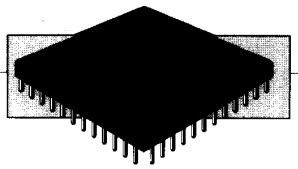


Sparcle: An Evolutionary Processor Design for Large-Scale Multiprocessors

Working jointly at MIT, LSI Logic, and Sun Microsystems, designers created the Sparcle processing chip by evolving an existing RISC architecture toward a processor suited for large-scale multiprocessors. This chip supports three multiprocessor mechanisms: fast context switching, fast, user-level message handling, and fine-grain synchronization. The Sparcle effort demonstrates that RISC architectures coupled with a communications and memory management unit do not require major architectural changes to support multiprocessing efficiently.

Anant Agarwal

John Kubiatowicz

David Kranz

Beng-Hong Lim

Donald Yeung

Massachusetts Institute of Technology

Godfrey D'Souza

LSI Logic

Mike Parkin

Sun Microsystems

The Sparcle chip clocks at no more than 40 MHz, has no more than 200,000 transistors, does not use the latest technologies, and dissipates a paltry 2 watts. It has no on-chip cache, no fancy pads, and only 207 pins. It does not even support multiple-instruction issue. Then why do we think this chip is interesting?

Sparcle is a processor chip designed to support large-scale multiprocessing. We designed its mechanisms and interfaces to provide fast message handling, latency tolerance, and fine-grain synchronization. Specifically, Sparcle implements

- *Mechanisms to tolerate memory and communication latencies, as well as synchronization latencies.* Long latencies are inevitable in large-scale multiprocessors, but current microprocessor designs are ill-suited to handle such latencies.
- *Mechanisms to support fine-grain synchronization.* Modern microprocessors pay scant attention to this aspect of multiprocessing, usually providing just a test-and-set instruction, and in some cases, not even that.
- *Mechanisms to initiate communication actions to remote processors across the communications network, and to respond rapidly to asynchronous events such as synchronization faults and message arrivals.* Current microprocessor designs do not support a clean communications interface between the processor and the communications network. Furthermore, traps and other asynchronous event-handlers are inefficient on many current microprocessors, often requiring tens of cycles to reach the appropriate trap service routine.

The impetus for the Sparcle chip project was our belief that we could implement a processor that provides interfaces for the above mechanisms by making small modifications to an existing microprocessor. Indeed, we derived Sparcle from Sparc[1] (scalable programmable architecture from Sun Microsystems), and we integrated it into Alewife,[2,3] a large-scale multiprocessor system being developed at MIT.

Sparcle tolerates long communication and synchronization latencies by rapidly switching to other threads of computation. The current implementation of Sparcle can switch to another thread of computation in 14 cycles. Slightly more aggressive modifications could reduce this number to four cycles. Sparcle switches to another thread when a cache miss that requires service over the communications network occurs, or when a synchronization fault occurs. Such a processor requires a pipelined memory and communications

0272-1732/93/0600-0048$03.00 © 1993 IEEE

system. In our system, a separate communications and memory management chip (CMMU) interfaces to Sparcle to provide the desired pipelined system interface. Our system also provides a software prefetch instruction. For a description of the modifications to a modern RISC microprocessor needed to achieve fast context switching, see our discussion under architecture and implementation of Sparcle later in the article.

Sparcle supports fine-grain data-level synchronization through the use of full/empty bits, as in the HEP computer.[1] With full/empty bits, a lock and access of the data word protected by the lock can be probed in one operation. If the synchronization attempt fails, the synchronization trap invokes a fault handler. In our system, the external communications chip detects synchronization faults and alerts Sparcle by raising a trap line. The system then handles the fault in software trap code.

Finally, Sparcle supports a highly streamlined network interface with the ability to launch and receive interconnection network messages. While this design implements the communications interface with the interconnection network in a separate chip, the CMMU, future implementations can integrate this functionality into the processor chip. Sparcle supports rapid response to asynchronous events by streamlining Sparc's trap interface and by supporting rapid dispatch to the appropriate trap handler. To achieve this, Sparcle provides two special trap lines for the most common types of events—cache misses to remote nodes and synchronization faults. Sparcle uses a third trap line for all other types of events. Also, this chip has an increased number of instructions in each trap dispatch entry so that vital trap codes can be put in line at the dispatch points.

Sparcle's design process was unusual in that it did not involve developing a completely new architecture. Rather, we implemented Sparcle with the help of LSI Logic and Sun Microsystems by slightly modifying the existing Sparc architecture. At MIT, we received working Sparcle chips from LSI Logic on March 11, 1992. These chips have already undergone complete functional testing. We are currently continuing to implement the Alewife multiprocessor so that we can thoroughly evaluate our ideas and subject the Sparcle chips to full-speed testing. Figure 1 shows an Alewife node with the Sparcle chip.

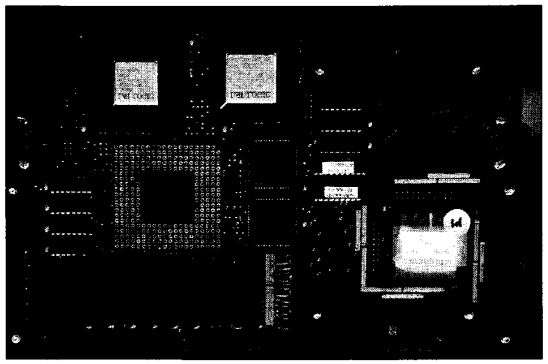


Figure 1. An Alewife node.

Mechanisms for multiprocessors

By supporting the widely used shared-memory and message-passing programming models, Sparcle eases the programmer's job and enhances parallel program performance. We have implemented programming constructs in parallel versions of Lisp and C that use these features. Sparcle's features fall into three areas, the first two of which support the shared-memory model:

- **Fine-grain computation.** Efficient support of fine-grain expression of parallelism and synchronization can enhance performance by increasing parallelism and reducing communication overhead. This enhancement relieves the programmer of undue effort in partitioning data and controlling flow into coarser chunks to increase performance.
- **Memory latency tolerance.** Context switching and data prefetching can reduce communication overhead introduced by network delays. For shared-memory programs, the switch must be very fast and occur automatically when a remote cache miss occurs.
- **Efficient message interface**. The ability to send and receive messages is needed to support message-passing programs. Such interfacing can also improve the performance of shared-memory programs in some common situations.

Before we can examine the implementation of these features in Sparcle, we need to consider each of these areas in turn, and discuss why they are useful for large-scale multiprocessing.

Fine-grain computation. As multiprocessors become larger, the grain size of parallel computations decreases to satisfy higher parallelism requirements. Computational grain size refers to the amount of computation between synchronization operations. Given a fixed problem size, the overhead of parallel and synchronization operations limits the ability to use a larger number of processors to speed up a program. Systems supporting fine-grain parallelism and synchronization attempt to minimize this overhead so that parallel programs can achieve better performance.

The challenge of supporting fine-grain computation is in implementing efficient parallelism and synchronization constructs without incurring extensive hardware cost, and without reducing coarse-grain performance. By taking an evolutionary approach in designing Sparcle, we have attempted to satisfy these requirements.

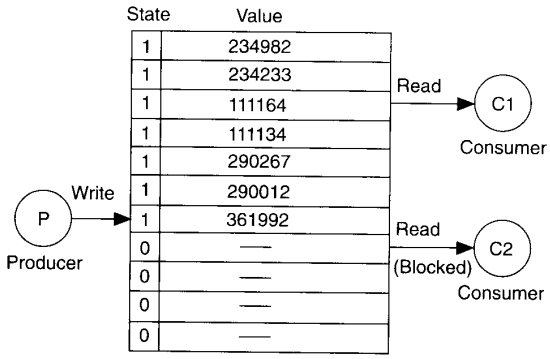


Figure 2. J-structures.

We can express fine-grain parallelism and synchronization at the data level (data-level parallelism) or at the thread level (control-level parallelism).

Data-level parallelism. Data-level parallelism and synchronization allows the program to synchronize at the level of the smallest possible unit—a memory word. At the programming language level, we provide parallel do-loops to express data-level parallelism, and J-structure and L-structure arrays to express fine-grain data-level synchronization.

Inspired by the I-structures of Arvind, Nikhil, and Pingali,[5] the J-structure is a data structure for producer-consumer style synchronization. It is like an array, but each element has an additional state—full or empty. The initial state of a J-structure element is empty. A reader of an element waits until the element's state is full before returning the value. A writer of an element writes a value, sets the state to full, and signals waiting readers to proceed. A write to a full element signals an error. For efficient memory allocation and cache performance, J-structure elements can be reset to an empty state. Figure 2 illustrates how J-structures can be used for data-level synchronization.

In the example of Figure 2, producer P is sequentially filling in the elements of a J-structure. Consumer C1 reads an element that is already filled and immediately gets its value. Consumer C2 reads an empty element and thus has to wait for P to write the element. Since we are synchronizing at the level of individual elements, both C1 and C2 can access the elements of the J-structure without waiting for P to completely fill all the elements of the J-structure.

L-structures are similar to J-structures but support three operations: a locking read, a nonlocking read, and a synchronizing write. A locking read waits until the element is full before emptying it (that is, locking it) and returning the value. A nonlocking read operation also waits until the element is full, but returns the value without emptying the element. A synchronizing write stores a value to an empty element and sets it to full, releasing any waiters. An L-structure thus allows mutually exclusive access to each of its elements and allows multiple nonlocking readers.

Sparcle supports J- and L-structures, as well as other types of fine-grain data-level synchronization, with per-word, full/empty bits in memory.[4] Sparcle provides new load/store instructions that interact with the full/empty bits. The design also includes an extra synchronous trap line to deliver the full/empty trap. This extra line allows Sparcle to immediately identify the trap.

Control-level parallelism. Control-level parallelism may be expressed by wrapping *future* around an expression or statement *X*. The *future* keyword declares that *X* and the continuation of the future expression may be evaluated concurrently. Fine-grain support allows the amount of computation needed for evaluating *X* to be small without severely affecting performance.

If the compiler or runtime system chooses to create a new task to evaluate *X*, it also creates an object known as a *placeholder* that is returned as the value of the future expression. The placeholder is created in an undetermined state. Evaluation of *X* yields its value and determines the placeholder. Any task that attempts to use the value of *X* before *X* has been completely evaluated will encounter the undetermined placeholder and will suspend operation until the placeholder is determined.

This functionality is implemented using (by software convention) the low bit of a data value as a placeholder tag; that is, a pointer to a placeholder has the low bit set and all other values have the low bit clear. New add, subtract, and compare instructions in Sparcle trap if the low bit of any operand is set. Likewise, dereferencing a pointer with the low bit set will cause an address alignment trap to a similar routine. If the trap handler can determine the value at the placeholders, it places this value in the target register, and normal execution resumes. Otherwise, the trapping task waits until the value of the placeholder becomes available.

With this support, a compiler can generate code without knowing which data values may be computed concurrently. Consequently, Sparcle incurs no runtime overhead to ensure the detection of placeholders.

Memory latency tolerance. Since memory in large-scale multiprocessors is distributed, cache misses to remote locations will incur long latencies and potentially reduce processor use. Figure 3 illustrates this problem by depicting processor and network activity when a single thread executes on the processor. When the thread suffers a long-latency cache miss, the processor waits for the miss to be satisfied before it can proceed. While waiting, both the processor and the network suffer idle time, thereby reducing their effective usage. Using latency tolerance mechanisms alleviates this problem and helps improve processor and network usage.

The general class of latency tolerance solutions all implement mechanisms that allow multiple outstanding memory transactions and can be viewed as a way of pipelining the processor and the network. The key difference between this pipeline into the network and the processor's execution pipeline is that the latency associated with the communication pipeline cannot be predicted easily at compile time. A compiler then has difficulty scheduling operations for maximal resource use. Systems must implement dynamic pipelines into the network in which the hardware ensures that multiple, previously issued memory operations have completed before issuing operations that depend on their completion. Context switching is one mechanism for dynamic pipelining. Other methods include prefetching and weak ordering.[6-8]

Sparcle implements fast context switching as its primary mechanism for dynamic latency tolerance. (Sparcle and its memory controller provide nonbinding prefetch instructions as well.) As illustrated in Figure 4, the basic idea is to overlap the latency of a memory request from a given thread of computation with the execution of a different thread. In the figure, when thread 1 suffers a cache miss, the processor switches to thread 2, thereby overlapping the cache miss latency of thread 1 with useful computation from thread 2.

In Alewife, when a thread issues a remote transaction or suffers an unsuccessful synchronization attempt, the Alewife CMMU traps the processor. If the trap resulted from a cache miss to a remote node, the trap handler forces a context switch to a different thread. Otherwise, if the trap resulted from a synchronization fault, the trap handling routine can switch to a different thread of computation. For synchronization faults, the trap handler might also choose to retry the request immediately (spin).

Processors that switch rapidly between multiple threads of computation are called multithreaded architectures. The prototypical multithreaded machine is the HEP. In the HEP, the processor switches every cycle between eight processor-resident threads. Cycle-by-cycle interleaving of threads is termed fine multithreading. Although fine multithreading offers the potential for high processor usage, it results in relatively poor single-thread performance and low processor use when there is not enough parallelism to fill all the hardware contexts.

In contrast, Sparcle employs block multithreading or coarse multithreading. That is, context switches occur only when a thread executes a memory request that must be serviced by a remote node in the multiprocessor, or on a failed synchronization request. Thus, a given thread continues to execute as long as its memory requests hit in the cache or can be serviced by a local memory module, and as long as synchronization attempts are successful. Block multithreading thus allows a single thread to benefit from the maximum performance of the processor. For multithreading to be useful in tolerating latency, however, the time required to switch to another thread must be shorter than the time to service a remote request. This requires multiple register sets or some other hardware-supported mechanism.

Efficient message interface. An efficient message interface that allows the processor to access the interconnection network directly makes some parallel operations significantly more efficient than if they were implemented solely with shared-memory operations. Examples include remote thread creation and barrier synchronization. With a fast message in Alewife, we can create a thread on a remote processor in 7 μs. Restricting ourselves to shared-memory operations, remote thread creation takes 24 μs. Kranz and associates[9] have studied the importance of an efficient message interface in a shared-memory setting.

In Sparcle, we accomplish a fast message send operation by using the cache bus and coprocessor interface to store data in registers directly into the network, and to load data from the network directly into registers. Two new load/store instructions handle the loading and storing. Sparcle also supports direct memory access for larger messages.

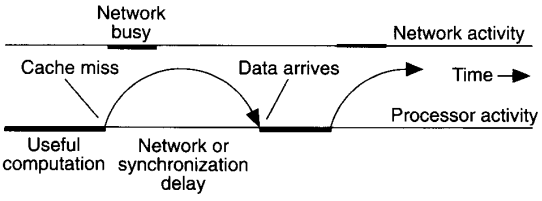


Figure 3. Processor and network activity when a single thread executes on the processor and no latency tolerance mechanisms are employed.

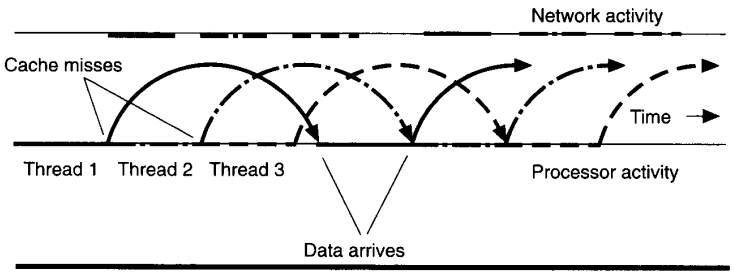


Figure 4. Processor and network activity when multiple threads execute on the processor and fast context switching is used for latency tolerance.

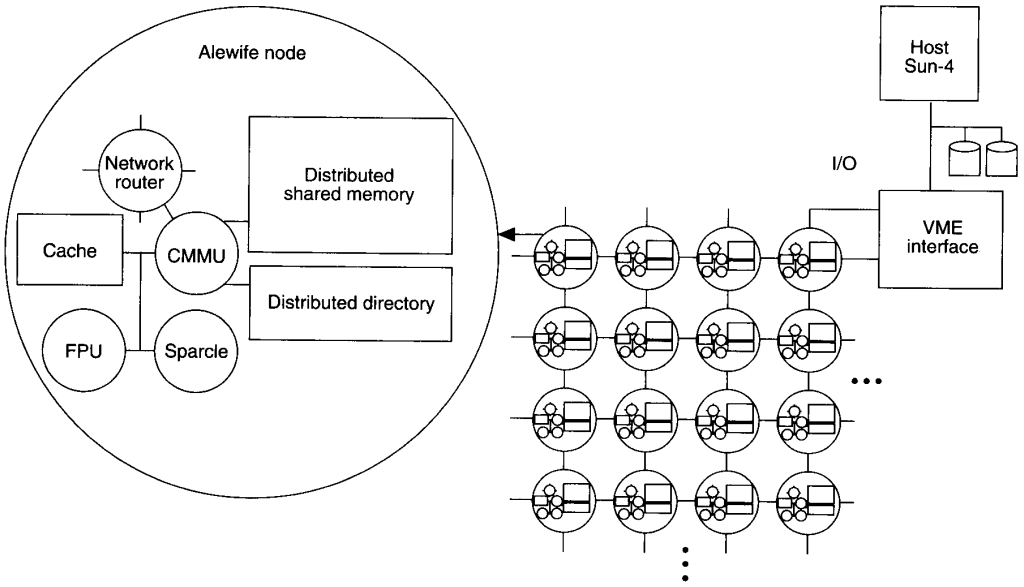


Figure 5. Structure of the Alewife machine.

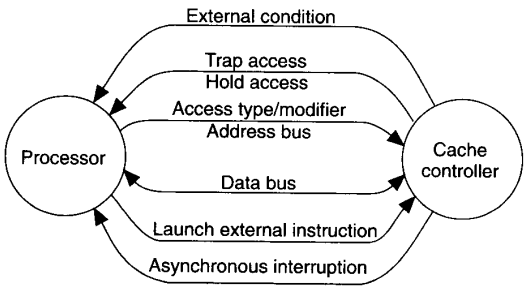


Figure 6. Interface between the processor pipeline and memory controller.

Alewife machine interfaces

The Sparcle chip is part of a complete multiprocessing system. It serves as the CPU for the Alewife machine[2]—a distributed shared-memory multiprocessor with up to 512 nodes and hardware-supported cache coherence. Figure 5 depicts the Alewife machine as a set of processing nodes connected in a mesh topology. Each Alewife node consists of a processor, a 64-Kbyte cache, a 4-Mbyte portion of globally-shared distributed memory, a CMMU, a floating-point coprocessor, and a network switch. An additional 4 Mbytes of local memory holds the coherence directory, code, and local data. The network switch chip is an Elko-series mesh routing chip (EMRC) from Caltech that has 8-bit channels. The network operates asynchronously with a switching delay of 30 ns per hop and 60 Mbytes/s through bidirectional channels.

The single-chip CMMU performs a number of tasks, including cache management, DRAM refresh and control, message queuing, remote memory access, and direct memory access. It also supports the LimitLESS cache-coherence protocol,[10] which maintains a few pointers per memory block in hardware (up to five in Alewife) and emulates additional pointers in software when needed. Through this protocol, all the caches in the system maintain a coherent view of global memory.

Sparcle implements a powerful and flexible interface to the CMMU. As depicted in Figure 6, this interface couples the processor pipeline with the CMMU. The interface can be divided into two general classes of signals: flexible data access mechanisms and flexible instruction extension mechanisms.

Together, the Access Type, Address Bus, Data Bus, and Hold Access line form the nucleus of data access mechanisms and comprise a standard external cache interface. To permit the construction of other types of data accesses for synchronization, we have supplemented this basic interface with three classes of signals:

- A *Modifier* that is part of the operation code for load/

store instructions and that is *not* interpreted by the core processor pipeline. The modifier provides several "flavors" of load/store instructions.

- Two *External Conditions* that return information about the last access. They can affect the flow of control through special branch instructions.
- Several vectored memory exception signals (denoted *Trap Access* in the figure). These synchronous trap lines can abort active load/store operations and can invoke function-specific trap handlers.

These mechanisms permit us to extend the load/store architecture of a simple RISC pipeline with a powerful set of operations.

An instruction extension mechanism permits us to augment the basic instruction set with external functional units. Instructions that are added in this way can be pipelined in the same fashion as standard instructions. To make this work, Sparcle reserves a special range of opcodes for external instructions. Also, the memory controller fetches new instructions from the cache bus at the same time that the processor does. Consequently, when the processor decodes an instruction in this range, it asserts the Launch External Inst signal, telling the CMMU to begin execution of the last fetched instruction. Note that the coprocessor interfaces of several microprocessors already provide this functionality.

We contend that we can design such a powerful interface between the processor pipeline and the communications and memory management hardware without significantly modifying the core RISC pipeline of contemporary processors. With this interface in mind, we first discuss several efficient multiprocessor mechanisms that are provided by the Sparcle processor. Later we touch upon the support which the memory controller must provide for these mechanisms.

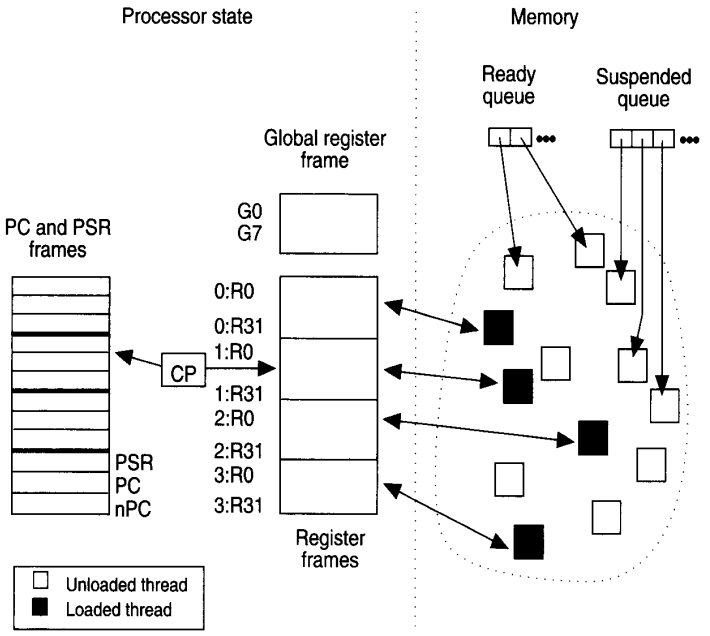


Figure 7. Block multithreading and virtual threads.

Sparcle architecture and implementation

Sparcle is best described as a conventional RISC microprocessor with a few additional features to support multiprocessing. These features include support for latency tolerance, support for fine-grain synchronization, and support for fast message handling. Before we describe how we implemented them in the Sparc processor, we need to discuss these features. Then we can indicate how they can also be implemented in other RISC microprocessors.

Mechanisms for latency tolerance. Figure 7 illustrates fast context switching on a generic processor. This diagram shows four separate register sets with associated program counters and status registers. Each register set represents a context. A hardware register called the context pointer or CP points to the active context. Consequently, a hardware context switch requires only that the context pointer be altered to point to another context. (Depending on details of the implementation, some number of cycles may be needed to flush the pipeline before executing a new context.) This figure also shows four threads actively loaded in the processor. These four threads are part of a much larger set of runnable and suspended threads that the runtime system maintains.

Implementation of fast context switching in Sparc. In a similar fashion, Sparcle uses multiple register sets to implement fast context switching. The particular Sparc design that we modified has eight overlapping register windows. Rather than using the register windows as a register stack, we used them in pairs to represent four independent, nonoverlapping contexts. We use one as a context for trap and message handlers, as described by Dally et al.[11] and Seitz et al.,[12] and the other three for user threads. The Sparc current window pointer (CWP) serves as the context pointer. Further, the window

```
RDPSR  R16        ; Save PSR in reserved register.
NEXTF  R0, R0, R0 ; Move to next active context.
WRPSR  R16        ; Restore PSR from other context.
JMPL   R17, R0    ; Restore PC
RETT   R18, R0    ; Restore nPC and return from trap.
```

Figure 8. Context switch trap code for Sparcle

Cycle	Operation
0	Fetch of data instruction (load or store)
1	Decode of data instruction (load or store)
2	Execute instruction (compute address)
3	Data cycle (which will fail)
→4	Pipeline freeze, indicate exception to processor
5	Pipeline flush (save PC)
6	Pipeline flush (save nPC, decrease CWP)
7	Fetch: RDPSR PSRREG (save PSR in reserved register)
8	Fetch: NEXTF (advance CWP to next active context using WIM)
9	Fetch: WRPSR PSRREG (restore PSR for new context)
10	Fetch: JMPL R17 (load PC, return from trap and)
11	Fetch: RETT R18 (reexecute trapping instruction)
12	Dead cycle from JMPL
13	First fetch of new instruction
14	Dead cycle from RETT (folded into switch time)

Figure 9. Anatomy of a context switch in Sparcle.

invalid mask (WIM) indicates which contexts are disabled and which are active. This particular use of register windows does not involve any modifications, just a change in software conventions.

Unfortunately, the Sparc processor does not have four sets of program counters and status registers. Since adding such facilities would impact the pipeline significantly, we implemented rapid context switching via a special trap with an extremely short trap handler. Thus, when the processor attempts to access a remote memory location that is not in the local cache, the CMMU causes a synchronous memory fault to Sparcle, while simultaneously sending a request for data to the remote node. The trap handler then saves the old program counter and status register, switches to a new context, restores a new program counter and status register, returns from the trap to begin execution in the new context.

With the goal of shortening this trap handler as much as possible, we made the following modifications to the Sparc architecture:

- So that the processor traps immediately to the context-switch code without having to decode the trap type, we added an extra synchronous trap line (with corresponding trap vector).
- We added a new instruction called NEXTF. It is much like the Sparc SAVE instruction except that the window pointer is advanced to the next active context as indicated by the window invalid mask register. If no additional contexts are active, it leaves the window pointer unchanged.
- We increased the number of instructions for each entry in the Sparc trap vector from 4 to 16. This allows the context switch and other small trap handlers to execute in the trap vector directly.
- We made the value of the current window pointer available on external pins. Among other things, this permits the emulation of multiple hardware contexts in the Sparc floating-point unit by modifying floating-point instructions in a context-dependent fashion as they are loaded into the FPU and by maintaining four different sets of condition bits. Consequently, the context-switch trap handler does not have to worry about the FPU.

Figure 8 shows the context-switch trap handler with these changes. When the trap occurs, Sparcle switches *one* window backward (as does a normal Sparc). This switch places the window pointer *between* active contexts, where the Alewife runtime system reserves a few registers for the context state. As with normal Sparc trapping behavior, the hardware writes the PC and nPC to registers R17 and R18. This trap code places the processor status register (PSR) in register R16.

As depicted in Figure 9, the net effect is that a Sparcle context switch takes 14 cycles. This illustrates the total penalty for a context-switch on a data instruction. Note that, while this diagram shows 15 cycles, one of them is the fetch of the first instruction from the next context.

By maintaining a separate PC and processor status register for each context, a more aggressive processor design could switch contexts much faster. However, even with 14 cycles of overhead and four processor-resident contexts, multithreading can significantly improve system performance.[13,14]

Support for fine-grain synchronization. As discussed earlier, fine-grain data-level synchronization is expressed with J- and L-structures and implemented using new instructions that interact with full/empty bits in memory. Sparc implements the new load, store, and swap instructions using the Sparc alternate address space instructions. We have modified these instructions in two ways:

1. The load, store, and swap alternate space instructions in Sparcle are unprivileged for ASI values in the range 0×80 to $0 \times FF$. They remain privileged for ASI values less than 0×80. The CMMU uses the ASI value as an extended opcode; that is, ASI 0×84 corresponds to the load and

trap if empty operation. This allows user code to interact directly with full/empty bits.

2. We have used several new opcodes to produce specific ASIs on the Sparcle output pins while allowing the register + offset addressing mode. The normal load/store ASI instructions only allow register + register addressing.

A new dedicated synchronous trap line carries full/empty trap signals. J- and L-structure operations are implemented with the following special load/store instructions:

LDN	Read location
LDEN	Read location and set to empty
LDT	Read location if full, else trap
LDET	Read location and set to empty if full, else trap
STN	Write location
STFN	Write location and set to full
STT	Write location if empty, else trap
STFT	Write location and set to full if empty, else trap

In addition to possible trapping behavior, each of these instructions sets a coprocessor condition code to the state of the full/empty bit at the time the instruction starts execution. Either trapping or an explicit test of this condition code will detect a synchronization failure. When a trap occurs, the trap handling software decides what action to take.

Implementation of J-structures. To demonstrate how the special load/store instructions can be used, we will describe how we implement J-structures and present the cycle counts for various synchronizing operations. Sparcle implements a J-structure allocation by allocating a block of memory with the full/empty bit for each word set to empty. Resetting a J-structure element involves setting the full/empty bit for that element to empty. Implementing a J-structure read operation is also straightforward: it is a memory read that traps if the full/empty bit is empty. Sparcle implements it with a single instruction:

```
LDT (R1),R2 ; R1 points to J-structure location
```

If the full/empty bit is empty, the reading thread may need to suspend execution and queue itself on a wait queue associated with the empty element. To minimize memory usage, we use a single memory location to represent *both* the value of the J-structure element and the wait queue. This implies that we need to associate two bits of state with each J-structure element: whether the element is full or empty and whether the wait queue is locked or not.

```
MOVE   $0, R3      ; set up swap register.
SWAPT  R3, (R1)    ; swap zero with J-structure location, trap if full.
CMP    $–1, R3     ; check if queue is empty.
BEG, a %done       ; branch if no waiters to wake up.
STFT   R2, (R1)    ; write value and set to full (delay slot).
   ⋮
<wake up waiters and store value>
   ⋮
%done
```

Figure 10. Machine code implementing a J-structure write.

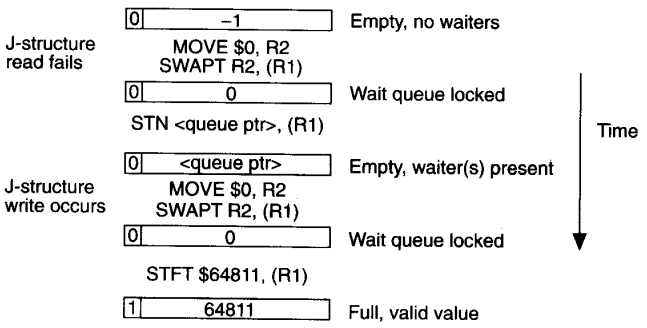


Figure 11. Reading and writing a J-structure slot.

Other architectures implement these two state bits directly in hardware by having multiple state bits per memory location.[15,16] Instead of providing an additional hardware bit, we take advantage of Sparc's atomic register-memory swap operation. Since the writer of a J-structure element knows that the element is empty before it does the write operation, it can use the atomic swap to synchronize access to the wait queue. With this approach, a single full/empty bit is sufficient for each J-structure element. A writer needs to check explicitly for waiters before undertaking the write operation.

Using atomic swap and full/empty bits, the machine code in Figure 10 implements a J-structure write. In this figure, R1 contains the address of the J-structure location to be written to, and R2 contains the value to be written. Also, –1 is the end of the queue marker, and 0 in an empty location means that the queue is locked. Compared with the hardware approach, this implementation costs an extra move, swap, compare, and branch to check for waiters. However, we believe that the reduction in hardware complexity is worth the extra instructions.

Figure 11 gives a scenario of accesses to a J-structure location under this implementation and illustrates the possible states of a J-structure slot. Here, R1 contains a pointer to the J-structure slot.

Table 1. Summary of fast-path costs of J-structure and L-structure operations, compared with normal array operations.

Element	Action	Instructions	Cycles
Array	Read	1	2
	Write	1	3
J-structure	Read	1	2
	Write	5	10
	Reset	1	3
L-structure	Read	2	5
	Write	5	10
	Peek	1	2

```
STIO    R2, $ipiout0 ; Store header.
STIO    R3, $ipiout1 ; Store data word.
STIO    R4, $ipiout2 ; Store address of data.
STIO    R5, $ipiout3 ; Store length of data.
IPILAUNCH 2, 1       ; Launch message. Descriptor is 2 double-
                     ; words long and contains 1 double-word
                     ; of explicit data (from R2 and R3).
```

Figure 12. Machine code implementing a message send.

Table 1 summarizes the instruction and cycle counts of J-structure and L-structure operations for the case where no waiting is needed on read operations and no waiters are present on write operations. In Sparcle, as in the LSI Logic Sparc, normal read operations take two cycles and normal write operations take three cycles, assuming cache hits. A locking read is considered a write and thus takes three cycles.

Support for futures and placeholders. To support futures and placeholders, Sparcle provides automatic and efficient detection and handling of placeholders via traps. Two Sparcle modifications are involved.

First, to detect placeholders, Sparcle adds two new instructions called NTADD and NTSUB. These instructions cause tag overflow traps whenever the low bit of either of their operands is set. (NTADD and NTSUB are modifications of the Sparc tagged instructions TADDCCTV and TSUBCCTV that trap whenever the low two bits of either of their operands are set.) As discussed earlier, only pointers to placeholders have the low bit set. With tag overflow traps, NTADD and NTSUB automatically detect placeholders in add, subtract, and compare operations. The address alignment trap in Sparcle detects placeholders in pointer dereferencing operations.

Second, to efficiently handle traps caused by placeholders, the trap vector number that is generated by tag overflow and address alignment traps depends on the register containing the placeholder. This feature saves the trap handler from having to waste cycles decoding the trapping instruction to find out which register contains the offending placeholder. Johnson[17] and Ungar et al.[18] have proposed similar mechanisms.

Fast message handling. Most distributed shared-memory machines are built on top of an underlying message-passing substrate. Traditional shared-memory machines provide a layer of hardware that implements some coherence protocol between the processor and the interconnection network. It is natural, then, to provide the processor with direct access to the network in addition to the shared-memory interface because many operations benefit greatly from direct network access. Sparcle supports sending and receiving messages via a memory-mapped interface to the interconnection network.

Send. Sparcle sends messages through a two-phase process: first describe, then launch. Sparcle composes a message by writing directly to the interconnection network queue using a special store instruction called STIO (for store IO). The queues are memory mapped as an array of network registers in the CMMU, called the output descriptor array. In terms of performance, write operations into this array incur the same cost as write hits into the cache.

The first word of the message must be a header indicating a message opcode and the destination node. Sparcle reserves a range of opcodes for privileged use by the operating system. The rest of the message can contain immediate values from registers, or address and length pairs which invoke DMA on blocks from memory.

After the message is composed, a coprocessor instruction launches the message. Figure 12 illustrates the sending of a single message with one data word and one block of data from memory. In addition to the required header, this message includes one explicit data word and one block of data from memory. On entry to this code sequence, register R2 contains the header, R3 contains the data word, R4 the address of the data block, and R5 the length of the data block. If Sparcle is in the user mode and the header is privileged, an exception will occur. The CMMU maintains the atomicity of messages as described in the next section.

Receive. A message arrival causes a trap. The trap handler can either load words directly from the incoming message into registers using a special load instruction called LDIO (for load IO) or initiate a DMA sequence to store the message into memory. If the latter option is chosen, the processor can direct the CMMU to generate an interrupt after the storeback is complete.

Support for message handling. The following features of Sparcle support messaging:

- Special user-level load/store instructions allow fast composition of outgoing messages and fast examination of

incoming messages. An ASI value is reserved for the transferring of data to and from message register values. This ASI is produced by two new Sparcle instructions, STIO and LDIO. Although these instructions support a memory-mapped interface to the network registers, addresses for the message queues fit completely into the address offset field. Consequently, the compiler can generate instructions that perform direct register-to-register moves between the processor and the network queues.

- Register windows permit fast processing of message interrupts. One of the four hardware contexts is reserved for message processing. Consequently, the message interrupt handler needs only to alter the current window pointer so that this special context is active. No registers need to be saved and restored.
- Coprocessor instructions for message launch and disposal permit pipelining of network operations. Further, opcode bits in the launch and disposal instructions contain information about the format of messages that are about to be sent or received into memory. Thus, message format is completely under control of the compiler. Finally, the coprocessor interface permits a precise identification of the commit point for launch instructions, ensuring that message launches are atomic.
- Fast interrupt operations allow rapid entry into message handler code on the arrival of a message. In our current implementation, because interrupts always force the processor into the supervisor mode, user-level receipt of messages requires a few extra cycles for the processor to transfer control to user code. In a more aggressive implementation, the processor would support a user-level return from trap.

The CMMU interface

From this discussion we can clearly see that the Sparcle processor is part of a complete system. Consequently, several of the mechanisms that were included in Sparcle are incomplete without the support of the CMMU. Here we briefly discuss the Alewife CMMU and how it interfaces to Sparcle. Although the Alewife CMMU provides a number of features, we focus on the cache controller and message interface.

Earlier, under Alewife machine interfaces, we discussed two categories of signals in the interface between processor and CMMU: flexible data access mechanisms and flexible instruction extension mechanisms. Figure 13 makes this interface more concrete by showing Sparcle equivalent names for all of the signals. Each signal in this figure corresponds directly to signals in Figure 6.

A few of the data access mechanisms require further discussion. The modifier is implemented with the Sparc ASI field. Again, Sparcle contains a number of new load/store instructions that differ only by the values that they place on the ASI pins during data cycles. These new load/store instructions are important to the implementation of full/empty bit synchronization and fast messages. The trap access signals are new versions of the Sparc memory exception signal MEXC, which have distinct trap vectors. These invoke context-switch and synchronization traps. The external condition bits are implemented through the Sparc coprocessor condition codes (CCC); consequently, "branch on condition-code" instructions in Sparc can be used to examine them.

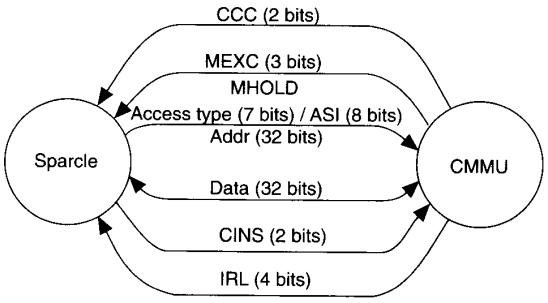


Figure 13. Sparcle signal names.

Finally, the external instruction interface is implemented directly through a Sparc coprocessor interface. Sparcle asserts one of the CINS signals to indicate that a coprocessor instruction has been decoded by the processor and should be executed by the coprocessor. Two CINS signals are required because pipeline interlocks can occasionally cause the instruction fetch unit to get ahead of the rest of the pipeline.

Latency tolerance. We already discussed rapid context switching for latency tolerance from the standpoint of the Sparcle processor. In addition to those Sparcle mechanisms, the cache controller must be able to handle multiple outstanding requests. This involves the ability to handle split-phase memory transactions (separating the request for data from the response) and to place returning data into the cache while the processor is performing some other task. Consequently, when the processor requests a data item that is not in the local cache, the cache controller asserts the appropriate trap line to initiate execution of the context-switch trap handler. At the same time, it sends a request message to the particular node that contains the requested data. Note that the mechanisms required to handle context switching differ little from those required for software prefetching. (However, see Kubiatowicz, Chaiken, and Agarwal[19] for some interesting forward-progress issues.)

Full/empty-bit synchronization. Full/empty-bit synchronization, as implemented in Alewife, requires support from the cache controller. Since full/empty-bit synchronization employs one synchronization bit for each data word, extra

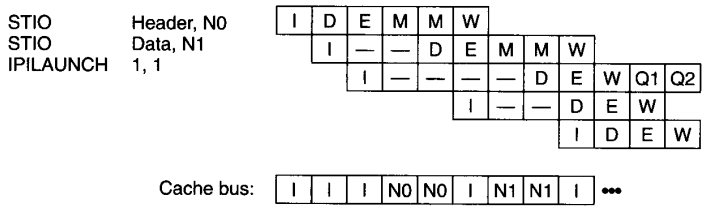


Figure 14. Pipelining for transmission of a message with a single data word.

storage must be reserved for these bits in the cache system. While these bits logically belong with the cache data, the Alewife CMMU implements them with the cache tags. This has a number of advantages. It eliminates a need for an odd number of bits in the physical memory used for cache data. It also makes access to the tags file much faster than access to the cache data, both because the tags file is smaller and because no chip crossings are required. This permits synchronization operations to occur in parallel with processing of the cache tags.

Of the Sparcle mechanisms, those important to full/empty synchronization are the external condition code, the access modifier (ASI), and one of the extra trap lines. All of the new synchronizing load/store instructions mentioned earlier are distinguished by the value of the ASI field that they generate (and whether they are read or write operations). For each data access, the Alewife CMMU takes the proffered ASI value along with the address and type of access. The CMMU uses the address to index into the tags file, retrieving both the tag and the appropriate full/empty bit. Simultaneously, it decodes the ASI value to produce two different actions, one which will be taken if the full/empty bit is full, and one if the full/empty bit is empty. When the tag lookup is completed, the CMMU completes both tags match and full/empty-bit operations simultaneously, either flagging a context-switch (on cache miss), a synchronization fault, or successful completion of the access. In all cases, the CMMU places the full/empty bit that was first retrieved from the tags file in one of the external condition codes for future examination by the processor.

The support that Alewife provides for full/empty-bit synchronization is external to the processor pipeline: that is, it occurs at the first-level cache. Consequently, full/empty bits never enter the processor core. Further, individual load/store instructions have varied semantics with respect to the full/empty bit: some cause test-and-set-like operations; others invoke traps. This places some data processing logic within the first-level cache. For modern processors that have one level of on-chip caching, a closer integration between the processor pipeline and full/empty bit synchronization might be desirable. This could include widening of internal processor registers and use of special full/empty-bit synchronization instructions that are sandwiched between Alpha-style[20] load-locked/store-conditional synchronization instructions.

Fast message handling. Fast messaging in Alewife relies on a number of features in the CMMU. All of the network queuing and DMA mechanisms are a part of this chip. Sparcle interfaces with these mechanisms through both the external instruction interface and through special loads and stores. As we discussed, Sparcle reserves one special load/store instruction (and corresponding ASI) for rapid descriptions of outgoing messages and rapid examination of incoming messages. The cache controller recognizes accesses with this ASI and causes data transfer to and from message queues instead of the cache. Message data thus transfers between the processor and network at the same speed as cached accesses.

Alewife uses the external instruction interface to implement the message launch mechanism. Consequently, message launches can be pipelined. Figure 14 gives a simple pipeline example. Here, the two-cycle latency for stores and the lack of an instruction cache limit the message throughput. More aggressive processor implementations would not suffer from this limitation. In this figure, Sparcle pipeline stages are Instruction fetch, decode, execute, memory, and writeback. Network messages are committed in the writeback stage. Stages Q1 and Q2 are network queuing cycles. The message data begins to appear in the network after stage Q2. Note that the use of DMA on message output adds additional cycles (not shown in the figure) to the network pipeline.

The close coupling between the message launch mechanism and the processor pipeline allows us to identify a precise launch completion point (corresponding to the writeback stage of the launch instruction). As a result, message launches are atomic. Before the launch instruction commits, no data is placed into the network. After the launch commits, Alewife sends a complete output packet to the network. These atomic semantics allow multiple levels of user and interrupt code to share a single network output port without requiring that the user disable interrupts before beginning to describe a message.

THE SPARCLE CHIP INCORPORATES MECHANISMS required for massively parallel systems in a Sparc RISC core. Coupled with a CMMU, Sparcle allows a fast, 14-cycle context switch, an 8-cycle user-level message send, and fine-grain full/empty-bit synchronization.

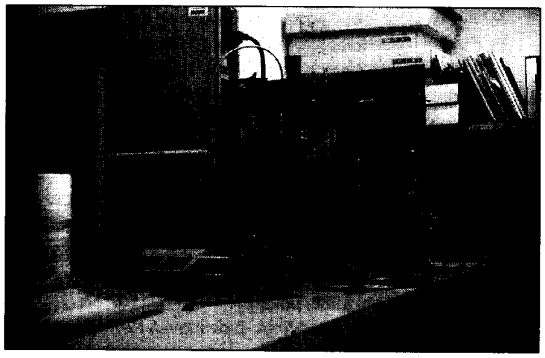


Figure 15. Sparcle's test system.

Before we received working Sparcle chips from LSI in the spring of 1992, we used an operating single-node test system. Also operational for several months was a compiler and a runtime system for our parallel versions of C and Lisp. The test system shown in Figure 15 comprises 256 Kbytes of static RAM memory, an I/O interface to the VMEbus for downloading programs and monitoring execution, and control logic to exercise the full/empty bit and context switching functionality. We had debugged the test system using Sparcs in place of Sparcles; it operated at a maximum clock frequency of about 25 MHz. (Sparc and Sparcle have only a few differing pins, and Sparcle even provides an input signal Mode pin that allows switching between Sparc and Sparcle modes.)

We have been running several parallel programs, including Sparcle's runtime system, to exercise all of Sparcle's functionality, at the maximum speed of the test bed. Scope measurements of critical signal timings on the chip's pins suggest we will be able to run the chips in an Alewife node board at roughly the same speed as the original, unmodified Sparcs.

Implementation of the Sparcle development relied on modifying an existing design through a unique collaboration with industry. Although we had our moments of trepidation, given the number of participants and the multiple failure modes (both technical and political), we believe this model of experimentation has been very successful. This implementation strategy not only allowed us, at a university, to experiment with architectural ideas in a real, contemporary processor design, it also significantly reduced the design effort from the concept stage to working chip.

Figure 16 depicts the resulting project schedule for Sparcle. We defined Sparcle's early architecture in April 1989. At MIT we also wrote a Sparcle compiler for a version of Lisp and implemented a cycle-by-cycle simulator. Later, we also developed a compiler for a parallel version of C. By March 1990, we had developed a detailed specification of the modifications to Sparc required to implement Sparcle. Then, Sun made high-level changes to Sparc functional blocks, and LSI made lower gate-level changes. We tested these changes against Sparcle binaries produced at MIT. Then LSI synthesized netlists and MIT tested them against several hundred thousands of test vectors. The test vectors included both Sparc vectors provided by LSI and Sparcle vectors obtained from the MIT Sparcle simulator. The test setup included a netlist module for the floating-point coprocessor and a behavioral model for the rest of the memory and communication systems. Finally, LSI undertook layout and fabrication, during which time we also implemented a test system for Sparcle.

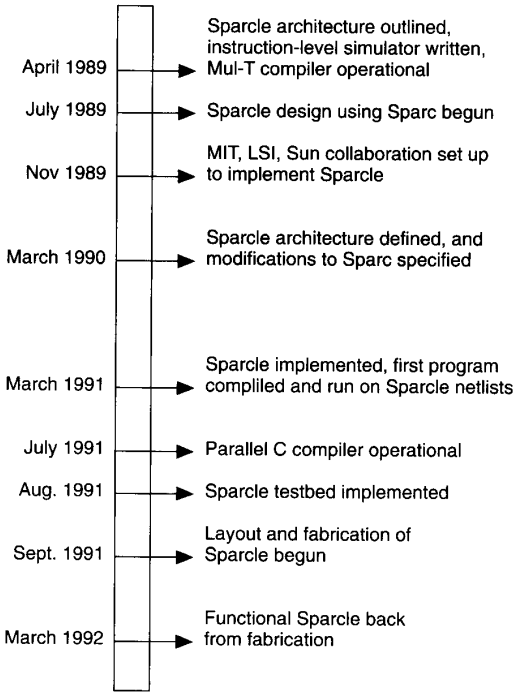


Figure 16. Sparcle's implementation schedule.

While the Sparcle chip project demonstrates that a contemporary RISC microprocessor can readily incorporate features considered by many to be critical for massively parallel multiprocessing, the end systems benefit of these mechanisms can only be evaluated in the context of a complete multiprocessor system. We are in the final stages of implementing the Sparcle-based Alewife multiprocessor system. Figure 1 shows an Alewife node board with the Sparcle and FPU. Figure 17 shows a 16-node Alewife system package developed by the Advanced Production Technology group

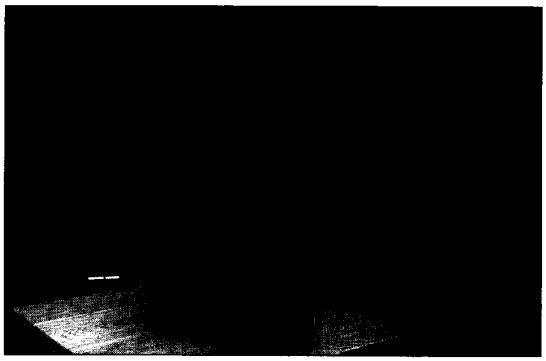


Figure 17. The 16-node Alewife package.

at the Information Sciences Institute in Los Angeles. The CMMU chip has been implemented and tested. It is being implemented in LSI Logic's LEA 300K process, and we expect to begin its fabrication shortly.

Acknowledgments

The Sparcle project is funded in part by DARPA contract N00014-87-K-0825 and in part by National Science Foundation grant MIP-9012773. LSI Logic and Sun Microsystems helped implement Sparcle, and LSI Logic supported the fabrication of Sparcle. We acknowledge the contributions of Dan Nussbaum, who was partly responsible for the processor simulator and runtime system, and was the source of several ideas. Halstead's work on multithreaded processors influenced our design. Our research also benefited significantly from discussions with Bert Halstead, Tom Knight, Greg Papadopoulos, Juan Loaiza, Bill Dally, Steve Ward, Rishiyur Nikhil, Arvind, and John Hennessy.

References

1. *Sparc Architecture Manual*, Sun Microsystems, Mountain View, Calif., 1988.
2. A. Agarwal et al., "The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor," *Proc. Workshop on Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, Boston, 1991. An extended version of this paper has been submitted for publication and appears as MIT/LCS Memo TM-454, 1991.
3. A. Agarwal et al., "APRIL: A Processor Architecture for Multiprocessing," *Proc. 17th Ann. Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., June 1990, pp. 104-114.
4. B.J. Smith, "Architecture and Applications of the HEP Multiprocessor Computer System," *Soc. of Photooptical Instrumentation Engineers*, Bellingham, Wash., Vol. 298, 1981, pp 241-248.
5. Arvind, R.S. Nikhil, and K.K. Pingali, "I-Structures: Data Structures for Parallel Computing," *Trans. on Programming Languages and Systems*, Vol. 11, No. 4, Oct. 1989, ACM Press, pp. 598-632.
6. M. Dubois, C. Scheurich, and F.A. Briggs, "Synchronization, Coherence, and Event Ordering in Multiprocessors," *Computer*, Vol. 21, No. 2, Feb. 1988, pp. 9-21.
7. S. V. Adve and M.D. Hill. "Weak Ordering—A New Definition," *Proc. 17th Ann. Int'l Symp. Computer Architecture*, IEEE CS Press, June 1990, pp 2-14.
8. D. Lenoski et al., "The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor," *Proc. 17th Ann. Int'l Symp. Computer Architecture*, IEEE CS Press, June 1990, pp. 148-159.
9. D. Kranz et al., "Integrating Message-Passing and Shared-Memory; Early Experience," to be published in *Conf. Principles and Practice of Parallel Programming*, ACM, May 1993, and appears as MIT/LCS Memo TM-478, 1993.
10. D. Chaiken, J. Kubiatowicz, and A. Agarwal, "LimitLESS Directories: A Scalable Cache Coherence Scheme," *Proc. Fourth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, ACM, Apr. 1991, pp. 224-234.
11. W.J. Dally et al. "The J-Machine: A Fine-Grain Concurrent Computer," *Proc. Int'l Federation for Information Processing (IFIP) 11th World Congress*, Elsevier Scientific Publishing, New York, 1989, pp. 1147-1153.
12. C.L. Seitz et al., "The Design of the Caltech Mosaic C Multicomputer," *Proc. 1993 Symp. Research on Integrated Systems*, G. Borriello and C. Ebeling, eds., MIT Press, Cambridge, Mass., 1993, pp. 1-22.
13. W-D. Weber and A. Gupta, "Exploring the Benefits of Multiple Hardware Contexts in a Multiprocessor Architecture: Preliminary Results," *Proc. 16th Ann. Int'l Symp. Computer Architecture*, IEEE CS Press, June 1989, pp. 273-280.
14. K. Kurihara, D. Chaiken, and A. Agarwal, "Latency Tolerance Through Multithreading in Large-Scale Multiprocessors," *Proc. Int'l Symp. Shared Memory Multiprocessing*, IPS Press, Japan, Apr. 1991, pp. 91-101.
15. G. Alverson, R. Alverson, and D. Callahan, "Exploiting Heterogeneous Parallelism on a Multithreaded Multiprocessor," *Workshop on Multithreaded Computers, Proc. Supercomputing*, ACM Sigraph, Nov. 1991.
16. G.M. Papadopoulos and D.E. Culler, "Monsoon: An Explicit Token-Store Architecture," *Proc. 17th Ann. Int'l Symp. Computer Architecture*, IEEE CS Press, June 1990, pp. 82-91.
17. D. Johnson, "Trap Architectures for Lisp Systems," *Proc. 1990 ACM Conf. on Lisp and Functional Programming*, 1990, pp. 79-96.
18. D. Ungar et al., "Architecture of SOAR: Smalltalk on a RISC," *Proc. 1984 Int'l Symp. on Computer Architecture*, 1984, pp. 188-197.
19. J. Kubiatowicz, D. Chaiken, and A. Agarwal, "Closing the Window

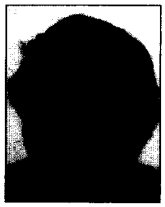
of Vulnerability in Multiphase Memory Transactions," *Fifth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*. ACM, Oct. 1992, pp. 274-284.
20. *Alpha Architecture Reference Manual*, Digital Press, Maynard, Mass., 1992.

Anant Agarwal serves at the Laboratory for Computer Science at MIT where he is an associate professor of electrical engineering and computer science. His current research interests include the design of scalable multiprocessor systems, VLSI processors, compilation and runtime technologies for parallel processing, and performance evaluation. At Stanford University, he participated in the MIPS and MIPS-X projects. He initiated the Alewife project at MIT, which is aimed at the design and implementation of a large-scale cache-coherent multiprocessor.

Agarwal received a B Tech in electrical engineering from the Indian Institute of Technology, Madras, India, and an MS and PhD in electrical engineering from Stanford. He is a member of the ACM and IEEE Computer Society.

John Kubiatowicz is a doctoral candidate in the Department of Electrical and Computer Science at MIT. His current research interests include parallel computer architecture, high-performance microprocessor design, and high-energy particle physics.

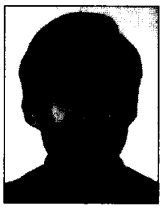
Kubiatowicz received BS degrees in electrical engineering and physics and an MS in electrical engineering from MIT.



David Kranz has been a research associate in the MIT Laboratory for Computer Science since 1987. His research interests are in programming language design and implementation for parallel computing.

Kranz received a BA from Swarthmore. While earning a PhD at Yale, he worked on high-performance compilers for Scheme and applicative languages.

Beng-Hong Lim is currently a doctoral candidate in MIT's Department of Electrical Engineering and Computer Science. His research interests include parallel computing and computer architecture. He received a BS and an MS in electrical engineering and computer science from MIT.



Donald Yeung is a PhD candidate at MIT. His research interests are in the area of multiprocessor design, including efficient hardware and software mechanisms for synchronization and latency tolerance.

Yeung received a BS from Stanford in computer systems engineering, and recently completed an MS in electrical engineering and computer science at MIT.

Godfrey D'Souza has been with the Sparc Systems Division and the CoreWare Group at LSI Logic where he is a senior design engineer involved with aspects of microprocessor and system level design.

D'Souza received a BS in electronics and communications engineering from the University of Baroda, India, and an MS in electrical engineering from the University of Washington, Seattle. He is a member of the IEEE Computer Society.

Mike Parkin is a staff engineer at Sun Microsystems, where he is currently part of a research team that is building a Sparc-based scalable multiprocessor system.

Parkin received a BSEE from Iowa State University and an MSEE from Stanford.

Direct questions concerning this article to David Kranz, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, or via e-mail at kranz@lcs.mit.edu.