where the read is initiated. Depending on the dynamic execution order, a value to be read may not have been written yet. If it is unknown whether a value being read has already been written, the reads and write require some synchronization. Synchronizing reads (*e.g.,*I-structures) may take an arbitrary amount of time to complete, since an arbitrary amount of time may elapse before the value is actually written. In the case where a read arrives before the write of a location, the second phase of the read request is deferred (*i.e.,* queued up) until the write occurs. It is important to note that deadlocks are not possible (as long as the program does not include any cyclic data dependence) because the node waiting for a reply does not block any processor or memory operations. Once the write is completed, the request is satisfied in the usual manner.

If every node of the dataflow graph is a simple operation like add or memory fetch, an inordinate number of synchronizations need to be performed, because almost every instruction requires synchronization. It is easy to see, however, that a node could consist of more than one operation. Operations within a node could themselves be organized as a dataflow graph or as a simple linear sequence of instructions. We call the latter case, where the nodes in the dataflow graph are sequences of instructions, *multithreading*, and each of the linear instruction sequences, *threads*. This model of multithreading is essentially identical to the Threaded Abstract Machine (TAM) proposed by Culler *et. el.* [9]. TAM distinguishes between inlet threads and computation threads — the former are threads executed in response to incoming messages. A thread scheduling model is also part of TAM. There are several other variants of multithreaded models – depending on whether threads are allowed to be suspended and resumed, and the types of instructions that can be included in a thread (see, for example, Traub [38] or Iannucci[14]). Nikhil[22] is a good introduction to compiling an implicitly parallel language into multithreaded code.

3.2 What architectural support does multithreading require?

Multithreading requires some hardware support to run efficiently. In the following, we discuss the categories of hardware support that would be useful, and how current stock processors fail to provide that support.

3.2.1 Message Passing

Messages are used to start threads, to make a request or reply, or to pass information. Since every instance of a dataflow graph node could potentially be executed on a different processor, a huge number of interprocessor messages could be generated by a multithreaded program. A good network interface is probably the single largest improvement that can be made to a processor to support multithreading[28]. A fast processor/network interface that requires the fewest number of processor cycles to send and receive messages would be vital. The efficiency of the interface can be measured in terms of the number of processor cycles needed to send a certain unit of data, say a 64-bit word. The raw network latency is somewhat less critical, since multithreading can be used to tolerate it.

The fastest possible network interface can format and send a message in a single cycle. Given a standard RISC processor, formatting and sending a message in a single cycle would require using registers to provide the necessary pieces of the message, forcing large hardware modifications to insert the network interface into the processor itself. It is possible, however, to get a reasonable network interface through a normal memory bus. The cost of sending a message could still be low as the processor does not have to wait for the message to actually get out of the chip. The latency of receiving a message, however, cannot be hidden, and may be significant. Pipelined memory systems in modern aggressive microprocessor adds many stages between the processor and the memory bus, and may also introduce the possibility of out-of-order stores. These make sense for executing long sequential threads, but hurt the performance of a bus based messaging interface.

Most commercial parallel machines take a substantial amount of time to get to the network, say 50 cycles or more if external hardware support is available (see CM-5 for example), or 1000's of cycles if software gets involved (see Paragon, for example). This level of processor latency encountered in accessing the network makes communication very expensive and thus, limits compilation options.

3.2.2 Thread Creation

In order to execute threads, they need to be created. When a multithreaded program first starts execution, an initial pool of threads is produced by the startup system code. The running program is then responsible for generating its own threads to get its job done. A mechanism for creating threads is a *fork*, which allows the running thread to split itself and execute down two paths. A minimal encoding of fork will require an instruction pointer which indicates one branch of the fork — the other branch being implicitly the next instruction. Thus, forking can be performed by a single instruction which takes an instruction pointer as an argument. Clearly, the cost of thread creation determines the rate at which parallelism can be expanded. It also determines whether it is worth spawning parallel threads at all.

Stock processors do not have any special support for thread creation. Creating a thread requires software conventions and some sort of thread queue which, on stock hardware, also needs to be managed by software.

3.2.3 Synchronization

When more than one value used for the same computation arrive on separate, arbitrarily ordered messages, the values must be *synchronized*. Synchronization ensures that the necessary values are available before continuing past the synchronization point. In a dataflow graph, each node must synchronize all of the incoming values before it can be fired. Though the frequency of synchronization events vary greatly from program to program, our preliminary compiler work indicates that synchronization events occur around once per 10 abstract dataflow instructions, which would expand to no more than 40 RISC instructions. This makes synchronization a very frequent event.

Synchronization required for multithreading can be emulated with software counters in normal memory locations. We call these locations *synchronization counters*. The essence of synchronization is an atomic read-modify-write to a the memory location, followed by a branch on the result. Synchronization is best viewed as a memory-to-memory operation because it is rare that the same synchronization counter is accessed more than once in a thread. Thus, a typical sequence of instruction to implement a synchronization will include a load, a decrement, a jump conditional and a store, with the additional requirement that atomic access to the counter is maintained between the load and the store. Such a sequence of instruction is not very cheap, particularly when this is expected to happen every 50 or so instructions. Thus, specialized hardware support in this area is highly desirable.

Most stock processors have some hardware support for synchronization, in the form of test-and-set, exchange-reg-with-mem, load-locked/store-conditional, memory-barrier, etc. These instructions essentially enforce some atomicity constraints which would be hard to emulate otherwise. They do not, however, provide the type of synchronization we require. Also, these multi-cycle instructions usually disrupt the processor pipeline. It is probable that emulating synchronization using the scheme discussed in the previous paragraph will be easier and more efficient than using these specialized synchronization instructions.

3.2.4 Context Switching

Whenever a synchronization event fails or a thread terminates, a *context switch* must occur. A context switch conceptually requires the previous state to be saved and the new state to be installed into the processor. Of course, only the state that has been modified and needs to be preserved for later use needs to be saved. The cost of each context switch depends on the size of context to save and restore. This is related to the size of the thread, with shorter threads having smaller states. However, shorter threads also entail more frequent context switching. In general, the lower frequency of context switching for longer threads tend to outweigh the increased cost per switch, making it cheaper to run longer threads. Multithreading with short threads is therefore at a disadvantage.

This disadvantage is aggravated on commercial microprocessors by the principles behind their design, namely, to run long threads and expect infrequent context switches. Accordingly, the processor is optimized to take advantage of that locality with a sophisticated memory hierarchy that makes frequent access to the lower hierarchy a costly event. In some sense, stock processors are designed to make context switching *expensive* in order to process long threads more efficiently.

Faster context switching can take on many different flavors. The simplest technique is to eliminate the memory hierarchy and allow the processor to address a monolithic memory directly. Thus, context switching amounts to loading a program counter and a frame base register. The obvious problem with this is the less competitive speed and bandwidth of the large memory as compared to registers and cache. While very long pipelines and multiple banks of memory could conceivably alleviate this problem, elimination of the local state will probably slow down sequential threads.

Another solution is to have multiple register files, each dedicated to a thread, and to switch between them. One might interleave threads in the pipeline and force a context switch every cycle [37, 3], or one might switch to another thread on encountering a long latency operation [2]. On the University of Tokyo's UNIRED-II processor[35] the frequency of thread switching depends on the number of active threads. One might also allow one register file to be swapped while another is in use. Another possible related solution is the Named State Register File[25] which uses a cache instead of a register file and has multiple pipeline registers to allow very rapid switching between threads. Yet another solution is to permit multiple instruction streams to execute on the same set of execution units. If sufficient load-store unit bandwidth is available, this could allow the overlap of useful computation with context switching overhead, thus effectively masking the cost of context switching.

3.2.5 Thread and Message Scheduling

The pool of ready threads in our model needs to be scheduled for execution in some way. The ordering method could be as simple as LIFO or FIFO or could be very complicated, such as taking the content of the threads into account. As mentioned earlier, a bad schedule could explode the amount of resources required to run the program. Since messages/tokens often contain very small amounts of work, it is generally too expensive to allow software to do much scheduling.

Automatic hardware scheduling of the "next" thread upon thread completion or suspension takes far fewer cycles than any software scheduler. Hardware queues are, however, less flexible in terms of the ways in which the thread queues can be managed. No commercial processor has direct hardware support for thread queues.

3.2.6 Global Memory

Access to global shared memory is a fundamental issue of parallel processing. It is very important for the parallel execution of most high level languages. Thus, we discuss the issues in greater detail in the next section.

3.3 Accessing Global Memory

Researchers generally agree that it is important to support the abstraction of a global shared memory for the programmer even though reality dictates that the memory has to be physically distributed on any scalable machine. There are several ways of supporting the global shared memory abstraction on machines with physically distributed memory. The two basic methods for accessing global memory are (1) protocols built on top of message passing and (2) global caching. The method provided has become the most popular way of differentiating parallel computers. In this section, we will discuss the strengths and weaknesses of these two methods.

3.3.1 Global Shared Memory through Message Passing

Message passing alone allows us to implement certain forms of global shared memory. A processor a which desires the value from a location in the memory of processor b can send a message to b requesting that value. After b receives the request, it replies to a with the desired value.

What does a do while it is waiting for b to respond? It could just wait for the result, not doing anything in the meantime. Just waiting, however, is prone to deadlock, since b could be waiting for a to return a result before it responds to a's request. Either an alternative path for other processors to access b's memory is required, or all processors must periodically satisfy other processors' requests. The former solution is used by *T (discussed in Section 5) and is showing up in commercial machines such as the Cray T3D[16]. This solution works as long as the value to be read is available. The latter solution, periodic polling, makes code generation more difficult and degrades sequential thread performance. Even if the polled message is just put aside, it can still negatively affect the caching behavior of thread in the processor pipeline.

In general, if the value to be read is not yet available, it is necessary to suspend the current thread and schedule another. This requirement stems from the fact that processor a may be required to execute a thread that creates, directly or indirectly, the value being read. If the processor is not relinquished, deadlock is possible. The request and reply messages must contain some information, a continuation, to indicate what should be done with the data when it returns to the requesting processor. Separating the request for a global memory location from its actual use is essentially the dataflow split-phase operation.

Switching threads has another good property; it overlaps computation and communication, potentially increasing processor efficiency. Reading a value from another processor potentially takes more than a hundred cycles in most parallel machines of reasonable size. It involves two message launches, two network trips, execution of two message handlers, a load and perhaps a context switch at the destination. Waiting for every global read will clearly destroy any chances of performance for the vast majority of programs. Switching threads, however, requires a context switch, scheduling a new thread, and other such overhead. If the aggregate overhead of switching a thread is longer than the total latency of remotely accessing the global location, it is not worth switching threads.

It is clear that the performance of global memory through message passing depends heavily on the performance of context switching and the processor/network interface (of course, the basic bandwidth of the network is also important, but is generally not the limiting factor). The faster the context switch, the greater the overlap of communication and computation. If switching cannot be done quickly, communication cannot be overlapped with computation since the latency of remote fetches is effectively added to the total computation time, often negating any advantage of using a parallel machine. Machines that do not have some special hardware for handling memory requests from other processors must context switch twice when doing a split-phase operation, first when the remote processor switches contexts to handle the request and second when the requesting processor switches contexts to accept the response. For these reasons, emulating word-level access to global memory on message passing machines using stock processors is generally inefficient. These characteristics bias the compilers to generate long threads and large messages, hallmarks of coarse-grain parallelism, to achieve acceptable performance.

3.3.2 Global Shared Memory Through Caching

The other basic approach to providing global shared memory is through the use of global caching, *i.e.,* caching global data in computation processors' local caches. The requesting processor a performs a load on a global location exactly as it would a local location; it checks the cache first. If the value is found in the cache, the computation continues as it would in a sequential machine; the value is put into a register directly. If the value is not in the cache, however, the value is fetched from its remote location through the standard memory bus interface of a. There must be some device sitting on the memory bus which recognizes global addresses, satisfying the requests by communicating with the memory of processor b that actually owns the desired value.

In the simplest case, when a misses on a global location in the cache, a waits until the value is returned to it. If a never hits in the cache, the situation becomes exactly the same as for message passing when context switching takes longer than the round-trip message time. Clearly, the more a hits in the cache, the fewer remote fetches of cache data need to be done. Any miss of global values in the cache, assuming that a waits for the value, add the fetch latency to the critical path. The execution time includes all the latency from accesses to global memory.

Since current processors have scoreboarding, they can execute past a load until the loading data is actually needed. Future microprocessors will allow multiple outstanding memory operations. All of these features allow some overlap of communication and computation. How much is actually achievable depends on the program behavior. The potential for overlap, however, is still far less than in the split-phase model.

There are several reasons for stalling the processor on a miss rather than switching threads. The most important reason is that if the processor suspends the waiting thread and switches to another thread, there is no guarantee that the cache line fetched from the remote location will still be in the cache by the time the suspended thread is resumed. Additional mechanisms such as locking a cache line (analogous to locking a page of physical memory during a page fault), or placing the fetched line in some buffer that will not be flushed would be needed. Such schemes need to be careful not to introduce deadlocks. [17] describes how this problem is dealt with in Alewife. A second reason is that unless multiple hardware contexts are provided, thread switching can be expensive, especially since code compiled for such machines usually keeps as much valid state in registers as possible. A third reason is that most code compiled for such machines is not multithreaded at the processor level; there is usually only one single thread running on each physical processor.

The big problem with global caching is keeping the caches coherent. If two processors each have a copy of a cache line, and one modifies that line, the other processor must somehow be automatically notified of the change. If the notification is not automatic, there is no point to general purpose global caching since processors must then always check to see if the cached object is up to date. Some scheme for dealing with the coherency of data is needed (see DASH[20, 19],KSR[6], SCI[33], and Alewife[1] for examples of specific coherency protocols). Cache coherency protocols for managing cache coherency in a distributed memory machine are very complicated. Also, support for global coherent cached memory traditionally requires a fairly sizable investment in hardware.

When there is sufficient locality in the code, global caching works well (see DASH results for examples). Prefetches can also be used to bring data into the cache before they are actually needed. However, the performance is sensitive to the nature of the code, and how code and data is distributed. The same study described examples like the MP3D code in the SPLASH[36] benchmarks where poor locality killed performance.

3.3.3 Split-Phase verses Global Caching

It is clear that both split-phase operations and global caching have their respective advantages and disadvantages. Split-phase operations can hide communication latency but demand fast context switching and network interfaces. Using proper message handlers, messages of any size and I-structure synchronized accesses can be supported. Global caching may reduce the average latency but does not hide it when a cache miss occurs. It is limited to cache-line sized accesses and requires more hardware support than message passing.

Is there a clear winner between the two methods? The short answer is no. We will discuss that question in greater depth in Section 5.4.

3.4 Does Hardware Support for Fine-grain Parallelism help other Programming Models?

It is not difficult to see that all the hardware modifications proposed above would help execution of programs written under other paradigms. The question is by how much. The relative importance of these features crucially depends upon the programs to be run. In parallel computing, much more so than in sequential computing, programmers take into account the characteristics of the machine on which the program is to be run. Thus, it is difficult to determine the importance of each feature simply by studying the behavior of existing parallel applications. In some sense, the other paradigms evolved to run on the available parallel hardware, and thus do not context switch, synchronize or access the network very much. Thus, the impact of the described modifications would be modest for most paradigms other than fine-grain parallel execution.

Context switching and synchronization support only helps if you plan on switching contexts and synchronizing very often. Unless the threads are generated automatically, it is unlikely that a programmer will write a program with a large number of threads or complex synchronization. It is simply not feasible for a user to manage fine-grain parallel execution. Context switching and synchronization support, therefore, are much less important in data parallel and control parallel models.

A fast network interface, on the other hand could potentially help other paradigms a great deal. Every parallel program has to access the network sometime, and if that access time was reduced, less care would be required to minimize communication. Performance would increase on codes that use the network extensively.

Support for shared memory will also help the compiler writers of almost all the high-level languages, including HPF. The global shared memory abstraction greatly simplifies access to distributed memory.

Even if all the features discussed in the previous sections could be included in a modern RISC processor without negatively affecting its performance, it would still be difficult to convince the manufacturer to include these features in his next generation processor. There is never sufficient silicon to include all one wants. However, if a feature, such as a better network interface, is useful for most programming models, its inclusion is more likely to be acceptable.

4 Monsoon: a Proof-of-Concept Dataflow Machine

Monsoon[10, 27, 39] is an existence proof that dataflow hardware is buildable. It incorporates both fast messaging and fast context switching as well as supports fast synchronization, all deemed desirable for fine-grain parallel codes. Monsoon was designed at MIT and built by Motorola. It was designed to run pure dynamic dataflow code, such as Id[21] programs compiled in that style. It is based on an eight-stage pipelined processing node running at ten MHz (see Figure 1). Monsoon's pipeline stages are interleaved like the Denelcor HEP[37] — a single thread can only execute during one out of every eight cycles. The global memory in Monsoon is supported in the form of I-structure boards, each of which contains four million 64-bit words. In addition, presence bits are associated with each word of I-structure memory. The I-structure boards perform the necessary manipulation of presence bits, including the "deferred list" management[10]. Monsoon processors and I-structure boards are connected by a 100 MBytes/second/port

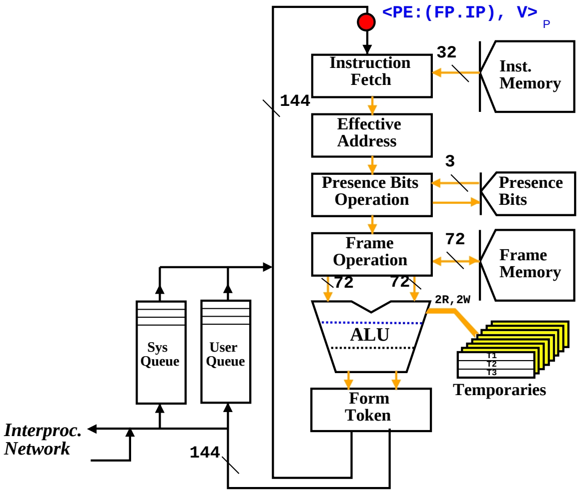

Figure 1: A Monsoon node (taken from [10])

network built out of PaRC switches. The largest configuration built contains eight Monsoon processing elements and eight I-structure boards each. One such system is located at MIT and the other at Los Alamos National Laboratory.

Monsoon demonstrates the feasibility of dataflow architectures as discussed in the previous section. Our performance studies to date show good speedups and cycle counts. For up to the largest machine configuration of 8 processors, Monsoon is able to achieve linear speedup for most programs (see Figure 2). We are able to explain the causes for less than perfect speedup[13]. Also, cycle counts of programs executed on Monsoon are only 3 times that of corresponding fully optimized C/Fortran programs executed on a typical RISC processor (see Figure 3). This demonstrates that with the proper architecture, it is possible to take advantage of fine-grain parallelism without paying an unduly high price.

4.1 Where Monsoon Succeeded

Since Monsoon was designed to execute dataflow code, its performance doing that is excellent. Monsoon's strengths can be categorized as follows:

1. Excellent Processor/Network Interface Integration: Monsoon is capable of sending a message to the network in a single instruction. There are many variations of the send instruction. The basic send instruction takes a *token* which is a tuple containing a continuation (instruction pointer and global frame pointer) and a value. The message is sent to the specified processor and, when scheduled, results in the execution of the continuation specified in the message. Variants of the send instruction allow us to encode operations, such as remote fetch, into a single instruction. For more complicated continuation manipulation, a specific functional unit (called the pointer increment unit) is available. Monsoon's functional unit and token formation unit were optimized for the message operations that would be most common in compiled Id code.

2. Implicit Forks: In order to expose parallelism to keep its pipelines busy and to tolerate latency, Monsoon must be able to fork off work. It does so by allowing instructions to produce up to two tokens in the last stage of the pipeline. This feature allows parallelism to expand quickly, with little added overhead.

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

Figure 2: Speedup Results: Id on Monsoon (taken from [13])

Program	MIPS R3000 (25 MHz)		1PE Monsoon (10 MHz)	
	($\times 10^6$ 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

Figure 3: Performance of Fortran/C on the MIPS R3000 verses Id on 1 processor Monsoon (taken from [13])

Since only one token can reenter the pipeline, one token is designated to be inserted into a token queue. The designation is encoded into the instruction being executed.

3. Fast Hardware Synchronization: Frequent use of cheap messaging and forking requires cheap synchronization. Monsoon has specialized hardware support for synchronization which consists of two components: a few bits of state (called presence bits) associated with each word of memory and a stage of the processor pipeline dedicated to synchronization. Instruction execution can be made conditional upon the presence bits of a memory location referenced by the instruction. Instructions can also alter the presence bits based on the previous values of these bits. For example, if synchronization fails, the presence bits are set and a *bubble* (*i.e.,* a `nop`) is sent down the rest of the pipeline. If synchronization succeeds, the presence bits are cleared and the operation is continued down the pipeline. Bubbles affect performance, but are much more efficient means of synchronization than if the operation was simulated with a sequence of RISC style instructions.

How instructions react to specific presence bit states and how they mutate that state can be changed by a system programmer. Monsoon's instruction set is modifiable because it uses loadable microcode. The microcode is the instruction decode tables for the processor pipeline. See [26] for more details.

4. Fast Context Switch: Monsoon is able to support very short threads because it is able to switch contexts very quickly. Monsoon performs context switching at two levels. First, its interleaved pipeline switches context every cycle, interleaving eight threads at a time, making these context switches free. Tokens that produce result tokens are generally allowed to put one result token back into the pipeline as soon as it is produced, allowing threading. If the current thread does not produce any other tokens, effectively terminating the thread, a token is taken automatically from the token queue for execution. Incoming network tokens are allowed to enter the pipeline at regular intervals, taking advantage of the fact that there is no cost to do so.

Associated with each of the eight threads is an extremely lightweight context, consisting of a single accumulator value (the value carried on the token being processed) and 3 registers. The second level of context switching occurs when a thread ends. If there are useful values in the registers, they have to be explicitly stored away by the thread. This context switch is not free, unlike the cycle-by-cycle interleaving, but the small context makes the switch very cheap nevertheless. Pure dataflow code does not make use of the temporary registers at all; it uses only single-instruction threads. While this means that context switching is free, this style of execution fails to exploit the advantages of state that may be passed through the registers.

5. Hardware Managed Token Queues: Monsoon has hardware managed token queues to store additional tokens that cannot be executed immediately in the pipeline. Additional tokens are produced since an instruction can produce up to two tokens and only one can reenter the pipeline. Also, if the pipeline is completely full, the network is allowed to occasionally interrupt the processor, forcing the executing token to be pushed onto the token queue. There are two queues available which can be accessed in either FIFO or LIFO mode.

6. Support for Global Shared Memory: Global shared memory is supported by split-phase operations which are implemented by fast messaging, fast context switching, and fast synchronization. Split-phase operations are implemented as specialized send instructions, which automatically set the instruction pointer to point to the correct handler. The fast synchronization mechanism of Monsoon quickly integrates the returned value into the computation. Since the messaging and synchronization mechanisms were designed with split-phase operations in mind, they implement the necessary functionality very efficiently and are extremely tolerant of memory latency.

4.2 Where Monsoon Failed

Though Monsoon showed that architectures that run dataflow code well can be built, it has several weaknesses that prevent it from being a truly practical architecture. Many of these defects were intended to simplify implementation and are not fundamental in nature. The weaknesses are described below.

Poor Single Threaded Performance: Monsoon's single thread performance was poor. There are two basic reasons for the poor performance: an interleaved pipeline and a weak instruction set. Each is described more fully below.

- **Interleaved Pipeline:** The latency incurred by any specific thread is eight times the number of instructions to execute. Execution of two contiguous instructions in a thread are separated by 7 cycles where instructions from other threads can be executed. If there is sufficient parallelism in the machine during this time, the overall performance of the machine is not degraded. In most programs, however, there are critical sections where shared resources are accessed. This happens within the run-time system, for instance, and can occur in user code as well. The extra thread latency can constrict the expansion of parallelism since the available parallelism is sequentialized through an expanded critical section. This problem can increase the total run time of a program. It can also increase the processor resources and parallelism required by a program, since 8 threads needs to be active to keep the machine busy.

  The interleaving used by Monsoon was expedient, as it simplified the pipeline. There is no reason why an aggressive implementation could not have a normal, non-interleaved pipeline and yet have the context switching capabilities of Monsoon.

- **Weak Instruction Set:** There are several problems with Monsoon's instruction set, which can be divided between those that are specific implementation problems and those that arise out of more fundamental architectural decisions.

  Among the implementation problems are Monsoon's instruction encoding, particularly its memory addressing modes. These are weak and inflexible, requiring address computation to be done separately from reads and requiring indirect reads to take two cycles. RISC-like 3-address instruction sets are more powerful, as they can specify more general arguments and more general destinations. In fact, the EM-4[30] has exactly that sort of instruction set. Monsoon is also capable of such instructions, but its instruction set is inherently limited in power because the number of registers is very small (3 per thread), and the local memory is not addressable via these registers.

  A more fundamental problem with the instruction set is the coupling of both synchronization and data delivery in the form of a token. This issue is related to the assumption about the frequency of synchronization which will be discussed in the next section. The immediate effect of this duality of token is that it makes it difficult for the compiler to perform certain types of optimizations. For example, if the compiler is able to determine that some synchronization is redundant, it is some times not able to optimize these synchronizations out because the data still needs to be delivered.

Optimized for Synchronization: Monsoon's architecture was optimized for synchronization. Most instructions were expected to synchronize, and thus context switches were expected to be very frequent. This assumption justifies the automatic context switching found in Monsoon, which in turn increases the amount of necessary synchronization.

Due to both the necessity of forking to exploit instruction-level parallelism and the interleaved nature of the pipeline, synchronization is required much more often than would be necessary in a sequential processor that assumes a sequential instruction stream. The pure dataflow model also increases the number of forks, and subsequent synchronizations. Conventional RISC processors, on the other hand, exploit instruction level parallelism within a sequential instruction stream using pipelining and superscalar organization, and do not require explicit synchronization. The required coordination is managed in hardware resulting in a more efficient strategy for exploiting instruction level parallelism.

Since the synchronization section of the pipeline is tightly coupled to the execution units of the pipelines, a miss in the matching section causes a bubble in the execution units. This bubble degrades performance; approximately 10% to 30% of all cycles executed bubble the execution units. The obvious solution of decoupling the matching section from the execution unit is difficult, however, and can still result in bubbles, either in the execution units or matching sections.

Moving more towards a model that has better threading support (by increasing available state) and less support for synchronization would probably have overall benefits. If everything is balanced, that is context switching and synchronization become less frequent as context switching and synchronization costs go up, there would be no difference in performance. The compiler, however, is generally able to reduce synchronization events when it is allowed to thread an instruction stream. This threading will benefit machines optimized for long threads.

Unrealistic Memory Hierarchy: Another artifact of Monsoon's full custom nature is its memory hierarchy, which was implemented using fast static RAMs to save on design time and complexity. Some argue that the hierarchy is unrealistic. One could argue, however, that Monsoon's frame memory, 256 Kwords = 2 MBytes, is *smaller* than caches in modern processors. Monsoon's frame memory can be considered a cache. Since I-structure memory was not cached and resided some 30 cycles away from the processors, it is clear from our performance studies that multithreading allows us to tolerate latency reasonably well.

The presence bit pipeline stage, which read-modifies-writes memory on every cycle is unconventional and may be difficult to build. Some modern RISC processors, however, support read-modify-write for byte writes, proving it may be possible to implement the presence bit stage in a fast pipeline. Until an actual high-speed machine is built with the support we want, it is difficult to say for sure whether it is an obstacle to high-speed implementation.

Little control over thread scheduling: In Monsoon, token scheduling cannot really be controlled by the programmer. The token queues are completely hardware managed; software is not permitted to read or write the token queue. In general, it is a bad idea if any significant storage in the machine is not easily accessible by software. The compiler can decide which token in a two-token-producing instruction gets pushed onto the queue, but there is no way to select the "best" token to execute next. However, executing a "bad" sequence of tokens could dramatically increase run-time or resource usage of the entire program. It would be desirable to have more flexibility in managing the execution order, such as the ability to examine and reorder the token queues.

4.3 Pragmatic Issues

Since Monsoon is a fully custom processor, it cannot leverage commercial processor hardware and software efforts, dooming it to technology generations behind the current state of the art. While this is not a technical point, it is nevertheless an overriding practical consideration. Most parallel computer builders acknowledge the need to use a commercial processor, or a commercial processor with very simple changes, in order to be competitive in performance. Custom processors lag behind commercial processors in raw speed (cycle time).

It is difficult to justify the efforts involved in building experimental machines and software systems if commercially available sequential machines perform better. The time spent doing even simple support software can be substantial; Monsoon's basic software, such as the microcode generator, client/server software and I/O took huge amounts of effort, probably well over 10 man years. Hence, even though there is a greater possibility of speedup for custom processors, their cost in terms of hardware and software development as well as their lower clock speed make them uncompetitive. Thus, it was imperative that our next machine be very close to a conventional processor, allowing us to use the basic support software and requiring only a custom compiler and run-time system.

5 *T: Modified Stock Processors

*T is an attempt to build a truly general building block for a parallel computer, one that will run any type of parallel code reasonably well[24]. It also tries to correct some of the short-comings of pure-dataflow, while at the same time bringing dataflow architecture closer to RISC architecture. The architecture can be viewed as a RISC processor augmented with coprocessors for handling various types of messages. *T supports the multithreaded model which was introduced in the dataflow community towards the end the 80s[15, 23, 31] and gained wide acceptance over the past few years[9, 32, 38, 29].

From a parallel system building view point, using a commercial microprocessor with very limited changes allows the use of software and hardware built for the commercial microprocessor. Changes will have to be made for the parallel machine, but substantial reuse of many components is likely. The effort of the project can thus be focused on relevant research issues rather then spent on bringing up the system from ground zero.

Another motivation for using stock processors is to build a general purpose machine that can easily support other parallel execution paradigms. As most other styles of parallel execution are characterized by fairly coarse grain parallelism with long sequential threads between synchronization points, a RISC style microprocessor will execute them efficiently. Finally, as stated in the previous section, the rapid rate at which stock microprocessors are improving makes it difficult for projects that build custom hardware to demonstrate a clear advantage as they will be obsolete by the time they are built.

5.1 *T: An Architecture for Multithreading

The original *T design logically partitions a node into three parts: the data processor (*dP*), the synchronization processor (*sP*) and the remote memory processor (*RMem*) (see Figure 4). Each processor is responsible for a different aspect of the computation. The synchronization processor's job is to receive messages from the network, write incoming values into frame memory for the data processor, synchronize the values with other input values for waiting threads, and notify the data processor when a thread is ready to execute by posting a continuation. The remote memory processor's job is to handle remote memory requests from other nodes. The data processor's job is to actually execute the user's program.

The *sP* must be able to implement the functionality of four new instructions: `start`, `post`, `next`, and `join`. The `start` instruction starts a new thread in a different context by sending arguments to that new thread. If the context in question is on a different processor, a message is automatically formatted to start the thread on that processor. The `post` instruction takes a continuation (instruction pointer-frame pointer pair) as its argument and puts it into the *dP*'s continuation queue, essentially passing work to the *dP*. The `next` instruction terminates the current thread, finds the next thread in the continuation queue and starts that next thread running. The `join` instruction increments a memory value and conditionally on the resulting value jumps to a given instruction pointer. *sP* may be viewed as a stripped down dataflow processor or a processor designed to execute short threads to completion.

The *RMem* needs to be able to send a message to resume a thread on the requester. *RMem* essentially behaves like the I-structure controller in Monsoon. The *dP* implements the functionality of the `start` and

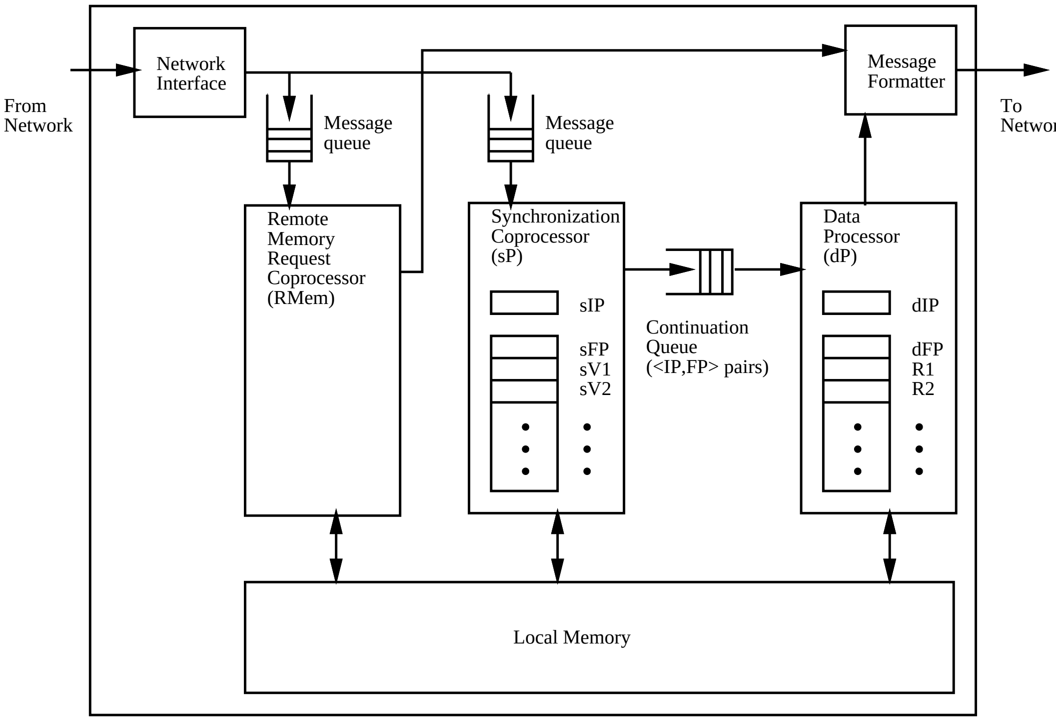

Figure 4: A *T node (taken from [24])

`next` operations as well. It must also be able to initiate remote loads `rload` and stores `rstore`. The `rload` operation takes a global address and a continuation. Its effect is to pass the contents of the address to the continuation. The `rstore` operation is basically the same as the `rload` — an acknowledgment after the store has been completed is sent to the continuation.

This design is motivated from the need to cater to both long and short threads. Fine-grain parallel execution introduces many frequent interrupts in the form of incoming messages from other processors. Stock microprocessors are however optimized to run long threads. To resolve this problem, any functionality that would require the *dP* to switch contexts often has been moved to the *sP* or *RMem*. This allows the functionality requirements of the *dP* to match that of stock microprocessors more closely.

5.2 88110MP: A Realization of *T

The 88110MP *T was based on a modified Motorola MC88110 (88110MP) and the Arctic[5] network routing chip. The 88110MP contains a MC88110 core along with an internal network interface operating as a functional unit and hardware support for thread scheduling. A diagram of an 88110MP processor is shown in Figure 5. The network interface is accessed via register operations, and is thus very fast. It utilizes the MC88110's dual instruction issue capability and wide internal datapath allowing up to 256 bits to be moved into the network buffer each cycle. A simple message could be formatted and launched in 6 cycles. Reading an incoming message is also done with simple register operations, albeit at a slightly lower bandwidth of 128 bits/cycle.

The 88110MP parallel machine consists of *nodes* connected by the Arctic network. The Arctic network is related to the Monsoon network but provides twice as much bandwidth per link (200 MBytes/sec/link). Each node consists of two 88110MP processors sitting on a shared memory bus (see Figure 6) — one node acts as a *RMem* while the other acts both as a *dP* and an *sP*. The remote memory processor would handle

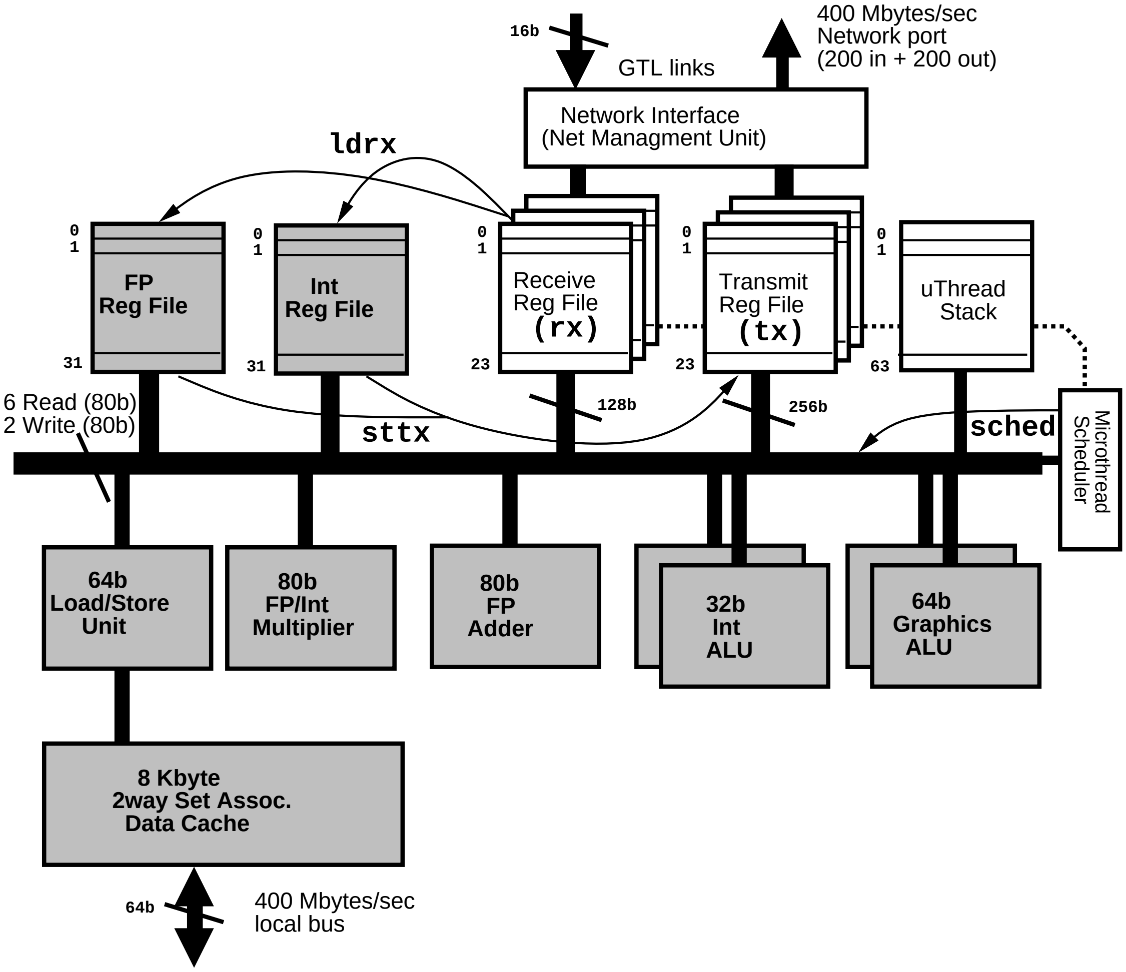

Figure 5: The 88110MP. (taken from [28])

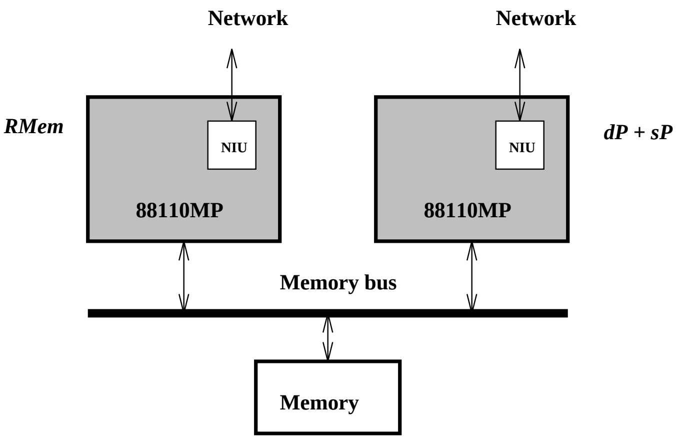

Figure 6: An 88110MP *T node

remote memory requests from other processors. We used a full 88110MP as an *RMem* for simplicity's sake. Since a memory controller, arithmetic operations and network capabilities are required of an *RMem*, it was easier and cheaper, at our quantities, to use a full 88110MP rather than build a separate part. The data processor would read messages from the network, run the message handlers associated with them, and also enqueue and run the ready threads. Of course, this division of labor is arbitrary and can easily be changed.

The 88110MP *T could execute the TAM execution model, based on *active messages*[40], in a fashion similar to the implementation running on the CM-5[9]. *T's superior network interface would give it a significant advantage over the CM-5. Other parallel programming models were also considered and partially implemented for the 88110MP *T. Unfortunately, the 88110MP project was canceled by Motorola because Motorola has shifted their emphasis from the 88110 architecture family to the PowerPC architecture family.

5.3 Evaluation of the 88110MP *T Architecture

In this section we will evaluate the 88110MP *T architecture on the criteria enumerated in Section 3. We will also compare the 88110MP *T to Monsoon. *T and Monsoon are very different machines, and not surprisingly, each is superior in some area. Monsoon is clearly better at switching contexts, synchronization, and accessing the network. *T is much better at single threaded performance. Though *T has fewer of the desirable traits to run multithreaded codes than Monsoon, it allows us to see how well multithreaded code will run on an almost stock processor, and how they should be augmented further.

1. Network Interface: The network interface on the 88110MP *T is much more aggressive then those available on other RISC style machines. Its short message launch time of approximately 6 cycles is an order of magnitude better than the CM-5's 40 cycles[9]. Additional hardware support is provided to aid in message construction. Its performance, however, is still worse than Monsoon's or EM-4's, where short messages are sent in a single cycle.

2. Thread Creation: The 88110MP *T supports the instruction `fork` which pushes a continuation specified in registers onto a continuation stack (described later in this section). It also contains some special features in the network interface that facilitate header construction. Even though Monsoon can fork a thread in fewer cycles than *T, the superior single thread performance of *T reduces the need for forks, as explained below.

In traditional dataflow code, instruction-level parallelism is expressed in the dataflow graph, requiring forking and synchronization. Sequential processors exploit instruction-level parallelism in hardware using a variety of non-trivial mechanisms. Thus, the code can be sequentialized into a thread, eliminating synchronization requirements within a thread. The processor is able to dynamically extract this parallelism. Forking is still required, but not to exploit instruction level parallelism.

3. Synchronization: The 88110MP *T does not have full hardware support for synchronization. It has a conditional post instruction, `cpost`, that conditionally performs a post based on the value of a register. No support, however, is provided for atomic update to a synchronization counter. Just as in thread creation, however, the need for synchronization in the 88110MP is less because of superior single-thread performance. Within a thread, synchronization between each instruction is implicit. Though Monsoon has better synchronization capabilities, it does not necessarily translate into superior overall performance.

*T also does not have hardware supported presence bits for I-structures. It is easy to simulate presence bits by allocating them out of ordinary memory. I-structure-like synchronization can be performed by a *RMem*. This may not be too inefficient because we expect network access overhead and network latency will dominate the extra instructions executed by the *RMem*.

4. Context Switching: No special hardware support is provided in the 88110MP *T for context switching. It adopts a software solution, relying on the compiler to generate code for saving registers value at the end of a thread, and loading registers at the beginning of a thread. The effect of this choice is not clear. On the one hand, it seems that if threads are short, the number of values to be saved or restored for each thread is not very large. Since the save/restore code is compiler generated, only the relevant register values are saved/restored, not the entire register file. On the other hand, fine-grain threads are not expected to be very long, making such switches frequent. Further study, with actual experimental data is needed to see if this is an adequate solution.

Thread switching also requires a switch (jump) in the instruction stream. If threads are short, such switches become frequent. The 88110MP *T has no modifications to its instruction fetch unit. It would have been desirable for the hardware to prefetch instructions for the next thread to be executed, thus eliminating the possible bubbles due to a jump.

The non-interleaved pipeline on the 88110MP is of obvious benefit to code segments that need to be sequential, such as critical sections (especially prevalent in run-time system code). Though 88110MP does not have multiple register contexts, the total number of registers are no less than Monsoon or any other microprocessor. There is always tension, when there are a fixed number of registers, between giving all the registers to one thread or dividing the registers among multiple threads. We do not understand this tradeoff very well, but sequential codes or parallel codes derived from them obviously benefit from having the entire register state.

5. Thread/Message Queue Management: The 88110MP *T has hardware support for managing continuations. A `fork` pushes a continuation onto a stack, while a `sched` instruction selects either an incoming message or the top of the continuation stack. It is also possible to explicitly read the continuation stack, opening the possibility to explicitly control thread scheduling. Scheduling during run-time, however, unless very simple, can become very inefficient very quickly. There is much opportunity for research in this area.

Monsoon's token queues do not require instructions to access, and thus have higher performance. *T's offers greater flexibility, however, which could turn into a great advantage.

6. Global shared memory: The 88110MP *T only supported split-phase access to global memory. `start` instructions were to be used to initiate all phases of a split-phase transaction. *T includes an *RMem* processor that allows access to the remote memory even if the data processor is busy; thus, the dP is not required to poll the network periodically.

Monsoon's support for split-phase operations, however, is superior to the 88110MP *T's support. Monsoon's split-phase operations are primitive instructions; all the information needed for an access are encoded within a single instruction. On *T, the request is a generic message which takes about 6 cycles to construct.

It is, however, not hard to design a RISC style instruction which encodes all the necessary information so that initiating a request takes only 1 instruction. Consider, for example, `rload rGAdd rIP rFP`, which loads from the address specified in `rGAdd`, and invoke a continuation with ip = `rIP` and fp = `rFP` when the value is returned.

Monsoon also has the advantage that for a read, the value that is returned comes in the form of a token, and thus incurs no additional processing cost. *T relegates the task of dealing with the returned value to the sP which will store it into some frame slot. To the extent that the sP is considered "extra hardware" whose cycles are "free", this task incurs no extra cost. However, the value has to be loaded into a register from frame memory before it can be used by the dP. Depending on how code is generated, testing of synchronization counters by the dP may be needed.

The design of *T specifically states that global caching is an orthogonal issue that would not be explored due to time constraints. When the 88110MP was designed, global caching was again ignored for expediency. We feel that global caching has great potential to reduce communication requirements which could reduce critical paths. The next section analyzes global caching and compares it to split-phase operations when solving the DAXPY inner loop.

5.4 Comparing Split-Phase and Global Caching on DAXPY

The DAXPY loop may be written as follows in a high-level language.

```
for i = 0 to N-1 do
  Y[i] = a * X[i] + Y[i]
```

The assembly code for both a uniprocessor version of DAXPY and a multithreaded version of DAXPY are taken from [24] and given in Appendix A. The assembly code for a global cache coherent machine would look very similar to the uniprocessor version.

If we assume the loop is unrolled k times, then in each iteration there will be `cmp + (fmul + fadd + add + add)*`k arithmetic operations, $2k$ `rload`'s and k `rstore`'s, and 1 conditional jump instruction. The multithreaded version, in addition, will have 4 local loads for the loop constants, $2k$ local loads for the actual values of $X[i]$ and $Y[i]$, and 2 local stores to store back pointers to $X[i+k]$ and $Y[i+k]$. It will also have a `next` instruction, which we count as a branch instruction.

The behavior of remote memory references is very different between global caching and split-phase operations. In global caching, a remote memory reference is actually made only when a reference misses in the cache. Let us assume that l is the average latency of accessing global memory through the cache, taking miss ratios into account. Therefore, the total number of cycles for a single iteration of the inner loop on a global cached machine would be

$$(1+4k)+(2kl+k)+1$$

$$=2kl+5k+2$$

We assume that all arithmetic instructions take one cycle. In the split-phase case, every remote memory reference goes to the network. Assuming there is enough parallelism, the processor only sees the number of cycles it must spend formatting and sending the messages. Again, assuming that arithmetic and local memory operations take one cycle, and it takes n cycles to format and send a message, the total number of cycles for a single iteration of the inner loop on a message passing machine would be

$$(1+4k)+(2kn+kn)+1+(4+2k+2)+1$$

$$=3kn+6k+9$$

This information is summarized in Figure 5.4.

To see where *T beats a global cached machine, we simply add up the two rows and see when *T's row is less. For *T to win, the following equation must be true.

$$3kn+6k+9<2kl+5k+2$$

$$3kn+k+7<2kl$$

	Arith	Br	Local		Remote	
			Load	Store	Load	Store
Global cache processor	$1+4k$	1	0	0	$2kl$	k
*T	$1+4k$	2	$4+2k$	2	$2kn$	kn

Figure 7: Instruction cycles in the inner loop of DAXPY unrolled k times, where l is the average latency and n is the cost of sending a message.

Clearly, the performance of the global cached machine is tied directly to its miss ratio and the amount of time it takes to get a global value. If the miss ratio is low and the time to get a global value is reasonable, there is no way for *T to win in terms of the number of cycles. *T can win if l is large or if n is especially small.

Let us assume that $k = 4$. Now the equation can be rewritten to

$$12n + 11 < 8l$$

For *T to beat the global cached machine, $n < 2/3l - 1$. If the miss ratio is an optimistic 1% and the global fetch latency is 100 cycles, there is no way for *T to win. For reasonable values of n, such as 6, l must be more than 10, which would imply miss rates close to 10% on global cached machines with global round-trip latencies of 100 cycles. There is no temporal locality, *i.e.,* if Y is not reused soon, it is very possible for the cache to never hit, favoring *T. If Y is reused, however, *T will most likely lose. If you had a processor where memory accesses can be overlapped, which is very likely in the next generation of processors, then it will be even harder for split-phase transactions to beat global caching in accessing global memory. For stock processors, global caching is likely to win, because n is generally closer to 60 than to 6.

This analysis shows that dynamic program behavior has to be taken into account. This example is too simple to show the full potential of efficient message passing and split-phase transactions. Split-phase operations hide communication latency, but do not reduce communication. Global caching reduces communication, but does not hide communication latency. The former requires some hardware support to run well while the later requires some hardware support to run at all. Clearly the two methods are complementary, solving different problems. A machine with the ability to perform both kinds of remote accesses would be much more flexible, performing better on a larger set of problems, than a machine with just one sort of access. Others have come to exactly the same conclusion[2, 18] as well. Therefore, our next machine, described in the following section, will support both split-phase transactions and global caching.

6 StarT, the Next Generation

We plan to build another parallel machine called *T-NG (StarT, the Next Generation) in collaboration with Motorola. The machine design is heavily influenced by the *T work described in the previous section. The main difference between the original *T and *T-NG, motivated by the analysis presented in the previous section, is the addition of coherent caches for global shared memory. Full support for message passing is to be retained in this new model.

Adding global caches does not create a fundamental shift in our software model for *T. Cache coherency has always been considered an orthogonal issue. Compiler implementations, however, will have to take temporal locality of memory references into account to exploit this new feature. Global locations that are known to be already written and likely to be reused should be accessed through the cache coherency

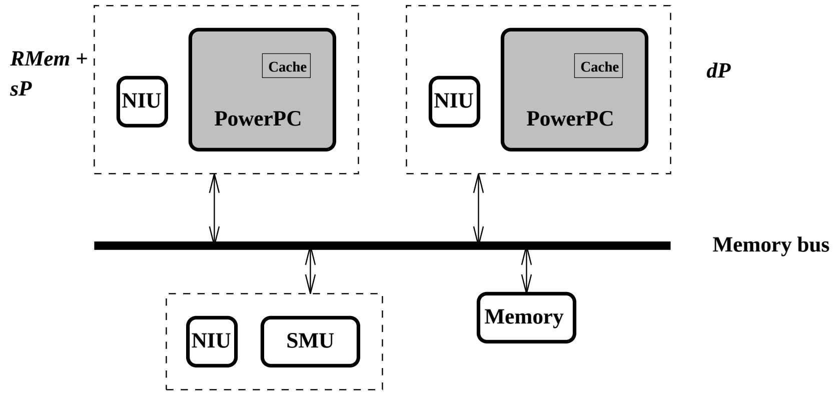

Figure 8: A *T-NG site

interface, while locations that may not have been "filled" should probably be accessed via split-phase operations.

We plan to implement *T-NG within the next two years, using some 64-bit processor from the IBM and Motorola PowerPC family. This choice is natural given our collaboration with Motorola, and the expected cost/performance of these processors. We are constrained to make no modification to either the processor or the operating system. Thus, all the hardware support for the desired features must be built around the processor. The PowerPC architecture has no on-chip network interface. Thus an external network interface unit (NIU) that communicates via existing memory datapaths must be provided.

In order to support global caching, we must provide an external piece of hardware to detect requests for a global location and to satisfy those requests. We call this piece of hardware the Shared Memory Unit (SMU). The logical place for the SMU is off the main memory bus of the processor. For reasons that will become clear shortly, the SMU will also have an NIU for accessing the network.

The basic building block, called a *site*, will consist of several PowerPC processors connected by a snoopy bus to memory and an SMU (see Figure 8). Some of the processors will have NIU's, which may be connected through the second-level cache interface. There are many possible configurations for a site. For discussion purposes, we will assume that a site contains two processors and both processors have an NIU. Furthermore, we will designate one of the two processors as the compute processor (dP) and the other as remote memory processor (*RMem*) which will also provide the functionality of the synchronization processor (sP). An entire processor is not required for an *RMem*. Using a full processor, however, is the simplest and cheapest solution in terms of the engineering effort required, since it already contains the memory bus interface required to access local memory, and has the computation power to run cache coherency protocols and operate on presence bits in software.

Sites will be connected through a network built from the same Arctic[5] chips that we had planned to use in the 88110MP-based *T machine. As stated earlier it will have at least double the bandwidth (200 MBytes/sec/link) as the Monsoon network. The network will probably be configured as a FAT tree.

Before we describe how message passing and global caching will be supported on this machine, we want to emphasize that our approach is minimalist. For example, we will not consider any external custom device whose complexity may approach that of the processor. However, we still have several options for site configurations which we will discuss at the end of this section.

6.1 Message Passing and Split-Phase Transactions on *T-NG

Our basic message passing primitive is the `start` operation. The `start` operation takes a continuation and some arguments and sends those arguments to the processor specified by the continuation and executes that continuation with the arguments. Since we are not permitted to change the processor architecture, `start` must be emulated in software. We can, of course, optimize each instance of the `start` instruction to execute as efficiently as possible in its context. However, the efficiency of the `start` instruction crucially depends upon the overhead of accessing the network through the NIU.

The NIU must be memory mapped since that is the only way to get large amounts of information in and out of the processor. The NIU may have its own memory to provide some message buffering. It will have an input queue and an output queue that allows the processor to send and receive messages respectively. The NIU has to be able to function at the processor's clock rate as well as the network's clock rate.

To execute a `start` operation, the processor must communicate data to the NIU by writing it to the memory mapped locations assigned to the output buffer on the NIU. Once the message is completely written, the processor then writes to some special location to indicate that the message is ready to be sent. Since many modern microprocessors do not preserve ordering on memory operations, care has to be exercised to ensure that the `send` command is not seen by the NIU before the message is fully written to the NIU buffers. Ensuring correct order may require execution of processor synchronization instructions.

On receiving the `send` command, the NIU formats the message for the network and negotiates with its Arctic to send that message. The NIU may support the retransmission of messages to provide fault-tolerance for the network. It may also be able to enforce end-to-end FIFO transmission of user messages.

Receiving a message is trickier than sending one. Either the NIU must interrupt the processor or the processor must poll the NIU sufficiently frequently to avoid buffer overflows. Interrupts in fast processors are generally extremely expensive; thus, we will probably rely on polling to service the network. A software convention, that polls sufficiently frequently, will have to be obeyed by *dP* threads. Since we expect the data processors to execute long threads, even polling can be both expensive and disruptive. In cases when a message is detected, even minimal handling may be disruptive, due to cache thrashing, context switching, etc.

A simple solution to the polling problem is to not accept incoming messages on the *dP*. All messages intended for a *dP* must be sent to the *RMem* associated with that *dP*. If *RMem*'s are programmed to execute only short threads, and they poll the network at the termination of each thread, polling will not be disruptive to the *RMem*. However, the *RMem* has to pass on at least some of the messages to its *dP*. Message passing between processors on the same site can be done via the normal (snoopy) bus-based shared memory. The latency to receive a message would be much higher, but the processor receiving a message would not be forced to poll periodically. Many of the returning messages may not require immediate action, such as those messages that unsuccessfully synchronize. If an *RMem* is delegated to performing those tasks, the overhead on the processor receiving the message is reduced. Each task passed from the *RMem* to the *dP* thus has more "weight" and will probably be, on average, a longer thread. The problem with this scheme is that thread operands must be copied at least once, from the *RMem*'s cache to the *dP*'s cache, thus increasing overall overhead.

Split-phase operations are special cases of the `start` instruction. Essentially, a split-phase operation is a `start` operation from the request processor to the home processor, and another `start` operation from the home processor back to the request processor. Split-phase handlers must be written to provide the desired functionality. Active Messages[40] provide a good abstraction for handling all incoming messages, including split-phase operations. An Active Message contains an instruction pointer for a handler which will process the message.

6.2 Global caching on *T-NG

To implement shared memory, we will assume two things: the address space, both virtual and physical, is partitioned into local and global memory, and every global address has an owner site, which we will call the home site of the address. For simplicity, we will only consider the case where the home node of a global memory location does not change during the execution of a program. However, it should be possible to build software/OS that supports migration of global memory addresses at the page level.

The SMU's together with the *RMem*'s implement global caching. The SMU detects a memory operation to global shared memory, and carries out the necessary action to complete the operation. It must be able to capture a global address, and format and send the message to fetch/store the requested/supplied data. Most of the processing needed to run cache coherency protocols is done in software on the *RMem* at the home site. To the first order, the complexity of the SMU is not related to the complexity of the cache coherence protocols.

The *RMem* maintains the coherence directory for all the addresses it owns, runs most of the coherence protocol, and supplies the requested data or performs the requested write. Memory for keeping the coherence directory will be part of the user's virtual memory. Implementing the coherence protocols in software will allow us to fix bugs and to experiment with different coherence protocols.

If a global address makes it to the memory bus, the corresponding memory operation could not have been satisfied in the on-chip cache or the second level cache. Completing the operation requires sending a request to the *RMem* that owns the cache line. If a reply to the operation is expected, the SMU receives the reply and puts it onto the processor bus.

The SMU behaves like a cache on the processor bus. In addition to the tasks mentioned earlier, it keeps track of operations in progress, and associates some state with each active operation to keep the cache coherent protocols manageable. The SMU will also handle cache coherence requests arriving from the network from the various *RMem*'s, such as requests to flush/invalidate cache copies of global data. Because of the need to send and receive remote requests, the SMU will need network access and will be provided with its own NIU.

Implementing Suspendable Global Loads We hope to extend the functionality of our global coherent caching to allow a thread switch in case of a cache miss. A cache miss would effectively be turned into a split-phase transaction; the current thread would be suspended until the value is returned and another thread is scheduled in its place. We call this hybrid approach to caching the *suspending* approach, and name the corresponding load instruction `sgload`. Converting a cache miss to a split-phase transaction will allow us to cache locations of I-structure memory as well, and will provide a better means of masking latency. Without such a feature, in general, it is not possible to cache I-structure memory. Of course, it would be desirable to have the option of using normal, blocking cache coherency as well as normal split-phase transactions. Alewife[1] already has a feature that allows thread switching on a global cache miss, but since the switching is done in hardware, only a small constant number of switches can occur. Our scheme will allow the user to retain control over thread switching.

The SMU will support the `sgload` instruction in the simplest fashion possible. If the SMU can notify the user code when a miss has occurred, the software on the requesting processor can take all the other required actions. Since the processor cannot tell if its `sgload` request was satisfied from the cache or through the SMU, a predefined pattern will be used by the SMU to indicate a missing value. The software tests the returned value against the predefined pattern, and switches the thread in case a missing value is detected. If the processor initiates a split-phase transaction for the missing value, the scheme can be made to work even if the predefined pattern is the actual content of the requested location, since the handler for the return of the value will not check the value against any pattern. The main cost is an unnecessary global cache line fetch. With the proper selection of the predefined pattern, we hope to avoid this problem

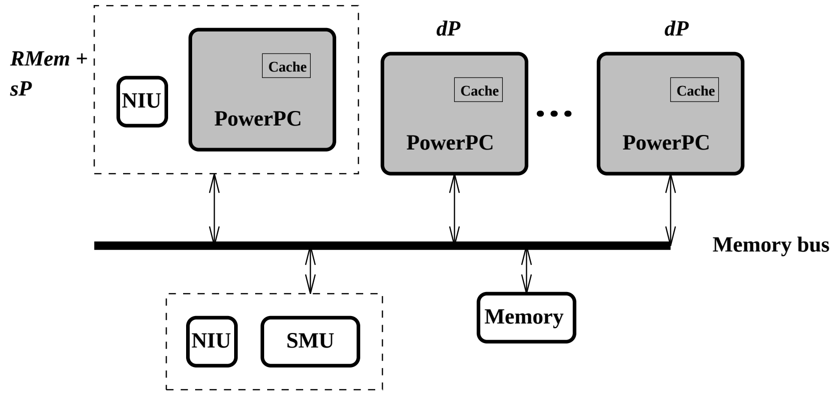

Figure 9: An alternative *T-NG site

all together.

Another possible implementation of the `sgload` instruction could have the SMU interrupt the processor when a miss occurs. We expect, however, that interrupts will be quite expensive. This solution also requires the SMU has the ability to interrupt the processor. If the miss ratio is very low, however, this scheme will probably perform better than the first one.

More sophisticated implementations of `sgload` seem to require much more hardware support on the processor side. In one extreme the processor can provide the illusion of multiple, unlimited number of register contexts, and switch among them transparently. However, since we cannot modify the processor, we will not explore these variations here.

Our `sgload` implementation is very much a hack made necessary by our need to use a stock microprocessor. Performance will be far from optimal, since each `sgload` must be checked at some point to make sure the obtained value is valid. It is also possible for the test to fail, if the value is actually the test pattern. The implementation, however, will allow our ideas for caching I-structures to be tried without changes to the microprocessor core and the on-chip cache.

6.3 Alternative Site Configurations

Up to this point, we have assumed that there are exactly two processors per site and that both processors have NIU's. It is possible that there may be more processors per site, and that not all processors will have NIU's. If there are more than two processors per site, but all processors still have NIU's, *T-NG can run pretty much the same as described in the previous sections. The correct mix of *dP*'s and *RMem*'s will have to be determined, but the ratio is changeable in software.

If there are processors that do not have an NIU, certain problems arise. Such a configuration is shown in Figure 9. There must be at least one processor per site that has an NIU to communicate with other processors. *RMem*'s must have a NIU, to perform their duties; thus, at least one of the processors with an NIU will be assigned to being the *RMem*. The SMU's NIU will probably be usable as well in this case. A processor without an NIU cannot send or receive messages directly, and must rely on an *RMem* or the SMU for messaging capabilities. All message traffic must go through the snoopy bus, however, further degrading performance. A processor that wants to send out a message in this case would write out a message using normal snoopy bus memory writes, then writing a "send" signal, again on the bus, to indicate that the message is fully formatted and ready to send. Again, writing out messages from a processor to the snoopy bus is not straightforward, due to the fact that the issue order of the writes may not be the order in which the stores are seen on the bus.

7 Conclusion

In this paper we have examined the problems of supporting fine-grain parallelism on a machine built out of stock processors. We have discussed the lessons we learned from the Monsoon project. Monsoon was a fully custom, pure dataflow machine. Though it was very successful running fine-grain parallel code, for pragmatic reasons we were forced to move to machines built from stock processors. We also briefly presented the *T model which was motivated by the desire to create a better building block for parallel machines. We then discussed our first implementation of a *T node which was based on a Motorola 88110 RISC processor. This project was abandoned in favor of a move towards the PowerPC architecture family. We took the opportunity to incorporate global caching into the *T model. The resulting model is called *T-NG for *T, the Next Generation.

There are many machines that have much in common with *T-NG. Japan's Electrotechnical Laboratory's (ETL) EM-4[30] and its descendant architectures are custom machines designed to execute multithreaded code. In fact, those architectures are far more aggressive since their network interfaces are embedded into the processing pipeline itself. A message can be sent in a single instruction. Synchronization is built into the processor as well. *T-NG's hardware support for messaging is far less aggressive, since we use stock processors, but it will use state-of-the-art processing elements and will support global caching.

The machines that are most similar to *T-NG are MIT's Alewife[1] and Stanford's FLASH[18]. Both machines support global cached memory as well as message passing, and both are based on stock processors. Alewife's processor, is modified to some degree for better support of cache coherency. It has greater support for user-level interrupts, including a division of the Sparc register set into four parts instead of using Sparc register windows, and has a few extra signalling pins. FLASH uses one stock processor and one very heavily modified processor (called MAGIC) to build a processing node. It is important to note that our implementation is far less aggressive than FLASH and, in many ways, Alewife. We plan to use completely stock processors running on a stock memory bus which will hopefully reduce development time relative to Alewife and FLASH. It will be interesting to see how close to FLASH and Alewife in performance *T-NG can get.

It is exciting to see that parallel architectures are converging. The machines mentioned in the previous paragraph differ only in detail from *T-NG. They all attempt to integrate shared memory with message passing, and are therefore significantly different from pure message passing machines, such as the J-machine[11], dataflow machines like Monsoon, and pure shared memory machines such as DASH. Eventual success of these attempts will depend very much on the quality of the hardware implementations, and the system software that is provided for using these machines.

8 Acknowledgments

We would like to thank R.S. Nikhil and Greg Papadopoulos for permitting us to use diagrams and codes from their papers in this document. We would also like to thank Jamey Hicks, R.S. Nikhil, Shail Aditya, R. Paul Johnson, and Yuli Zhou, for reading various drafts of this paper and offering helpful comments.

The discussion on *T-NG has involved many people, both within our group and at Motorola. We have had very extensive discussions with Mike Beckerle, Bob Greiner and Jamey Hicks at Motorola and with Greg Papadopoulos, James Hoe, Chris Joerg, and Andy Boughton at MIT.

This paper 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 N00014-92-J-1310.

A Appendix: DAXPY Assembly code

The loop compiled to uniprocessor code looks like this. (This code segment is lifted nearly verbatim from [24].)

```
  load  rXP, dFP[XP]      -- load ptr to  X
  load  rYP, dFP[YP]      -- load ptr to  Y
  load  rA,  dFP[A]       -- load loop const: a
  load  rYlim, dFP[YLim]  -- load loop const: Y  ptr limit
  cmp   rB,rYP,rYLim      -- compare ptr to  Y  with limit
  jgt   rB, OUT           -- zero-trip loop if greater

LOOP:
  load  rXI, rXP          -- X[i]  into rXI   (L1)
  load  rYI, rYP          -- Y[i]  into rYI   (L2)
  add   rXP,rXP,8         -- incr ptr to X
  fmul  rTmp,rA,rXI       -- a*X[i]
  fadd  rTmp,rTmp,rYI     -- a*X[i] + Y[i]
  store rYP, rTmp         -- store into Y[i]   (S1)
  add   rYP,rYP,8         -- incr ptr to Y

  cmp   rB,rYP,rYLim      -- compare ptr to  Y  with limit
  jle   rB, LOOP          -- fall out of loop if greater

OUT:
   ... loop sequel ...
```

The loop compiled to *T code looks like this. (This code segment is again lifted nearly verbatim from [24].)

```
;;
;;  Synchronization Processor Message Handlers
;;
L1_S:
  store sFP[XI],   rV1    -- store away incoming  X[I]
  join  c1, 2, CONTINUE_D -- attempt continuation of loop
  next                    -- next message

L2_S:
  store sFP[YI],   rV1    -- store away incoming  Y[I]
  join  c1, 2, CONTINUE_D -- attempt continuation of loop
  next                    -- next message

S1_S:
  load  rN, sFP[N]        -- total number of stores
  join  c2, rN, OUT_D     -- sequel when all stores complete
;;
;;  Data Processor Threads
;;
  load  rXP, dFP[XP]      -- load ptr to  X
  load  rYP, dFP[YP]      -- load ptr to  Y
  load  rYlim, dFP[YLim]  -- load loop const: Y  ptr limit
  cmp   rB,rYP,rYLim      -- compare ptr to  Y  with limit
  jgt   rB, OUT_D         -- zero-trip loop if greater

LOOP_D:
  rload rXP, L1_S         -- initiate load  X[i]  (L1)
```

```
  rload rYP, L2_S          -- initiate load Y[i]  (L2)
  next

CONTINUE_D:
  load  rA,  dFP[A]        -- load loop const: a
  load  rXP, dFP[XP]       -- load ptr to  X
  load  rYP, dFP[YP]       -- load ptr to  Y
  load  rYLim, dFP[YLim] -- load loop const: Y  ptr limit

  load  rXI, dFP[XI]       -- load copy of  X[I]
  load  rYI, dFP[YI]       -- load copy of  Y[I]

  fmul  rTmp,rA,rXI        -- a*X[i]
  fadd  rTmp,rTmp,rYI      -- a*X[i] + Y[i]
  rstore rYP, rTmp,S1_S    -- store into Y[i]  (S1)
  add   rXP,rXP,8          -- increment ptr to X
  add   rYP,rYP,8          -- increment ptr to Y
  store dFP[XP],    rXP    -- store ptr to  X
  store dFP[YP],    rYP    -- store ptr to  Y

  cmp   rB,rYP,rYLim       -- compare ptr to  Y  with limit
  jle   rB, LOOP_D          -- fall out of loop if greater
  next

OUT_D:
   ... loop sequel ...
```

References

[1] A. Agarwal, D. Chaiken, G. D'Souza, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B. H. Lim, G. Maa, D. Jussbaum, M. Parkin, and D. Yeung. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In *Proceedings of Workshop on Scalable Shared Memory Multiprocesors*. Kluwer Academic Publishers, 1991.

[2] A. Agarwal, J. Kubiatowicz, D. Kranz, B.-H. Lim, D. Yeung, G. D'Souza, and M. Parkin. Sparcle: An Evolutionary Processor Design for Large-Scale Multiprocessors. *IEEE Micro*, Jun 1993.

[3] R. Alverson and et. al. The Tera Computer System. In *Proceedings of the International Conference on Supercomputing*, pages 1–6, Jun 1990.

[4] 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.

[5] A. Boughton and et. el. Arctic User's Manual. CSG Memo 353, Laboratory for Computer Science, MIT, Feb 1994.

[6] H. Burkhardt III, S. Frank, B. Knobe, and J. Rothnie. Overview of the KSR1 Computer System. Technical Report KSR-TR-9202001, Kendall Square Research, Boston, MA, Feb 1992.

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

[8] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. v. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proc. Supercomputing 93, Portland, OR*, Nov 1993.

[9] D. E. Culler, S. C. Goldstein, K. E. Schauser, and T. von Eicken. TAM — A Compiler Controlled Threaded Abstract Machine. *Journal of Parallel and Distributed Computing*, 18(3):347 – 370, 1993.

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

[11] W. J. Dally and et al. Architecture of a Message-Driven Processor. *IEEE Micro*, 12(2):23–39, 1992.

[12] A. Geist and et. el. PVM 3 User's Guide and Reference Manual. Technical Report 12187, ORNL, May 1993.

[13] J. Hicks, D. Chiou, B. S. Ang, and Arvind. Performance Studies of the Monsoon Dataflow Processor. *Journal of Parallel and Distributed Computing*, 18(3):273 – 300, 1993.

[14] R. A. Iannucci. Toward a Dataflow/von Neumann Hybrid Architecture. In *Proceedings of 15th Annual International Symposium on Computer Architecture, Honolulu, HI*, pages 131–140, May 1988.

[15] R. A. Iannucci. *Parallel Machine, Parallel Machine Languages: The Emergence of Hybrid Dataflow Computer Architectures.* Kluwer Academic Publishers, 1990.

[16] R. E. Kessler and J. L. Schwarzmeier. Cray T3D: A New Dimension for Cray Research. In *CompCon93*, pages 176–182, Feb 1993.

[17] J. Kubiatowicz, D. Chaiken, and A. Agarwal. Closing the Window of Vulnerability in Multiphase Memory Transactions. In *Proceedings of the Fifth International Conference on Architectureal Support for Programming Languages and Operating Systems, Boston, MA*, Oct 1992.

[18] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH Multiprocessor. In *to appear in ISCA 94, Chicago, IL*, 1994.

[19] D. Lenoski, J. Laudon, K. Charachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocesor. In *Proceedings of The 17th Annual International Symposium on Computer Architecture, Seattle, WA*, pages 148–159, 1990.

[20] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH Prototype: Implementation and Performance. In *Proceedings of The 19th Annual International Symposium on Computer Architecture, Gold Coast, Australia*, pages 92–103, May 1992.

[21] R. S. Nikhil. Id Reference Manual, Version 90.1. CSG Memo 284-2, Laboratory for Computer Science, MIT, Cambridge MA, Sep 1990.

[22] R. S. Nikhil. The Parallel Programming Language Id and its Compilation for Parallel Machines. *Intl. J. of High Speed Computing*, 5(2):171–223, 1993.

[23] R. S. Nikhil and Arvind. Can Dataflow Subsume von Neumann computing? In *Proceedings of the 16th. Annual International Symposium on Computer Architecture, Jerusalem, Israel*, pages 262–272, May 29-31 1989.

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

[25] P. R. Nuth. *The Named-State Register File*. PhD thesis, Department of EECS, MIT, Cambridge MA, Aug 1993.

[26] G. M. Papadopoulos. Program Development and Performance Monitoring on the Monsoon Dataflow Multiprocessor. In M. Simmons, R. Koskela, and I. Bucher, editors, *Instrumentation for Future Parallel Computing Systems*. Addison-Wesley Publishing Company, 1989.

[27] G. M. Papadopoulos. *Implementation of a General-Purpose Dataflow Multiprocessor*. MIT Press, 1992.

[28] G. M. Papadopoulos, G. A. Boughton, R. Greiner, and M. J. Beckerle. ⋆T: Integrated Building Blocks for Parallel Computing. In *Proceedings of Supercomputing '93, Portland, OR*, Nov 1993.

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

[30] 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, May 1989.

[31] 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, May 1992.

[32] K. E. Schauser, D. E. Culler, and T. von Eicken. Compiler-Controlled Multithreading for Lenient Parallel Languages. In *Proceedings of the Symposium of Functional Programming Languages and Computer Architecture, Cambridge, MA*, pages 50–72, 1991.

[33] *IEEE Standard for Scalable Coherent Interface (SCI)*. IEEE, 345 East 47th Street, New York, NY 10017, USA, 1993. (IEEE Std 1596-1992).

[34] A. Shaw, Y. Kodama, M. Sato, S. Sakai, and Y. Yamaguchi. Performance of Data-Parallel Primitives on the EM-4 Dataflow Parallel Supercomputer. In *Proceedings of Frontiers '92: The 4th Symposium on the Frontiers of Massively Parallel Computation, McLean, VA*, pages 302–309, Oct 1992.

[35] K. Shimada, H. Koike, and H. Tanaka. UNIREDII: The High Performance Inference Processor for the Parallel Inference Machine PIE64. In *Proceedings of the International Conference of Fifth Generation Computer Systems*, pages 715–722, 1992.

[36] J. P. Singh, W. D. Weber, and A. Guta. SPLASH: Stanford Parallel Applications for Shared Memory. Technical Report CSL-TR-91-469, Stanford University, 1991.

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

[38] K. R. Traub. Multi-thread Code Generation for Dataflow Architectures from Non-Strict Programs. In *Proceedings of the Symposium on Functional Programming Languages and Computer Architecture, Cambridge, MA*, pages 73–101, 1991.

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

[40] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture, Gold Coast, Australia*, pages 256–266, May 1992.

Contents