Appears in the Proceedings of MICRO-28.

The M–Machine Multicomputer

Marco Fillo fillo@ai.mit.edu
Stephen W. Keckler skeckler@ai.mit.edu
William J. Dally billd@ai.mit.edu

Nicholas P. Carter npcarter@ai.mit.edu
Andrew Chang achang@ai.mit.edu
Yevgeny Gurevich yev@ai.mit.edu
Whay S. Lee wslee@ai.mit.edu

Artificial Intelligence Laboratory
Laboratory for Computer Science
Massachusetts Institute of Technology
545 Technology Square
Cambridge, MA 02139

Abstract

The M–Machine is an experimental multicomputer being developed to test architectural concepts motivated by the constraints of modern semiconductor technology and the demands of programming systems. The M–Machine computing nodes are connected with a 3–D mesh network; each node is a multithreaded processor incorporating 12 function units, on-chip cache, and local memory. The multiple function units are used to exploit both instruction-level and thread-level parallelism. A user accessible message passing system yields fast communication and synchronization between nodes. Rapid access to remote memory is provided transparently to the user with a combination of hardware and software mechanisms. This paper presents the architecture of the M–Machine and describes how its mechanisms attempt to maximize both single thread performance and overall system throughput. The architecture is complete and the MAP *chip, which will serve as the M–Machine processing node, is currently being implemented.*

1 Introduction

Because of the increasing density of VLSI integrated circuits, most of the chip area of modern computers is now occupied by memory and not by processing resources. The M–Machine is an experimental multicomputer being developed to test architectural concepts motivated by these constraints of modern semiconductor technology and the demands of programming systems, such as faster execution of fixed sized problems and easier programmability of parallel computers.

Advances in VLSI technology have resulted in computers with chip area dominated by memory and not by processing resources. The normalized area (in λ^2) of a VLSI chip[1] is increasing by 50% per year, while gate speed and communication bandwidth are increasing by 20% per year [14]. As a result, a 64-bit processor with a pipelined FPU ($400\mathrm{M}\lambda^2$)[2] is only 8% of a $5\mathrm{G}\lambda^2$ 1996 0.35μm chip. In a system with 256 MBytes of DRAM, the processor accounts for 0.13% of the silicon area in the system. The memory system, cache, TLB, controllers, and DRAM account for most of the remaining area. Technology scaling has made the memory, rather than the processor, the most area–consuming resource in a computer system.

To address this imbalance, the M–Machine increases the fraction of chip area devoted to processor, making better use of the critical memory resources. An M–Machine multi-ALU processor (MAP) chip contains four 64-bit three-issue *clusters* that comprise 32% of the $5\mathrm{G}\lambda^2$ chip and 11% of an 8 MByte (six-chip) node. The multiple execution clusters will provide better peak performance than using a single cluster and a large on-chip cache in the same chip area. The high ratio of arithmetic bandwidth to memory bandwidth (12 operations/word) allows the MAP to saturate the costly DRAM bandwidth even on code with high cache-hit ratios. A 32-node M–Machine system with 256 MBytes of memory has 128 times the peak performance of a 1996 uniprocessor with the same memory capacity at 1.5 times the area, a 85:1 improvement in peak performance/area. Even at a small fraction of this peak performance, such a machine allows the costly, fixed-sized memory to handle more problems per unit time resulting in more cost-effective computing.

The M–Machine is intended to extract more parallelism from problems of a fixed size, rather than requiring enormous problems to achieve peak performance. To do this, nodes are designed to manage parallelism at a variety of granularities, from the instruction level to the process level. The 12 function units in a single M–Machine node are controlled using a form of Processor Coupling [18] to exploit instruction level parallelism by executing 12 operations from the same thread, or to exploit thread-level parallelism by executing operations from up to six different threads. The fast internode communication allows collaborating threads to reside on different nodes.

The M–Machine also addresses the demand for easier programmability by providing an incremental path for increasing parallelism and performance. An unmodified sequential program can

*The research described in this paper was supported by the Advanced Research Projects Agency and monitored by the Air Force Electronic Systems Division under contract F19628-92-C-0045.

[1] The parameter λ is a normalized, process independent unit of distance equivalent to one half of the gate length [23]. For a $0.5\mu m$ process, λ is $0.25\mu m$.

[2] Area was determined by measuring the processing components of various chips, in particular the R4600 described in [12].

run on a single M–Machine node, accessing both local and remote memory. This code can be incrementally parallelized by identifying tasks, such as loop iterations, that can be distributed both across nodes and within each node to run in parallel. A flat, shared address space simplifies naming and communication. The caching of remote data in local DRAM automatically migrates a task's data to exploit locality.

Previous publications have introduced some of the mechanisms used in the M–Machine. The first description of Processor Coupling, a method for exploiting instruction level parallelism, appeared in [18]. The novel capability-based memory protection system of the M–Machine was described in [5]. This paper describes the M–Machine's other features which include an improved form of Processor Coupling as well as communication and global addressing mechanisms. The M–Machine architectural design is complete and the MAP chip, which will serve as the M–Machine processing node, is currently being implemented.

Section 2 gives an overview of the machine architecture, including the physical resources of the M–Machine. Section 3 describes the updated version of Processor Coupling that simplifies hardware implementation and is expected to improve performance. Instead of lock-step execution of the wide instruction words across all of the function units, an instruction stream is partitioned by the compiler into horizontal threads (*H–Threads*), which run concurrently on different execution clusters to exploit instruction level parallelism. Synchronization between them is performed explicitly using instructions that write to one another's register files. In addition, the function units are time-shared among vertical threads (*V–Threads*) which exploit runtime parallelism and mask pipeline, memory, and communication latencies. Events are handled asynchronously in a dedicated *V–Thread* so that event handling may proceed in parallel with user program execution and the issued instructions of the thread that caused the event need not be cancelled. Section 4 discusses inter-node communication including the user-level communication primitives, the global mapping of virtual addresses to physical memory and to remote processors, and how they are used to provide global coherent memory access. Finally, Section 5 describes the M–Machine software effort, including a brief overview of the compiler and runtime system.

2 M–Machine Architecture

The M–Machine consists of a collection of computing nodes interconnected by a bidirectional 3-D mesh network. Each six-chip node consists of a multi–ALU (MAP) chip and 1 MW (8 MBytes) of synchronous DRAM (SDRAM). The MAP chip includes the network interface and router, and it provides an equal bandwidth of 800 MBytes/s to the local SDRAM and to each network channel. Each node contains a dedicated I/O bus; I/O devices may be connected to either every node or a subset of nodes, for example, all nodes on a face of the mesh. The target clock rate for the MAP is 100MHz.

As shown in Figure 1, a MAP contains four execution clusters, a memory subsystem comprised of four cache banks and an external memory interface, and a communication subsystem consisting of the network interfaces and the router. Two crossbar switches interconnect these components. Clusters make memory requests to the appropriate bank of the interleaved cache over the 150-bit wide

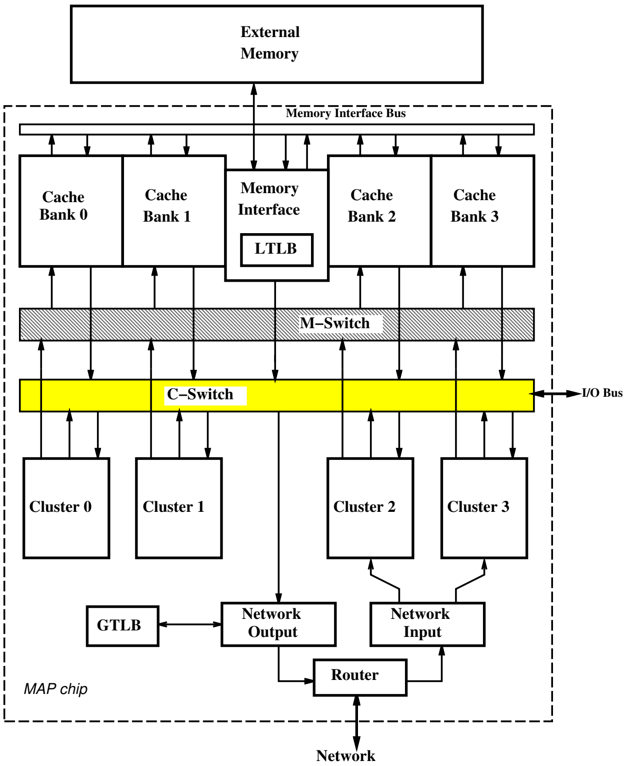

Figure 1: The MAP architecture.

(address+data) 4×4 M–Switch. The 90-bit wide 10×4 C–Switch is used for inter-cluster communication and to return data from the memory system. Both switches support up to four transfers per cycle, one transfer per output port.

MAP Execution Clusters: Each of the four MAP clusters is a 64-bit, three-issue, pipelined processor consisting of two integer ALUs, a floating-point ALU, associated register files, and a 1KW (8KB) instruction cache, as shown in Figure 2. One of the integer ALUs in each cluster, termed the memory unit, is the interface to the memory system. Each MAP instruction contains 1, 2, or 3 operations, zero or one for each ALU. All operations in a single instruction issue together but may complete out of order. Every operation may be conditionally executed depending on the one-bit value of one of the condition code registers.

Memory System: As illustrated in Figure 1, the 128KB on–chip cache is organized as four word-interleaved 4KW (32KB) banks to permit accesses to consecutive addresses to proceed in parallel. The cache is virtually addressed and tagged. The cache banks are pipelined with a three-cycle read latency, including switch traversal.

The external memory interface consists of the SDRAM controller and a local translation lookaside buffer (LTLB) used to cache local page table (LPT) entries. Pages are 512 words (64 8-word cache blocks). The SDRAM controller exploits the pipeline and page modes of the external memory and performs single error correction and double error detection on the data transferred from

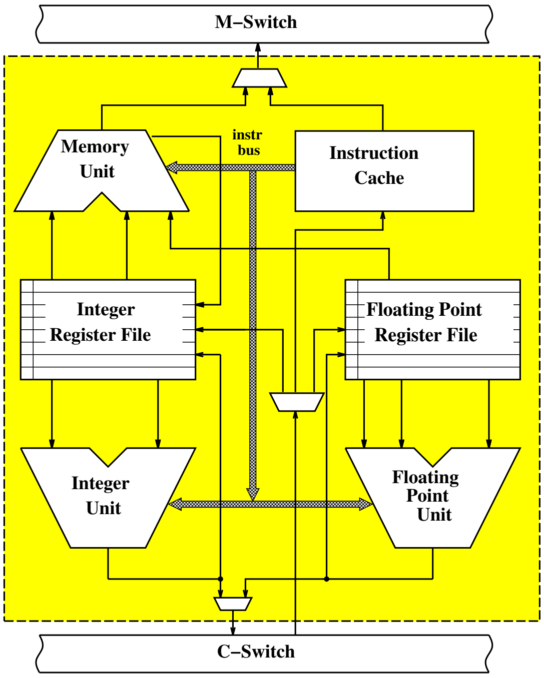

Figure 2: A MAP cluster consists of 3 execution units, 2 register files, an instruction cache and ports onto the memory and cluster switches.

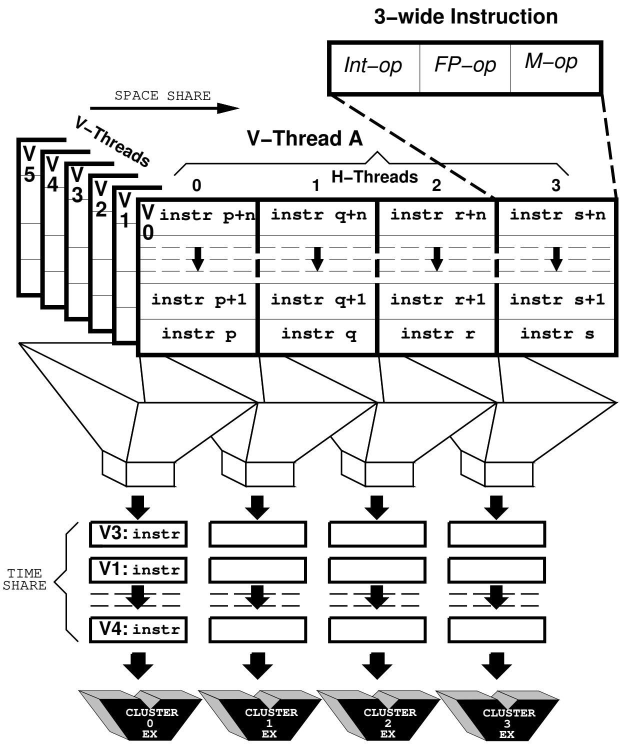

Figure 3: Multiple V-Threads are interleaved dynamically over the cluster resources. Each V–Thread consists of 4 H–Threads which execute on different clusters.

external memory.

A synchronization bit is associated with each word of memory. A pair of load and store operations specify a precondition and a postcondition on the synchronization bit and are used as atomic read-modify-write memory operations.

The M–Machine supports a single global virtual address space. A light-weight capability system implements protection through guarded pointers [5], while paging is used to manage the relocation of data in physical memory within the virtual address space. The segmentation and paging mechanisms are independent so that protection may be preserved on variable-size segments of memory. The memory subsystem is integrated with the communication system and can be used to access memory on remote nodes, as described in Section 4.2.

Communication Subsystem: Messages are composed in the general registers of a cluster and launched atomically using a user-level SEND instruction. To provide protection, messages must be sent to virtual addresses which are automatically translated into physical node identifiers via a global translation lookaside buffer (GTLB). The GTLB caches entries of a software global destination table (GDT), much like a TLB caches page table entries. Arriving messages are queued in a register-mapped hardware FIFO readable by a system-level message handler. Two network priorities are provided, one each for requests and replies. Messages are routed in dimension order using up to four virtual channels.

3 Intra–node Concurrency Mechanisms

The amount and granularity of parallelism varies enormously across application programs and even during different phases of the same program. Some phases have an abundance of instruction level parallelism that can be extracted at compile time. Others have data dependent parallelism that can be only be exploited using multiple threads; the task size to achieve maximum concurrency may vary widely.

The M–Machine is designed to efficiently execute programs with either compiler or runtime scheduled parallelism, and with a range of granularities. The M–Machine architecture contains two mechanisms for intra-node concurrency: Vertical Threads (V–Threads) and Horizontal Threads (H–Threads). A *V–Thread* is similar to a standard process and is composed of up to four H–Threads. An *H–Thread* is a 3-wide instruction stream which is statically scheduled and executes on a single MAP cluster. The H–Threads of the same V–Thread can be either independently scheduled or scheduled together by the compiler to achieve 12-wide instruction level parallelism (ILP).

The MAP has sufficient hardware resources to support up to six resident V–Threads; on each cluster, constituent H–Threads are interleaved on a cycle-by-cycle basis over the shared execution resources. Consecutive instructions executed by a given cluster may be from distinct H–Threads and instructions executed at the same time on different clusters may be from distinct V–Threads. This flexible interleaving allows the MAP to exploit thread-level paral-

(a) Single H–Thread

	MEM Unit	*FP Unit*
1.	`load` r_u	
2.	`load` r_d	
3.	`load` r_n	$t_2 = r_u + r_d$
4.	`load` r_s	$t_2 = t_2 + r_n$
5.	`load` r_e	$t_2 = t_2 + r_s$
6.	`load` r_w	$t_2 = t_2 + r_e$
7.	`load` r_*	$t_2 = t_2 + r_w$
8.	`load` u_*	$t_2 = b \times t_2$
9.		$t_1 = a \times r_*$
10.		$t_1 = t_1 + t_2$
11.		$u_* = u_* + t_1$
12.	`store` u_*	

(b) Two concurrent H–Threads

H–Thread 0

	MEM Unit	*FP Unit*
1.	`load` r_u	
2.	`load` r_d	
3.	`load` r_*	$t_2 = r_u + r_d$
4.	`load` u_*	$t_2 = b \times t_2$
5.		$t_1 = a \times r_*$
6.		$t_1 = u_* + t_1$
7.		$H1.t_2 = t_1 + t_2$

H–Thread 1

	MEM Unit	*FP Unit*
1.	`load` r_n	
2.	`load` r_s	`empty` t_2
3.	`load` r_e	$t_1 = r_n + r_s$
4.	`load` r_w	$t_1 = t_1 + r_e$
5.		$t_1 = t_1 + r_w$
6.		$t_1 = b \times t_1$
7.		$u_* = t_1 + t_2$
8.	`store` u_*	

Figure 4: Example of H–Threads used to exploit instruction level parallelism: (a) single H–Thread, (b) two H–Threads. The computation is a smoothing operator using a 7-point stencil on a 3–D grid: $u_* = u_* + a \times r_* + b \times (r_u + r_d + r_n + r_s + r_e + r_w)$.

lelism and to mask variable pipeline, memory, and communication delays.

The arrangement of V–Threads, H–Threads, instructions, and operations is summarized in Figure 3. The contexts of six V–Threads are resident in the clusters' register files. Each V–Thread includes four H–Threads, one on each cluster. Each H–Thread consists of a sequence of 3-wide instructions containing integer, memory, and floating point operations. On subsequent cycles, a cluster, as demonstrated by cluster 0, may issue instructions from different V–Threads.

3.1 V–Threads

A *V–Thread* consists of at least one and up to four H–Threads, each running concurrently on a different cluster. The MAP has sufficient hardware resources to hold the state of six V–Threads (24 H–Threads), each one occupying a *thread slot*. Four of these slots are *user* slots, one is the *event* slot, and one is the *exception* slot. User threads run in the user slots, handlers for asynchronous events run in the event slot, and handlers for synchronous exceptions detected and localized within a cluster, such as protection violations, run in the exception slot. Message arrival is treated as an asynchronous event.

The H–Threads within the same V–Thread may communicate and synchronize via registers, while H–Threads of different V–Threads must synchronize and communicate through memory or messages.

On each cluster, six H–Threads (one from each V–Thread) are interleaved dynamically over the cluster resources on a cycle-by-cycle basis. A synchronization pipeline stage holds the next instruction to be issued from each of the six V–Threads until all of its operands are present and all of the required resources are available, similar to the architecture described in [18]. At every cycle this stage decides which instruction to issue from those which are ready to run. An H–Thread that is stalled waiting for data or resource availability consumes no resources other than the thread slot that holds its state. Multiple V–Threads may be interleaved with zero delay, allowing task switching to mask even very short pipeline latencies as well as longer communication and synchronization latencies. As long as its data and resource dependencies are satisfied, a single thread may issue an instruction every cycle. Therefore, single thread performance is not penalized as a result of the M–Machine's support for multithreading.

3.2 H–Threads

An *H–Thread* runs on a single cluster and executes a sequence of operation triplets (zero or one operation for each of the 3 ALUs in the cluster) that are issued simultaneously. Within an H–Thread, instructions are guaranteed to issue in order, but may complete out of order. An H–Thread may communicate and synchronize via registers with the 3 other H–Threads within the same V–Thread. Each H–Thread may only read operands from its own register file, but can write directly into the register files of the H–Threads that are within the same V–Thread on other clusters.

The H–Thread mechanism can support multiple execution models. H–Threads can execute as independent threads with possibly different control flows to exploit loop-level or thread-level parallelism. Alternatively, the compiler can schedule the four H–Threads in a V–Thread as a unit to exploit instruction level parallelism, as in a VLIW machine. In this case the compiler must insert explicit register-based synchronization operations to enforce instruction ordering between H–Threads. Unlike the lock-step execution of traditional VLIW machines, H–Thread synchronization occurs only as required by data or resource dependencies. While explicit synchronization incurs some overhead, it allows H–Threads to slip relative to one other in order to accommodate variable–latency operations such as memory accesses.

Figure 4 shows a simple illustrative example of the instruction sequences of a program fragment on 1 and 2 H–Threads. The program is the body of the inner loop of a "smoothing" operation using a 7-point stencil on 3-D grid. On a particular grid point, the smoothed value is given by $u_* = u_* + a \times r_* + b \times (r_u + r_d + r_n + r_s + r_e + r_w)$, where r_* is the residual value at that point, and r_u, r_d, r_n, r_s, r_e and r_w are the residuals at the neighboring grid points in the six directions UP, DOWN, NORTH, SOUTH, EAST and WEST respectively. In order to better illustrate the use of H–Threads, advanced optimization (such as software pipelining) is not performed.

Figure 4*(a)* shows the single H–Thread program, with a 12 long instruction stream which includes all of the memory and floating point operations. The weighting constants `a` and `b` are kept in registers. Figure 4*(b)* shows the instruction streams for two H–Threads working cooperatively. Each H–Thread performs four memory operations and some of the arithmetic calculations. Instruction 7 in H–Thread 0 calculates a partial sum and transmits it directly to register `t2` in H–Thread 1. The `empty` instruction on H–Thread 1 is used to prepare `t2` for H–Thread synchronization; H–Thread 1 will not issue instruction 7 until the data arrives from H–Thread 0 as explained below.

The use of multiple H–Threads reduces the static depth of the instruction sequences from 12 to 8. On a larger 27-point stencil, the depth is reduced from 36 to 17 when run on 4 H–Threads. The actual execution time of the program fragments will depend on the pipeline and memory latencies.

H–Thread Synchronization

As shown in the example above, H–Threads within the same V–Thread synchronize with one another through registers. A scoreboard bit associated with the destination register is cleared (empty) when a multicycle operation, such as a load, issues and is automatically set (full) by MAP hardware when the result is available. An operation that uses the result will not be selected for issue until the corresponding scoreboard bit is set.

All inter-cluster data transfers require explicit register synchronization. To prepare for inter-cluster data transfers, the receiving H–Thread executes an EMPTY operation to mark empty a set of destination registers. As each datum arrives from the transmitting H–Thread over the C-Switch, the corresponding destination register is automatically set full by MAP hardware. An instruction in the receiving H–Thread that uses the arriving data will not be eligible for issue until its data is available. Therefore, explicit synchronization operations required by VLIW style execution across H–Threads may be overlapped with the inter-cluster data transfers inherent in the executing program.

Each V–Thread has an independent set of global condition code (CC) registers. Each set is composed of four pairs of single-bit

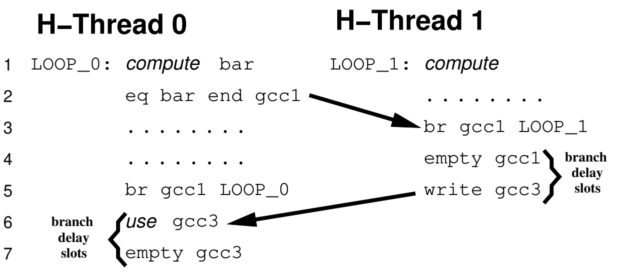

Figure 5: Loop synchronization between two H–Threads using MAP global condition code (CC) registers.

global CC registers and is used to broadcast binary values between H–Threads within a V–Thread. Similar to data registers, each global CC register has an accompanying scoreboard bit. The MAP global CC registers are physically replicated on each of the clusters and are not centrally located. An H–Thread may broadcast to other H–Threads of the same V–Thread using either of its writable global CC registers (one unique writable pair per H–Thread), but may read and mark empty its local copy of *any* global CC register in the set. Using these registers, all four H–Threads can execute conditional branches and assignment operations based on a comparison performed by a single H–Thread.

The scoreboard bits associated with the global CC registers may be used to rapidly synchronize among the H–Threads within a V–Thread. Figure 5 shows an example of two H–Threads synchronizing at loop boundaries. Two registers are involved in the synchronization, in order to provide an interlocking mechanism ensuring that neither H–Thread rolls over into the next loop iteration.

H–Thread 0 computes `bar`, compares it (using `eq`) to `end`, and broadcasts the result by targetting `gcc1`. H–Thread 1 uses `gcc1` to determine whether to branch, marks `gcc1` empty again, and writes to `gcc3` to notify H–Thread 0 that the current value of `gcc1` has been consumed. H–Thread 0 blocks until `gcc3` is full, and then empties it for the next iteration. Neither thread can proceed with the next iteration until both have completed the current one. Due to the multicopy structure of MAP global CC registers, this protocol can easily be extended to perform a fast barrier among 4 H–Threads executing on different clusters, without combining or distribution trees.

3.3 Asynchronous Exception Handling

Exceptions that occur outside the MAP cluster are termed *events* and are handled asynchronously by generating an *event record* and placing it in a hardware event queue. Local TLB misses, block status faults, memory synchronizing faults, and message arrivals are events that are handled asynchronously. These events are precise in the sense that the faulting operation and its operands are specifically identified in the event record, but they are handled asynchronously, without stopping the thread. Each H–Thread in the event V–Thread slot handles one class of events. Memory synchronization and status faults are handled on cluster 0, local TLB misses are handled on cluster 1, and arriving messages are handled on clusters 2 and 3, depending on the priority of the message.

The dedicated handler located in each H–Thread of the *event* V–Thread slot processes event records to complete the faulting operations. The event handler loops, reading event records from the register-mapped queue and processing them in turn. A read from the queue will not issue if the queue is empty. For example, on a local TLB miss, the hardware formats and enqueues an event record containing the faulting address as well as the write data or read destination. The sotware TLB miss handler reads the record, places the requested page table entry in the TLB, and restarts the memory reference. The thread that issued the reference does not block until it needs the data from the reference that caused the miss. Similarly, inter-node message arrival is also treated as an event in which the contents of the message are written into the appropriate event queue (which serves as the message queue).

Handling events asynchronously obviates the need to cancel all of the issued operations following the faulting operation, a significant penalty in a 12-wide machine with deep pipelines. Dedicating H–Threads to this purpose accelerates event handling by eliminating the need to save and restore state, and allows concurrent (interleaved) execution of user threads and event handlers. Asynchronous event handling does require sufficient queue space to handle the case where every outstanding instruction generates an exception. To reduce queue size requirements, those exceptions that can be detected in the first execution cycle, such as protection violations and some arithmetic exceptions, stall all user H–Threads in the affected cluster, and are handled synchronously by the local H–Thread of the *exception* V–Thread. User H–Threads executing on neighboring clusters are unaffected.

3.4 Discussion

There are two major methods of exploiting instruction level parallelism. Superscalar processors execute multiple instructions simultaneously by relying upon runtime scheduling mechanisms to determine data dependencies [31, 17]. However, they do not scale well with increasing number of function units because a greater number of register file ports and connections to the function units are required. In addition, superscalars attempt to schedule instructions at runtime (much of which could be done at compile time), but they can only examine a small subsequence of the instruction stream.

Alternatively, Very Long Instruction Word (VLIW) processors such as the Multiflow Trace series [6] use only compile time scheduling to manage instruction-level parallelism, resource usage, and communication among a partitioned register file. However, the strict lock-step execution is unable to tolerate the dynamic latencies found in multiprocessors.

Processor Coupling, originally introduced in [18], used implicit synchronization between the clusters on every wide instruction. Relaxing the lock-step synchronization, as described in this section, has several advantages. First, it is easier to implement because control is localized completely within the clusters. Second, it allows more slip to occur between the instruction streams running on different clusters (H–Threads), which eliminates the automatic blocking of one thread on long latency operations of another, providing more opportunity for latency tolerance. Finally, the H–Threads can be used flexibly to exploit both instruction and loop level parallelism. When H–Threads must synchronize, they do so explicitly through registers, at a higher cost than implicit synchronization. However, fewer synchronization operations are required, and many of them can be included in the data transfer between clusters, inherent in the executing program.

Using multiple threads to hide memory latencies and pipeline delays has been examined in several different studies and machines. Gupta and Weber explore the use of multiple hardware contexts in multiprocessors [10], but the context switch overheads they used are too large to mask pipeline latencies. MASA [13] as well as HEP [29] and TERA [3] use fine grain multithreading to issue an instruction from a different context on every cycle in order to mask pipeline latencies. However, with the required round-robin scheduling, single thread performance is degraded by the number of pipeline stages. The zero cost switching among V–Threads and the pipeline design of the MAP provide fast single thread execution as well as latency tolerance for better local memory bandwidth utilization. Furthermore, none of the multithreaded machines have multiple clusters for exploiting wide instruction level parallelism.

Various machines optimized for dataflow languages [24, 16, 28] provide hardware support for fine grained synchronization between threads (usually via memory synchronization bits), but they do not exploit instruction level parallelism, nor do they provide low cost register-based synchronization between threads. The XIMD architecture [33] uses multiple ALUs to exploit instruction level parallelism as well as thread level parallelism. However, it uses a single global register file and does not interleave multiple threads over the same execution units. Two approaches that do exploit instruction level parallelism using multiple threads and multiple ALUs include [30] and [32].

4 Inter–node Concurrency Mechanisms

The M–Machine provides a fast, protected, user-level message passing substrate. A user program may communicate and synchronize by directly sending messages or by reading and writing remote memory using a coherent shared memory system layered on the message-passing substrate. Direct messaging provides maximum performance data transfer and synchronization while shared memory access simplifies programming. Remote memory access is implemented using fast trap handlers that intercept load and store operations which reference remote data. These handlers send messages to other nodes to complete remote memory references transparently to user programs. Additional hardware and software mechanisms allow remote data to be cached locally in both the cache and external memory.

4.1 Message Passing Support

The M–Machine provides hardware support for injecting a message into the network, determining the message destination, and dispatching a handler on message arrival. For example, Figure 6 shows the M–Machine instruction sequences for both the sending and receiving components of a remote memory store. The message sending sequence (Figure 6*(a)*) loads the data to be stored into general register `MC1`. The `SEND` instruction takes three arguments, the target address (Taddr) contained in `Raddr`, the dispatch *(a) Message Send*

```
LOAD    A[0], MC1           ; load A[0] into register 1
SEND    Raddr, Rdip, #1     ; send a 1 word remote store
                            ; message to the processor
                            ; containing VA in Raddr
```

(b) Message Receive

```
loop:
JMP       Rnet          ; jump to DIP (remote write)
; start of remote write code
MOVE      Rnet, R1      ; move virtual address into R1
STORE     Rnet, R1      ; store word to memory
BRANCH    loop          ; branch to message dispatch code
```

Figure 6: Example of M–Machine code implementing a remote store: *(a)* Sending a 3 word remote store message. *(b)* Receiving and performing the store. On the receiving end `Rnet` is the register mapped to the head of the message queue.

				Extent		
Virtual Page	Starting Node	Page–group Length	Pages/ Node	Z	Y	X
42 bits	16 bits	6 bits	6 bits	3 bits each		

Figure 7: Format of a Global Destination Table (and GTLB) entry, used to determine a physical node identifier from a virtual address.

instruction pointer (DIP) in `Rdip`, and the message body length (`#1`). When the `SEND` issues, the Global Translation Lookaside Buffer (GTLB) translates virtual address `Raddr` into a physical node identifier and a 3 word message containing DIP, Taddr, and the contents of `MC1` is sent to that node. When the message arrives at the destination (Figure 6*(b)*) hardware enqueues it in the priority 0 message queue. An H–Thread dedicated to message handling jumps to the handler via the DIP contained in the first word of the message, executes a store operation, and branches back to the dispatch portion of the code.

Message Injection: A message is composed in a cluster's general registers and transmitted atomically with a single `SEND` instruction that takes as arguments a *destination* virtual address, a *dispatch instruction pointer (DIP)*, and the message body length. Hardware composes the message by prepending the destination and DIP to the message body and injects in into the network. Two message priorities are provided: user messages are sent at priority zero, while priority 1 is reserved for system level message reply, thus avoiding deadlock.

Message Address Translation: As described in [25], the explicit management of processor identifiers by application programs is cumbersome and slow. To eliminate this overhead, the MAP implements a Global Translation Lookaside Buffer (GTLB), backed by a software Global Destination Table (GDT), to hold mappings of virtual address regions to node numbers. These mappings may be changed by system software. The user specifies the destination of a message with a virtual address, which the network output interface hardware uses to access the GTLB and calculate the physical destination node.

With a single GTLB entry, a range of virtual addresses (called a page-group) is mapped across a region of processors. In order to simplify encoding, the page-group must be a power of 2 pages in size, where each page is 1024 words. The mapped processors must be in a contiguous 3–D rectangular region with a power of 2 number of nodes on a side. This information is encoded in a single GTLB entry as shown in Figure 7. The *virtual page* field is used as the tag during the fully associative GTLB lookup. The *starting node* enumerates the coordinates of the origin of the region of mapped processors, while the *extent* specifies the base 2 logarithm of the X, Y, and Z dimensions of the region. The *page-group length* field specifies the number of local pages that are mapped into the page group. The *pages-per-node* field indicates the number of pages placed on each consecutive processor, and is used to implement a spectrum of block and cyclic interleavings.

Message Reception: At the destination node, an arriving message is automatically placed in a hardware message queue. The head of the message queue is mapped to a register accessible by an H–Thread (in either cluster 2 or 3, depending on message priority) in the *event* V–Thread. The message dispatch handler code running in that H–Thread stalls on the empty register until a message arrives, marking the register full; the handler then reads the dispatch instruction pointer (DIP) from the register and jumps to it. This starts execution of the specific handler code to perform the action requested in the message. Some of the actions include remote read, remote write, and remote procedure call. The message need not be copied to or from memory, as it is accessible via a general register. In order to avoid overflow of the fixed size message queue and back up of the network, only short, well–bounded tasks are executed by message handlers. Longer tasks are enqueued to be run as a user process on a user V–Thread.

Protection: The M–Machine communication substrate provides fully protected user-level access to the network. The `SEND` instruction atomically launches a message into the network, preventing a user from occupying the network output indefinitely. The automatic translation provided by the GLTB ensures that a program may only send messages to virtual addresses within its own address space. Finally, restricting the set of user accessible DIPs prevents a user handler from monopolizing the network input. If an illegal DIP is used, a fault will occur on the sending thread before the message is sent.

Throttling: In order to prevent a processor from injecting messages at a rate higher than they can be consumed, the M–Machine implements a *return-to-sender* throttling protocol. A portion of a local node's memory is used for returned message buffering. When a message is sent, a counter is automatically decremented, which reserves buffer space for that message, should it be returned. If the counter is zero, no buffer space is available and no additional messages may be sent; threads attempting to execute a `SEND` instruction will stall. When the message reaches the destination, a reply is sent indicating whether the destination was able to handle the message. If the message was consumed, the reply instructs the source processor to increment its counter, deallocating the buffer space. Otherwise, the reply contains the contents of the original message which are copied into the buffer and sent again later.

Discussion: The M–Machine provides direct register-to-register communication, avoiding the overhead of memory copying at both the sender and the receiver, and eliminating the dedicated memory for message arrival, as is found on the J–Machine [8]. Register-mapped network interfaces have been used previously in the Mars Machine [2], J-Machine, and iWarp [4], and have been described by *T [26] as well as Henry and Joerg [15]. However, none of these systems provide protection for user-level messages.

Systems, like the J-Machine, that provide user access to the network interface without atomicity must temporarily disable interrupts to allow the sending process to complete the message. The M–Machine's atomic `SEND` instruction eliminates this requirement at the cost of limiting message length to the number of cluster registers. Most messages fit easily in this size and larger messages can be packetized and reassembled with very low overhead.

Automatic translation of virtual processor numbers to physical processor identifiers is used in the Cray T3D [7]. The use of virtual addresses as message destinations in the M–Machine has two advantages. When combined with translation hardware, it provides protection for user initiated messages, without incurring the overhead of operating system invocation, as messages may not be sent to processors mapped outside of the user's virtual address space. It also facilitates the implementation of global shared memory. The interleaving performed by the GTLB, although not as versatile as the CRAY T3D address centrifuge or the interleaving of the RP3 [27], provides a means of distributing ranges of the address space across a region of nodes.

In contrast to both *T and FLASH [19] which use a separate communication coprocessor for receiving incoming messages, the M–Machine incorporates that function on its already existing execution resources, an H–Thread in the event V–Thread. This avoids idling a dedicated processor when it is not in use. During periods of few messages, user threads may make full use of the cluster's arithmetic and memory bandwidth.

4.2 Non-Cached Shared Memory

Fast access to remote memory is provided transparently to the user with a combination of hardware and software mechanisms. When a load or store operation to a global virtual address causes a Local Translation Lookaside Buffer (LTLB) miss, a software trap is signalled. Like the hardware dedicated to message arrival, one H–Thread in the *event* V–Thread is reserved for handling LTLB misses. The LTLB miss handler code probes the GTLB to determine where the requested data is located, and if necessary, sends a message to the destination node. If the data is in fact local, the LTLB miss handler fetches the required page table entry and places it in the LTLB. Using a small portion of the execution resources for fast trap handling reduces the latency of both local LTLB misses and remote data access.

The sequence of operations required to satisfy a remote memory load is shown below. The labels *HW* and *SW* indicate whether the action is performed by hardware or software.

1. *HW*: Memory operation accesses the cache and misses (2 cycles).
2. *HW*: LTLB miss occurs, enqueueing an event (2 cycles).

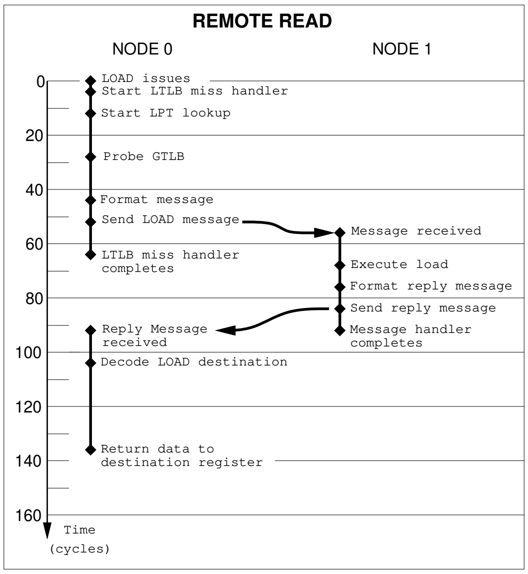

Figure 8: Remote read access.

3. *SW*: Software accesses the local page table (LPT), probes the GTLB, and composes and sends a message containing the referenced and return addresses (48 cycles).
4. *HW*: Message delivered to remote node (5 cycles).
5. *SW*: Message handler fetches requested data from memory, formats a reply message, and sends it (29 cycles).
6. *HW*: Return message delivered (5 cycles).
7. *SW*: Message handler decodes the original load destination register and writes the data directly there (41 cycles).

Timelines for both remote read and write accesses are shown in Figures 8 and 9. These measurements are based on prototype message and event handlers written in assembly code and running on the M–Machine simulator. A user level program running on node 0 makes read and write requests to memory on neighboring node 1. Except for the message handler that runs on demand, node 1 is idle. All references to memory system data structures in the software handlers are assumed to cache hit.

Table 1 shows a comparison of preliminary results of local and remote access latencies (in cycles), for single word accesses. A read is completed when the requested data has been written into the destination register. A write is completed when the line containing the data has been fully loaded into the cache. The remote read and write accesses are larger than their local counterparts due to the software intervention required to send the message to the remote node. However, the time to perform a remote read that hits in the cache is only a twice as large as a local read that requires software intervention (LTLB miss). For the remote write, which does not require return data, the difference is only 10%.

The primary contributors to remote access latency in the M–Machine are searching for the faulting address in the local page table and decoding the reply message (about 40 cycles each). The page-table search is required only when accessing the first

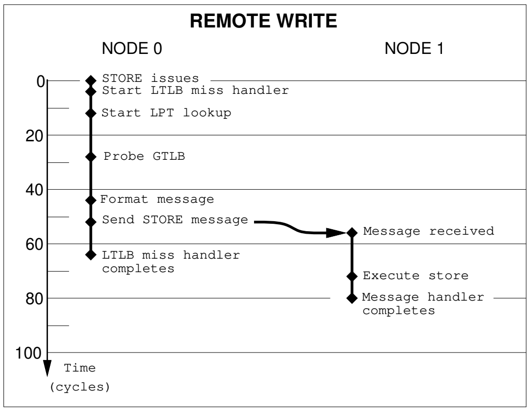

Figure 9: Remote write access.

	Access Times (cycles)	
Access Type	READ	WRITE
Local Cache Hit	3	2
Local Cache Miss	13	19
Local LTLB Miss	61	67
Remote Cache Hit	138	74
Remote Cache Miss	154	90
Remote LTLB Miss	202	138

Table 1: Comparison of local and remote access times, assuming no resource contention.

block of a page. Access to subsequent blocks cause block-status faults (rather than page faults) which skip the page-table access. The reply decode could be accelerated by prohibiting the faulting V–Thread from swapping out during the memory operation.

4.3 Caching and Coherence

Even though remote accesses are fast, their latency is still large compared to local memory references. This overhead reduces the ability of the MAP to use the network and remote memory bandwidth effectively. To reduce overall latency and improve bandwidth utilization, each M–Machine node may use its local memory to cache data from remote nodes.

In addition to the virtual to physical mapping, each LTLB (and LPT) entry contains 2 status bits for each cache block in the page. These *block status* bits are used to provide fine-grain control over 8 word blocks, allowing different blocks within the same mapped page to be in different states. This fine-grain control over data is similar to that provided in hardware based cache coherent multiprocessors, and alleviates the false sharing that exists in other software data coherence systems [21]. The two block status bits are used to encode the following four states:

- `INVALID`: The block may not be read, written, or placed in the hardware cache.
- `READ-ONLY`: The block may be read, but not written.
- `READ/WRITE`: The block may be read or written.
- `DIRTY`: The block may be read or written, and it has been written since being copied to the local node.

One software policy that uses the block status bits fetches remote cache blocks on demand. When a memory reference occurs, the block status bits corresponding to the global virtual address are checked in hardware. If the attempted operation is not allowed by the state of the block, a software trap called a *block status* fault occurs. The trap code runs in the event V–Thread, in the H–Thread that is reserved for handling block status and synchronization events. The block status handler sends a message to the home node, which can be determined using the GTLB, requesting the cache block containing the data. The home node logs the requesting node in a software managed directory and sends the block back. When the block is received, the data is written to memory and the block status bits are marked valid. If the virtual page containing the block is not mapped to a local physical page, a new page table entry is created and only the newly arrived block is marked valid. The remote data may be loaded into the on-chip cache, and modifications to the data will automatically mark the block state dirty. More complex coherence schemes can map blocks from different virtual pages into the same physical page, reducing the amount of unmapped physical memory.

The software handlers used to transmit data from node to node may implement a variety of coherence policies and protocols. This code is easily incorporated within the remote read and write handlers described in Section 4.2. Using local memory as a repository will allow more remote data to be cached locally than could fit in the on-chip cache alone.

Discussion: Directory-based, cache coherent multiprocessors such as Alewife [1] and DASH [20] implement coherence policies in hardware. This improves performance at the cost of flexibility. Like the M–Machine, FLASH [19] implements remote memory access and cache coherence in software, but uses a coprocessor. However, this system does not provide block status bits in the TLB to support caching remote data in DRAM. The subpage status bits of the KSR-1 [9] perform a function similar to that of the block status bits of the M–Machine.

Implementing remote memory access and coherence completely in software on a conventional processor would involve delays much greater than those shown in Table 1, as evidenced by experience with the Ivy system [21]. The M–Machine's fast exception handling in a dedicated H–Thread avoids the delay associated with context switching and allows the user thread to execute in parallel with the exception handler. The GTLB avoids the overhead of manual translation and the cost of a system call to access the network. Finally, the M–Machine provides memory-mapped addressing of thread registers to allow the operation to be completed in software.

5 M–Machine Software

The M–Machine addresses the problem of parallel software by supporting an incremental approach to parallelization. Unlike conventional parallel machines, the M–Machine is designed to efficiently run a sequential program that uses all the machine's memory, including that on remote nodes. A shared address space, high-performance messaging, and caching remote data in local DRAM provide fast access to remote data. The programmer can then incrementally improve program performance by adding parallelism. The cache coherence mechanisms enable efficient sharing of data across processors. The high-speed messaging network and runtime system support allow for low-overhead task parallelism. The ability to support fine-grain parallelism increases the number of suitable tasks and allows extraction of more parallelism from small problems. Support for synchronizing memory operations and global addressing simplifies user-level communication and synchronization between tasks and reduces overhead. Caching in DRAM automates much of the data placement and migration. For the cases where a programmer wants to extract the maximum performance, fast, protected, user-level messages may be employed.

The M–Machine software is being designed and implemented jointly with the Scalable Concurrent Programming Laboratory at Caltech. The Multiflow compiler [22] is being ported to the M–Machine to generate long instructions spanning multiple clusters. The Multiflow compiler is designed to generate VLIW instructions from a sequential source program using Trace Scheduling. The modifications required to generate multicluster code for the M–Machine consist of partitioning the graph (DAG) of the trace into sub-DAGs that may be executed on different clusters with minimal communication. The sub-DAGs are then scheduled for each cluster using the greedy instruction scheduler of the Multiflow compiler. Explicit synchronization is required to take the place of the implicit synchronization of a true VLIW. Communication is implemented by writing to remote registers, while the global condition registers are used to implement explicit barrier synchronization. An algorithm that might be used to discover the synchronization points is described in [34]. The compiler currently generates code for a single cluster, but adding the partitioning and synchronization as well as integrating the standard optimizations is underway.

A prototype runtime system consisting of primitive message and event handlers has also been implemented. Approximately 90 percent of the runtime system code is implemented in C, compiled using the Multiflow compiler, and runs on the M–Machine simulator; the remaining 10 percent are assembly code routines which access hardware features not exposed to the compiler. The runtime system consists of independent modules which manage virtual memory allocation, physical memory allocation, memory coherence between nodes, and multiple threads on a single node and across nodes. The implementation of the runtime system is described more fully in [11].

6 Conclusion

In this paper we have described the architecture of the M–Machine with an emphasis on its novel features. The M–Machine, currently under development, is a 3–D mesh, each node of which contains a multi-ALU processor (MAP) and 8 MBytes of synchronous DRAM. Each MAP chip consists of four 64-bit 3-issue clusters connected by a cluster switch, a 4-way interleaved on-chip cache, an external memory interface, and on-chip network interfaces and routers.

Instruction level parallelism is exploited both within a cluster and across clusters using H–Threads. An H–Thread may communicate and synchronize through registers with H–Threads on different clusters but within the same V–Thread. A 27 point stencil computation on 4 H–Threads (12-wide issue) has half the static instruction count of 1 H–Thread (3-wide issue).

To increase use of the local memory and execution bandwidth, multiple tasks, called V–Threads, are interleaved on a cycle-by-cycle basis independently on each of the clusters. Each cycle, a different thread may be selected for execution, or if only one V–Thread is resident, it may issue an instruction every cycle on each cluster.

The M–Machine has a user–level, protected, fast message passing substrate to reduce communication and remote memory latencies. Messages are composed in general registers and sent via a user level `SEND` instruction. Arriving messages are extracted by a system-level software message dispatch handler, which is always resident in the *event* V–Thread. The message contents are accessed via a register mapped queue. The message need not be copied to or from memory on either the sending or receiving side. Two level translation is used to independently relocate objects in the physical address space on a node, and in the processor namespace.

The fast message system is used to provide the user with transparent access to remote memory. When a user's load or store instruction traps to software on a LTLB miss, a message is sent to a remote node to perform the access. While slower than local accesses, a remote load can be satisfied in 138 cycles, while a remote store can be satisfied in 74 cycles. In order to facilitate local caching of remote data, 2 status bits for each block (8 words) in a page are added to the LTLB and page table entries. When an invalid block is accessed, a trap to software occurs which can retrieve the missing block from a remote node, copy it into local memory, and mark the status bits valid.

A cycle-accurate simulator of the M–Machine has been completed and is being used for software development. The hardware design of the MAP is currently underway; 80% of the modules have been designed and 60% of the execution unit schematics as well as layout of several modules is complete. The MAP will be implemented on a single integrated circuit with a projected area of $17mm \times 18mm$ in $0.5\mu m$ CMOS with 5 metal layers. The target clockrate is 100MHz and tapeout is expected in 1996.

The M–Machine addresses the issues of non-uniform technology scaling and of programmability. By changing the ratio of processor to memory area, the M–Machine better balances cost and improves the utilization of the increasingly critical memory bandwidth. The M–Machine increases the ratio of processor to memory silicon area to 11% from 0.13% for a typical 1996 system. A 32–node (128 clusters) M–Machine with a total of 256 MBytes of memory requires 50% more area than a uniprocessor with the same amount of memory but provides 128 times as much peak performance, a 85:1 improvement in peak-performance/area. This increase in processing resources allows the M–Machine to saturate the costly DRAM bandwidth even for problems with good locality and thus is expected to run programs faster, allowing a fixed-size memory system to run more programs per unit time. The 85:1 improvement in peak-performance/area makes the increased parallelism of the M–Machine cost effective even in cases where only a small fraction of its peak performance is realized.

We expect that the architecture concepts demonstrated in the M–Machine will be useful in machines ranging from single-node personal computers, through workstations with tens of nodes, to servers with hundreds to thousands of nodes. Memory bandwidth and capacity are becoming the dominant factor in the cost and performance of systems of all scales. By changing the processor/memory ratio, providing methods for extracting parallelism at all levels, and supporting an incremental approach to parallelism, the M–Machine's mechanisms will lead to more cost effective and programmable machines across the price-performance spectrum.

Acknowledgements

We would like to thank the members of the Scalable Concurrent Programming Laboratory at Caltech for their feedback on the M-Machine architecture and for their work on the M-Machine software effort: Steve Taylor, Daniel Maskit, and Bryan Chow. In additional we would also like to thank Kathy Knobe and the anonymous referees for their comments on various versions of this paper.

References

[1] AGARWAL, A., ET AL. The MIT Alewife machine: A large-scale distributed-memory multiprocessor. In *Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1991.

[2] AGRAWAL, P., DALLY, W., FISCHER, W., JAGADISH, H., KRISHNAKUMAR, A., AND TUTUNDJIAN, R. MARS: A multiprocessor-based programmable accelerator. *IEEE Design Test 4* (October 1987), 28–36.

[3] ALVERSON, R., ET AL. The Tera computer system. In *Proceedings of the 1990 International Conference on Supercomputing* (Sept. 1990), ACM SIGPLAN Computer Architecture News, pp. 1–6.

[4] BORKAR, S., ET AL. Supporting systolic and memory communication in iWarp. In *Proceedings of the 17th International Symposium on Computer Architecture* (May 1990), pp. 70–81.

[5] CARTER, N. P., KECKLER, S. W., AND DALLY, W. J. Hardware support for fast capability-based addressing. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)* (Oct. 1994), Association for Computing Machinery Press, pp. 319–327.

[6] COLWELL, R. P., HALL, W. E., JOSHI, C. S., PAPWORTH, D. B., RODMAN, P. K., AND TORNES, J. E. Architecture and implementation of a VLIW supercomputer. In *Proceedings of Supercomputing '90* (November 1990), IEEE Computer Society Press, pp. 910–919.

[7] CRAY RESEARCH, INC. *Cray T3D System Architecture Overview*. Chippewa Falls, WI, 1993.

[8] DALLY, W. J., ET AL. The J-Machine: A fine-grain concurrent computer. In *Proceedings of the IFIP Congress* (Aug. 1989), G. Ritter, Ed., North-Holland, pp. 1147–1153.

[9] FRANK, S. J., ET AL. Multiprocessor digital data processing system. United States Patent No. 5,055,999, October 8 1991.

[10] GUPTA, A., AND WEBER, W.-D. Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: Preliminary results. In *Proceedings of 16th Annual Symposium on Computer Architecture* (May 1989), IEEE, pp. 273–280.

[11] GUREVICH, Y. The M-Machine operating system. Master of Engineering Thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, September 1995.

[12] GWENNAP, L. New MIPS chip targets windows NT boxes. *Microprocessor Report* (November 18, 1992).

[13] HALSTEAD, R. H., AND FUJITA, T. MASA: a multithreaded processor architecture for parallel symbolic computing. In *15th Annual Symposium on Computer Architecture* (May 1988), IEEE Computer Society, pp. 443–451.

[14] HENNESSY, J. L., AND JOUPPI, N. P. Computer technology and architecture: An evolving interaction. *Computer* (Sept. 1991), 18–29.

[15] HENRY, D. S., AND JOERG, C. F. A tightly-coupled processor-network interface. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)* (Oct. 1992), ACM, pp. 111–122.

[16] HUM, H. H., ET AL. A design study of the EARTH multiprocessor. In *International Conference on Parallel Architectures and Compilation Techniques* (1995), pp. 59–68.

[17] JOHNSON, W. M. *Superscalar Microprocessor Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[18] KECKLER, S. W., AND DALLY, W. J. Processor coupling: Integrating compile time and runtime scheduling for parallelism. In *Proceedings of the 19th International Symposium on Computer Architecture* (Queensland, Australia, May 1992), ACM, pp. 202–213.

[19] KUSKIN, J., OFELT, D., HEINRICH, M., HEINLEIN, J., SIMONI, R., ET AL. The Stanford FLASH multiprocessor. In *Proc. 21st International Symposium on Computer Architecture* (Apr. 1994), IEEE, pp. 302–313.

[20] LENOSKI, D., LAUDON, J., JOE, T., NAKAHIRA, D., STEVENS, L., GUPTA, A., AND HENNESSY, J. The DASH prototype: Implementation and performance. In *Proceedings of 19th Annual International Symposium on Computer Architecture* (1992), IEEE, pp. 92–103.

[21] LI, K. Ivy: A shared virtual memory system for parallel computing. In *International Conference on Parallel Processing* (1988), pp. 94–101.

[22] LOWNEY, P. G., FREUDENBERGER, S. G., KARZES, T. J., LICHTENSTEIN, W. D., NIX, R. P., O'DONNELL, J. S., AND RUTTENBERG, J. C. The multiflow trace scheduling compiler. *The Journal of Supercomputing 7*, 1-2 (May 1993), 51–142.

[23] MEAD, C. A., AND CONWAY, L. A. *Introduction to VLSI Systems*. Addison-Wesley, Reading, Mass, 1980.

[24] NIKHIL, R. S., PAPADOPOULOS, G. M., AND ARVIND. *T: A multithreaded massively parallel architecture. Computation Structures Group Memo 325-1, Laboratory for Computer Science, Massachusetts Institute of Technology, Nov. 1991.

[25] NOAKES, M. D., WALLACH, D. A., AND DALLY, W. J. The J-Machine multicomputer: An architectural evaluation. In *Proceedings of the 20th International Symposium on Computer Architecture* (San Diego, California, May 1993), IEEE, pp. 224–235.

[26] PAPADOPOULOS, G. M., BOUGHTON, G. A., GRAINER, R., AND BECKERLE, M. J. *T: Integrated building blocks for parallel computing. In *Proc. Supercomputing 1993* (1993), IEEE, pp. 624–635.

[27] PFISTER, G., ET AL. The IBM research parallel processor prototype (RP3): Introduction and architecture. In *Proc. International Conference on Parallel Processing* (1985), pp. 764–771.

[28] SAKAI, S., KODAMA, Y., AND YAMAGUCHI, Y. Prototype implementation of a highly parallel dataflow machine em-4. In *Proceedings of the Fifth International Parallel Processing Symposium* (May 1991), ieee Computer Society, ieee, pp. 278–286.

[29] SMITH, B. J. Architecture and applications of the HEP multiprocessor computer system. In *SPIE Vol. 298 Real-Time Signal Processing IV* (1981), Denelcor, Inc., Aurora, CO, pp. 241–248.

[30] SOHI, G. S., BREACH, S. E., AND VIJAYKUMAR, T. Multiscalar processors. In *Proceedings of the 22nd International Symposium On Computer Architecture* (May 1995), pp. 414–425.

[31] TOMASULO, R. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal 11* (January 1967), 25–33.

[32] TULLSEN, D. M., EGGERS, S. J., AND LEVY, H. M. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd International Symposium On Computer Architecture* (May 1995), pp. 392–403.

[33] WOLFE, A., AND SHEN, J. P. A variable instruction stream extension to the VLIW architecture. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (April 1991), ACM Press, pp. 2–14.

[34] ZAAFRANI, A., DIETZ, H. G., AND O'KEEFE, MATTHEW, T. Static scheduling for barrier MIMD architectures. In *1990 International Conference on Parallel Processing* (1990).