# Sparcle: An Evolutionary Processor Design for Large-Scale Multiprocessors

Anant Agarwal, John Kubiatowicz, David Kranz,
Beng-Hong Lim, Donald Yeung, Godfrey D'Souza, and Mike Parkin
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

12 March 1993

**Abstract**

Sparcle is a processor chip developed jointly by MIT, LSI Logic, and SUN Microsystems, by evolving an existing RISC architecture towards a processor suited for large-scale multiprocessors. Sparcle 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.

## 1 Introduction

The Sparcle chip will clock at no more than 50 MHz, has no more than 200K 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 day 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.** A clean communications interface between the processor and the communications network is not supported in current microprocessor designs. 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 thesis of the Sparcle chip project is that a processor providing interfaces for the above mechanisms can be implemented with small modifications to an existing microprocessor. Indeed, Sparcle is derived from the SPARC [17] architecture, and is being integrated into Alewife [3, 2], 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. With slightly more aggressive modifications, this number can be reduced to 4 cycles. Sparcle switches to another thread when a cache miss that requires service over the communications network occurs, or when a synchronization fault is detected. Such a processor requires a pipelined memory and communications system. In our system, a separate communications and memory management chip (CMMU) interfaces to Sparcle to provide the desired pipelined system interface. A software prefetch instruction is also provided. The modifications to a modern RISC microprocessor to achieve fast context switching are described in Section 4.1.

Sparcle supports fine-grain data-level synchronization through the use of full/empty bits, as in the HEP [16]. With full/empty bits, the probe of a lock and access of the data word protected by the lock can be accomplished in one operation. If the synchronization attempt fails, then the synchronization trap is invoked to handle the fault. In our system, the external communications chip is responsible for detecting synchronization faults and alerting Sparcle by raising a trap line. The fault is then handled in software trap code.

Finally, Sparcle supports a highly streamlined network interface with the ability to launch and receive interconnection network messages. While the communications interface with the interconnection network is implemented in a separate chip, the CMMU, future implementations can integrate this functionality into the processor chip. Sparcle provides support for rapid response to asynchronous events by streamlining SPARC's trap interface and providing support for dispatching rapidly to the appropriate trap handler. Sparcle achieves this by providing two special trap lines for the most common types of events, namely, cache misses to remote nodes and synchronization faults. A third trap line is used for all other types of events. Also, the number of instructions in each trap dispatch entry is increased so that vital trap codes can be inlined at the dispatch points.

Sparcle's design process was unusual in that it didn't involve developing a completely new architecture. Sparcle was implemented with the help of LSI Logic and SUN Microsystems by making slight modifications to 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 in our implementation effort of the Alewife multiprocessor which will allow a thorough evaluation of our ideas and an at-speed test of the Sparcle chips.

The rest of this paper is organized as follows. Section 2 discusses the software mechanisms that the Sparcle modifications support while Section 3 provides an overview of the Alewife machine interfaces. Section 4 describes the Sparcle architecture and discusses the modifications to the SPARC processor, and to RISC processors in general, to support the multiprocessor features. The CMMU interface to Sparcle for multiprocessor operation is discussed in Section 5. Section 6 presents the current status of the project and concludes the paper.

# 2 Mechanisms for Multiprocessors

In this section we motivate the features of Sparcle for supporting the widely used shared-memory and message-passing programming models. These features ease the programmer's job and enhance parallel program performance. We have implemented programming constructs in parallel versions of Lisp and C that use these features. We break the features down into three areas, where the first two are in support of 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 eases the programmer's job by relieving her from expending undue effort in trying to partition data and control 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** This is needed for supporting message-passing programs but it can also improve the performance of shared-memory programs in some common situations.

The rest of this section considers each of these areas in turn, and motivates why they are useful for large-scale multiprocessing. Section 4 will present details on the implementation of these features in Sparcle.

## 2.1 Fine-grain Computation

As multiprocessors scale in size, 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 ability to utilize a larger number of processors to speed up a program is limited by the overhead of parallel and synchronization operations. Systems supporting fine-grain parallelism and synchronization attempt to minimize this overhead so as to allow parallel programs to achieve better speedups.

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.

Fine-grain parallelism and synchronization can be expressed 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.

A J-structure is a data structure for producer-consumer style synchronization and was inspired by Arvind *et al.*'s I-structures [6]. It is like an array, but each element has 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

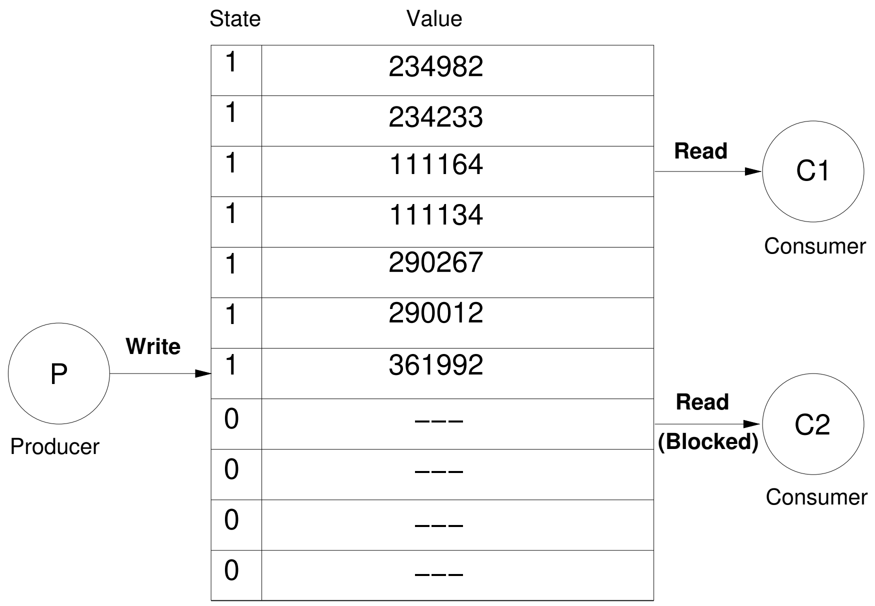


Figure 1: Example use of J-structures. Producer P is sequentially filling in the elements of a J-structure. Consumer C1 reads an element that is already filled and immediately gets its value. Consumer C2 reads an empty element and thus has to wait for P to write the element. Since we are synchronizing 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.

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 1 illustrates how J-structures can be used for data-level synchronization.

L-structures are similar to J-structures but support three operations: a locking read, a non-locking read, and a synchronizing write. A locking read waits until the element is full before emptying it (*i.e.*, locking it) and returning the value. A non-locking read 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. Thus, an L-structure allows mutually exclusive access to each of its elements, and allows multiple non-locking readers.

J- and L-structures, as well as other types of fine-grain data-level synchronization, are supported by Sparcle with per-word, full/empty bits in memory [16]. Sparcle provides new load and store instructions that interact with the full/empty bits. An extra synchronous trap line has also been added to deliver the full/empty trap. This extra line allows the trap to be immediately identified.

**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 associated with evaluating $X$ to be small without severely affecting performance.

If the compiler or run-time system chooses to create a new task to evaluate $X$, an object known as a *placeholder* is also created and returned as the value of the future expression. The placeholder is created in an *undetermined* state. Evaluation of $X$ yields its value and determines

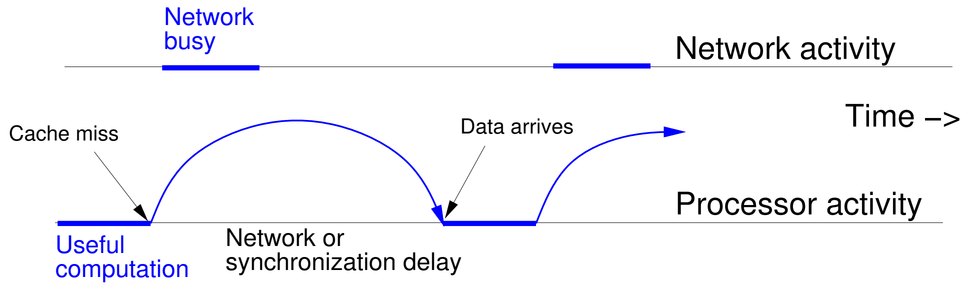


Figure 2: Processor and network activity when a single thread executes on the processor when no latency tolerance mechanisms are employed.

the placeholder. Any task that attempts to use the value of $X$ before the evaluation of $X$ has completed will encounter the undetermined placeholder and suspend until the placeholder is determined.

This functionality is implemented using (by software convention) the low bit of a data value as a placeholder tag, *i.e.* 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 placeholder is determined, the value can be placed in the target register and normal execution resumed. Otherwise, the trapping task waits until the value of the placeholder is available.

With this support, a compiler can generate code without knowing which data values may be computed concurrently. There is thus no runtime overhead for ensuring that operations on placeholders are detected.

### 2.2 Memory Latency Tolerance

Since memory in large-scale multiprocessors is distributed, cache misses to remote locations will incur long latencies and potentially reduce processor utilization. Figure 2 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 utilization. The use of latency tolerance mechanisms alleviates this problem and helps improve processor and network utilization.

The general class of solutions to latency tolerance have an important common feature: all solutions implement mechanisms for allowing 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 is not easily predictable at compile time, making it difficult for a compiler to schedule operations for maximal resource utilization. Thus, 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 for dynamic pipelining including prefetching and weak ordering [9, 1, 13].

Sparcle implements fast context switching as its primary mechanism for dynamic latency

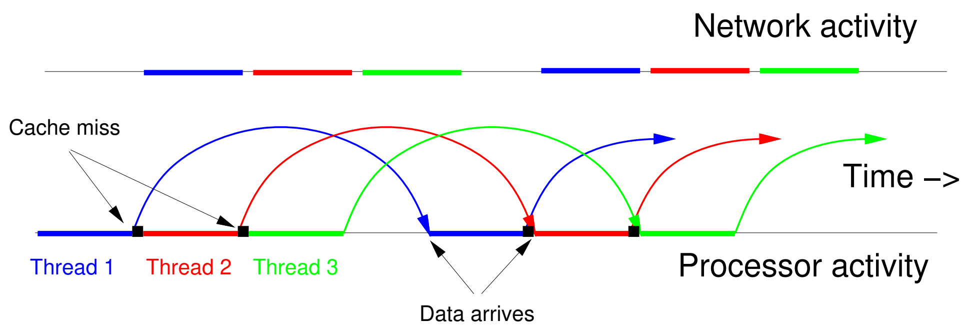


Figure 3: Processor and network activity when multiple threads execute on the processor and fast context switching is used for latency tolerance.

tolerance. (Sparcle and its memory controller provide non-binding prefetch instructions as well.) As illustrated in Figure 3, 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 [16]. 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 utilization, it results in relatively poor single-thread performance and low processor utilization 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. Thus, *block multithreading 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. Sparcle's mechanism to support this is described in Section 4.1.

### 2.3 Efficient Message Interface

Even for implementing a shared-memory model, 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 with solely with shared-memory operations.

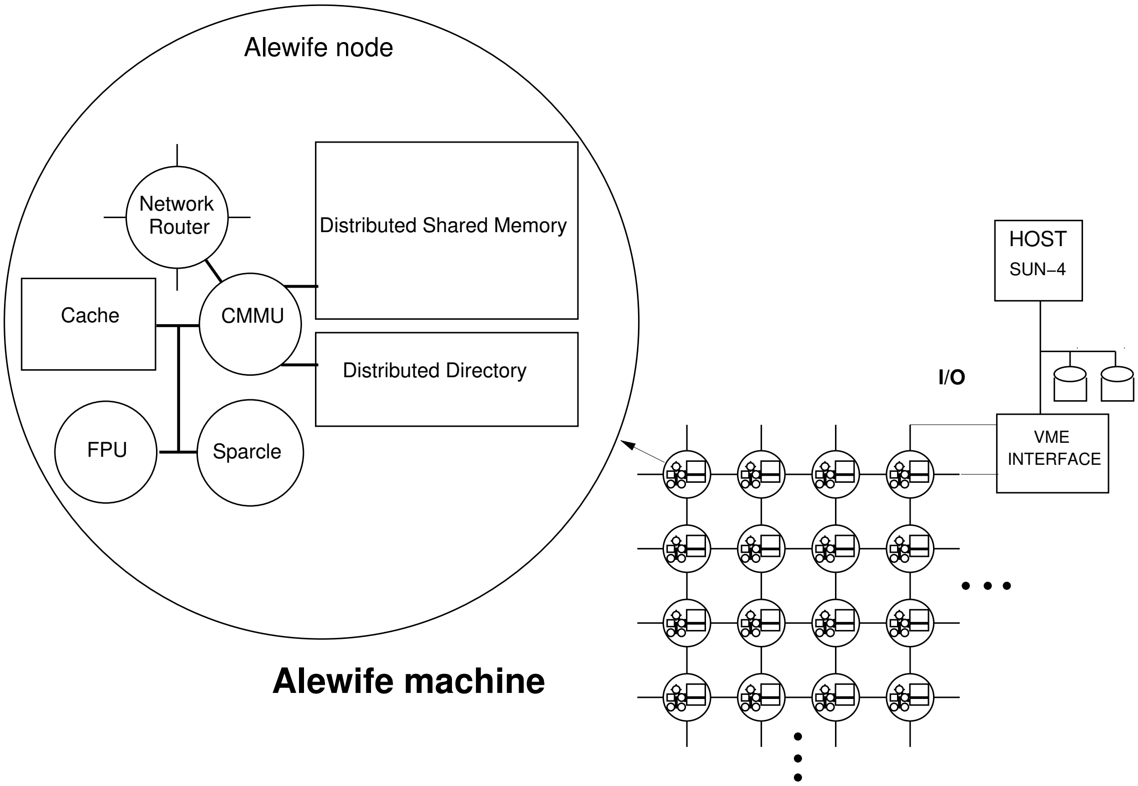


Figure 4: Structure of the Alewife machine.

Examples include remote thread creation and barrier synchronization. In Alewife, with a fast message, a thread can be created on a remote processor in 7$\mu$sec. Restricting ourselves to shared-memory operations, remote thread creation takes 24$\mu$sec. See [10] for a study on the importance of an efficient message interface in a shared-memory setting.

In Sparcle, a fast message send is accomplished 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. The loading and storing is done with two new load and store instructions. Sparcle also supports DMA for larger messages. The mechanics of sending and receiving messages are described in Section 4.

# 3 Overview of the 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 4 depicts the Alewife machine as a set of processing nodes connected in a mesh topology. Each Alewife node consists of a processor, a 64K byte cache, a 4M byte portion of globally-shared distributed memory, a communications and memory-management unit (CMMU), a floating-point coprocessor, and a network switch. An additional 4M bytes of local memory holds the coherence directory and operating system. The network switch chip is an Elko series Mesh Routing Chip (EMRC) from Caltech which has eight-bit channels. The network operates asynchronously with a switching delay of 30 nanoseconds per hop and 60M bytes per second through bidirectional channels.

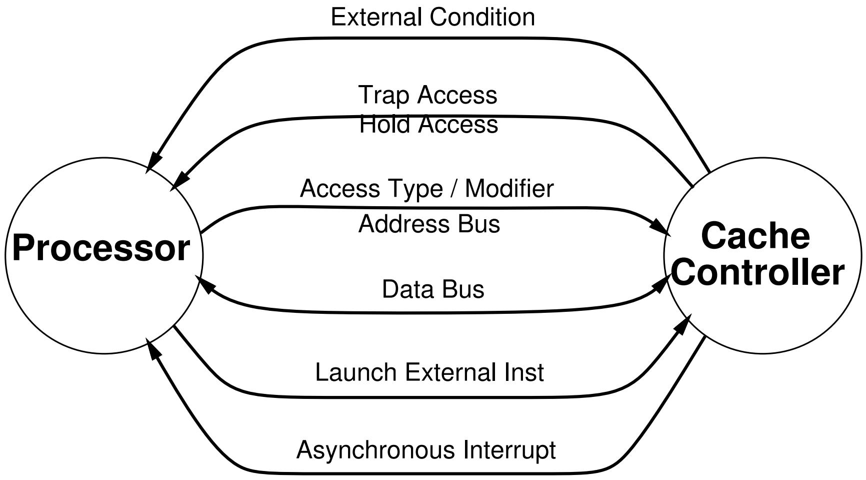


Figure 5: Interface between the processor pipeline and memory controller.

The single-chip CMMU performs a number of tasks, including cache management, DRAM refresh and control, message queuing, remote memory access and DMA. It also supports the LimitLESS cache-coherence protocol [7], 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 of the caches in the system maintain a coherent view of global memory.

Sparcle implements a powerful and flexible interface to the communications and memory management unit. This interface couples the processor pipeline with the CMMU and is depicted in Figure 5. 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*, which is part of the opcode for load and store instructions and which is *not* interpreted by the core processor pipeline. The modifier is used to provide several "flavors" of load and store instructions.
- Two *External Conditions* which return information about the last access. They can be used to affect the flow of control through special branch instructions.
- Several vectored memory exception signals (denoted *Trap Access* in the figure). These synchronous trap lines are used to abort active load and store operations and to invoke function-specific trap handlers.

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

An *instruction extension mechanism* permits the basic instruction set to be augmented with external functional units. Instructions which are added in this way can be pipelined in the same fashion as standard instructions. To make this work, a special range of opcodes is reserved for external instructions. Further, 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 this functionality is already provided the coprocessor interfaces of several microprocessors.

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

# 4 The Architecture and Implementation of Sparcle

Sparcle is best described as a conventional RISC microprocessor with a few additional features to support multiprocessing. These features can be classified into support for latency tolerance, support for fine-grain synchronization, and support for fast message handling. The ensuing sections discuss these features and describe how we implemented them in the SPARC processor, and indicate how they can be implemented in other RISC microprocessors as well.

## 4.1 Mechanisms for Latency Tolerance

Fast context switching on a generic processor is illustrated by Figure 6. This diagram shows four separate register sets with associated program counters and status registers. Each register set represents a context. The active context is pointed to by a hardware register called the *context pointer* or CP. Consequently, a hardware context-switch requires only that the context pointer be altered to point to another context[1] 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 which are maintained by the operating system.

**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, non-overlapping contexts. We use one as a context for trap handlers, as in [8, 15] and the other three for user threads. The SPARC *Current Window Pointer* (CWP) serves as the context pointer. Further, the Window Invalid Mask (WIM) is used to indicate 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 in a nontrivial fashion, 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 which is not in the local cache, the CMMU causes a synchronous memory fault to Sparcle, while simultaneously sending

[1] Depending on details of the implementation, some number of cycles may be needed to flush the pipeline before beginning execution in a new context.

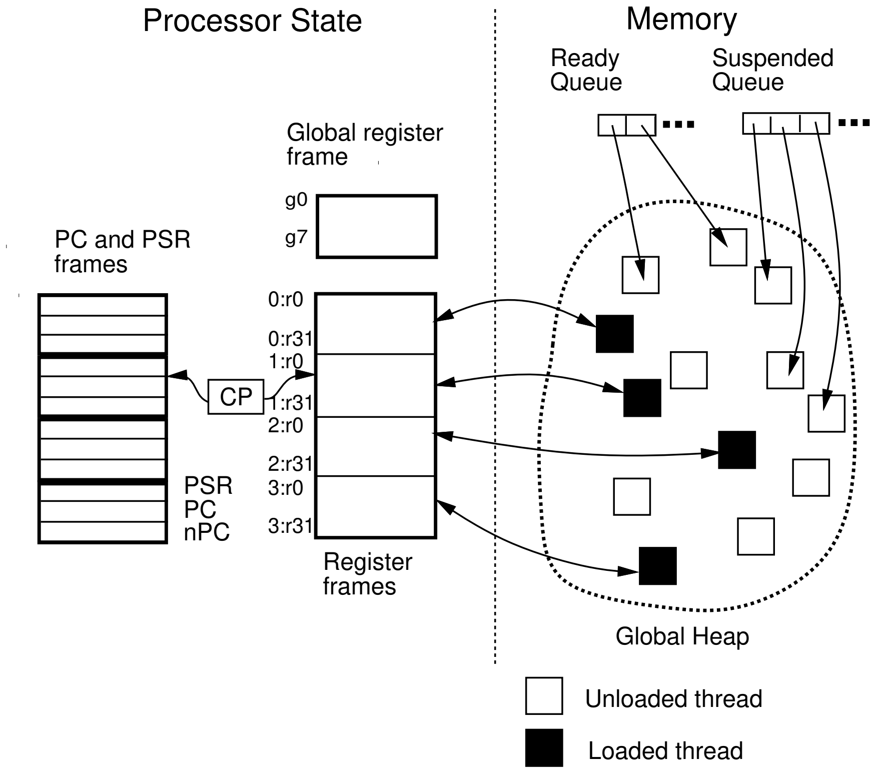


Figure 6: Block Multithreading and Virtual Threads

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, then 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:

- An extra synchronous trap line (with corresponding trap vector) was added so that the processor traps immediately to the context-switch code without having to decode the trap type.
- A new instruction called `nextf` was added. It is much like the SPARC `save` instruction except that the window pointer is advanced to the next active context as indicated by the WIM register. If no additional contexts are active, it leaves the window pointer unchanged.
- The number of instructions for each entry in the SPARC trap vector is increased from 4 to 16. This allows the context switch and other small trap handlers to execute in the trap vector directly.
- The value of the CWP is made 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 floating-point unit.

```
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 7: 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 of 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, decr. CWP)
   7  Fetch:  rdpsr psrreg (save PSR in reserved reg.)
   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 8: Anatomy of a context switch in Sparcle.

With these changes, the context-switch trap handler is shown in Figure 7. When the trap occurs, Sparcle switches *one* window backward (as does a normal SPARC). This switch places the window pointer *between* active contexts, where a few registers are reserved by the Alewife operating-system for context state. As per normal SPARC trapping behavior, the PC and nPC are written to registers r17 and r18 by the hardware. This trap code places the PSR in register r16.

The net effect is that a Sparcle context switch can be done in 14 cycles as depicted in Figure 8. This illustrates the total penalty for a context-switch on a data instruction. Note that, while 15 cycles are shown in this diagram, one of them is the fetch of the first instruction from the next context.

By maintaining a separate PC and PSR 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 [18, 12].

### 4.2 Support for Fine-Grain Synchronization

As described in Section 2.1, 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.

The new load, store and swap instructions are implemented in Sparcle using the SPARC alternate address space instructions. These instructions have been modified in two ways:

1. The load, store and swap alternate space instructions in Sparcle are *unprivileged* for ASI values in the range 0x80 to 0xFF. They remain privileged for ASI values less than 0x80. The CMMU uses the ASI value as an extended opcode, *e.g.*, ASI 0x84 corresponds to the load and trap if empty operation. This allows user code to interact directly with full/empty bits.

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

Full/empty traps are signalled through a new dedicated synchronous trap line. J- and L-structure operations are implemented with the following special load and 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. A synchronization failure can be detected either by trapping, or by an explicit test of this condition code. When a trap occurs, the trap handling software decides what action to take.

**Implementation of J-Structures** To give an example of how the special load and store instructions can be used, we will describe how we implement J-structures and present the cycle counts for various synchronizing operations. Allocating a J-structure is implemented 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 is also straightforward: it is a memory read which traps if the full/empty bit is empty. It is implemented 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 utilization, 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.

Other architectures implement these two state bits directly in hardware by having multiple state bits per memory location [5, 14]. 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, it can use the atomic swap to

```
        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.
        beq,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 9: Machine code implementing a J-structure write. `r1` contains the address of the J-structure location to be written to, and `r2` contains the value to be written. `-1` is the end of queue marker and `0` in an empty location means that the queue is locked.

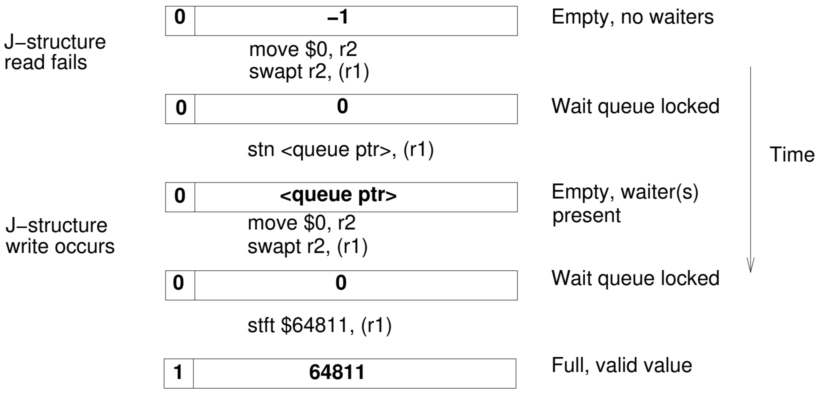


Figure 10: Reading and writing a J-structure slot. `r1` contains a pointer to the J-structure slot. The possible states of a J-structure slot are illustrated here.

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 doing the write.

Using atomic swap and full/empty bits, the machine code in Figure 9 implements a J-structure write. 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 10 gives a scenario of accesses to a J-structure location under this implementation and illustrates the possible states of a J-structure slot.

Table 1 summarizes the instruction and cycle counts of J-structure and L-structure operations for the case where no waiting is needed on reads and no waiters are present on writes. In Sparcle, as in the LSI Logic SPARC, normal reads take two cycles and normal writes 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 by automatic and efficient detection and handling of placeholders via traps. This is achieved with two Sparcle modifications.

| | 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 |

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

First, to detect placeholders, two new instructions called `ntadd` and `ntsub` are added. These instructions cause tag overflow traps whenever the low bit of either of their operands is set[2]. As detailed in Section 2, 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.

### 4.3 Fast Message Handling

Most distributed shared-memory machines are built on top of an underlying message-passing substrate. While traditional shared-memory machines provide a layer of hardware that implements some coherence protocol between the processor and the interconnection network, it is natural 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. This section describes the message mechanism supported by Sparcle.

Sparcle supports the 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*. A message is composed by writing directly to the interconnection network queue using a special store instruction called `stio`. The queues are memory-mapped as an array of network registers in the CMMU, called the *output descriptor array*. In terms of performance, writes 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. A range of opcodes is reserved 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.

[2] `ntadd` and `ntsub` are modifications of the SPARC tagged instructions `taddcctv` and `tsubcctv` which trap whenever the low two bits of either of their operands is set.

```
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 11: Machine code implementing a message send. 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.

After the message is composed, a coprocessor instruction is used to launch the message. Figure 11 illustrates the sending of a single message with one data word and one block of data from memory. If Sparcle is in USER mode and the header is privileged, an exception will occur. The CMMU maintains the atomicity of messages as described in Section 5.

**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` 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 and 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` (for *store IO*) and `ldio` (for *load IO*). 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 which 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.

- Use of 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 which 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 of for launch instructions, ensuring that message launches are atomic.

- Fast interrupts allow rapid entry into message handler code on the arrival of a message. In our current implementation, because interrupts always force the processor into supervisor

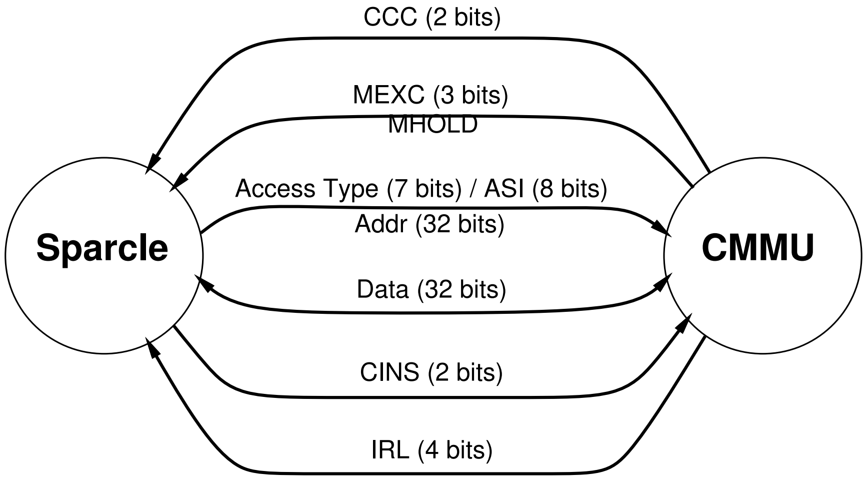


Figure 12: Actual Sparcle signal names.

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 trap.

# 5 The CMMU Interface

As alluded to in earlier sections, the Sparcle processor is part of a complete system. Consequently, several of the mechanisms which were included in Sparcle are incomplete without the support of the communications and memory-management unit (CMMU). This section will briefly discuss the Alewife CMMU and how it interfaces to Sparcle. Although the Alewife CMMU provides a number of features, the following discussion will focus on the cache controller and message interface.

Section 3 discussed two categories of signals in the interface between processor and CMMU: flexible data access mechanisms and flexible instruction extension mechanisms. Figure 12 makes this interface more concrete by showing Sparcle equivalent names for all of the signals. Each of the signals in this figure corresponds directly to signals in Figure 5.

A few of the data access mechanisms require further discussion. The *Modifier* is implemented with the SPARC *ASI* field. As mentioned in earlier sections, Sparcle contains a number of new load and store instructions which differ only by the values which they place on the ASI pins during data cycles. These new load and 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 are used to invoke context-switch and synchronization traps. Third, the *External Condition* bits are implemented through Sparcle coprocessor condition codes (*CCC*). These signals were a part of unmodified SPARC processor; consequently, SPARC branch on condition-code instructions can be used to examine them.

Finally, the external instruction interface is implemented directly through the SPARC coprocessor interface. SPARC 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.

### 5.1 Latency Tolerance

Rapid context-switching for latency tolerance was discussed in earlier sections from the standpoint of the Sparcle processor. In addition to the Sparcle mechanisms discussed earlier, 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 performing some other task. Consequently, when the processor requests a data item which 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 which contains the requested data. Note that the mechanisms required to handle context-switching differ little from those required for software prefetching[3].

### 5.2 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 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. First, it eliminates a need for an odd number of bits in the physical memory used for cache data. Second, access to the tags file is much faster than access to the cache data, both because the tags file is smaller and because no chip-crossings are required. This is advantageous in that it 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. Each of the new synchronizing load and store instructions mentioned in Section 4.2 are distinguished by the value of the ASI field which they generate (and whether they are read or write operations). For each data access, the Alewife CMMU takes the proferred ASI value along with the address and type of access. The address is used to index into the tags file, retrieving both the tag and the appropriate full/empty bit. Simultaneously, the ASI value is decoded to produce two different actions, one which will be taken if the full/empty bit is full and one which will be taken if the full/empty bit is empty. When the tag lookup completes, both tags match and full/empty bit operations are completed simultaneously, either flagging a context-switch (on cache miss), a synchronization fault, or successful completion of the access. In all cases, the full/empty bit which was first retrieved from the tags file is placed in one of the external condition codes for future examination by the processor.

The support which Alewife provides for full/empty bit synchronization is external to the processor pipeline: i.e. it occurs at the first-level cache. Consequently, full/empty bits never enter the processor core. Further, individual load and 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

[3]Some interesting forward-progress issues appear, however. See [11].

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| `stio` | `Header, N0` | I | D | E | M | M | W | | | | | | |
| `stio` | `Data, N1` | | I | — | — | D | E | M | M | W | | | |
| `ipilaunch` | `1, 1` | | | I | — | — | — | — | D | E | W | Q1 | Q2 |
| | | | | | | | I | — | — | D | E | W | |
| | | | | | | | | | | I | D | E | W |
| | Cache Bus: | I | I | I | N0 | N0 | I | N1 | N1 | I | ··· | | |

Figure 13: Pipelining for transmission of a message with a single data word. Sparcle pipeline stages are **I**nstruction fetch, **D**ecode, **E**xecute, **M**emory, and **W**riteback. 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**.

which 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 which are sandwiched between Alpha-style [4] load-locked/store-conditional synchronization instructions.

### 5.3 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 mentioned earlier, one special load and store instruction (and corresponding ASI) are reserved for rapid descriptions of outgoing messages and rapid examination of incoming messages. The cache controller recognizes accesses with this ASI and causes data transfers to occur to and from message queues instead of the cache. This permits message data to be transferred between the processor and network at the same speed as cached accesses.

Alewife employs the external instruction interface to implement the message launch mechanism. Consequently, message launches can be pipelined. A simple pipeline example is given in Figure 13. Here, the two-cycle latency for stores and the lack of an instruction cache limits the message throughput. More aggressive processor implementations would not suffer from this limitation. Note that the use of DMA on message output adds additional cycles (not shown here) to the network pipeline.

The close coupling between the message launch mechanism and the processor pipeline allows a precise launch completion point to be identified (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, a complete output packet is committed to the network. These atomic semantics allow a single network output port to be shared by multiple levels of user and interrupt code without requiring that the user disable interrupts before beginning to describe a message.

Figure 14: A photograph of Sparcle's test system.

## 6 Summary and Status

The Sparcle chip incorporates mechanisms required for massively parallel systems in a SPARC RISC core. Coupled with a communications and memory management unit, Sparcle allows a fast, 14-cycle context switch, an 8-cycle user-level message send, and fine-grain full/empty bit synchronization. The Sparcle chip was developed jointly by MIT, LSI Logic and SUN.

We received working Sparcle chips from LSI in March 1992. A single-node test system was operational by the time Sparcle's arrived. A compiler and a runtime system for our parallel versions of C and lisp was also operational for several months. The test system shown in Figure 14 comprises 256K bytes of SRAM memory, an I/O interface to the VME bus for downloading programs and monitoring execution, and control logic to exercise the full/empty bit and context switching functionality. The test system had been debugged using SPARCs[4] in place of Sparcles and could operate at a maximum clock frequency of about 25 MHz.

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 suggests we will be able to run the chips in an Alewife node board at roughly the same speed as the original, unmodified SPARCs.

One of unique aspects of the development of Sparcle was that its implementation relied on modifying an existing design through a 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 for us. This implementation strategy not only allowed us, at a university, to experiment with architectural ideas in a real, contemporary processor design, but it significantly reduced the design effort from the concept stage to working chip.

Figure 15 depicts the resulting project schedule for Sparcle. Sparcle's early architecture was

[4]SPARC and Sparcle have only a few differing pins, and in fact, Sparcle provides an input signal pin $M$, which allows switching between SPARC and Sparcle modes.

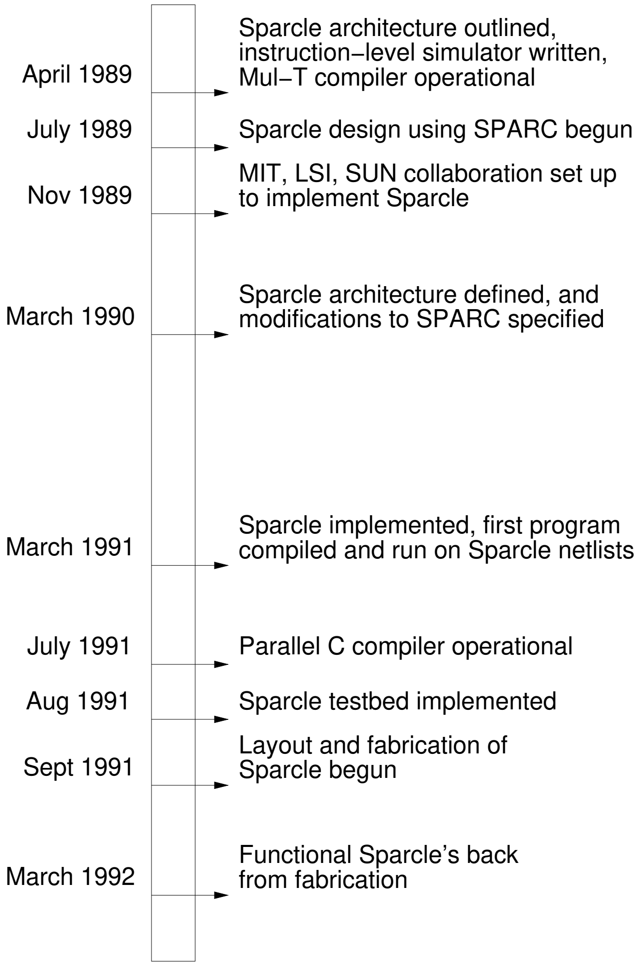


Figure 15: Sparcle's implementation schedule.

defined in April 1989. A Sparcle compiler for a version of Lisp was written, and a cycle-by-cycle simulator was also implemented at MIT. Later, a compiler for a parallel version of C was also developed. A detailed specification of the modifications to SPARC required to implement Sparcle was developed by March 1990. Then, high-level changes to SPARC functional blocks were made at SUN and lower gate-level changes level were made at LSI. These changes were tested against Sparcle binaries produced at MIT. Then net lists were synthesized at LSI and tested against several hundred thousands of test vectors at MIT. The test vectors included both SPARC vectors provided by LSI and Sparcle vectors obtained from the MIT Sparcle simulator. The test set up included a net list module for the floating-point coprocessor, and a behavioral model for the rest of the memory and communication systems. Finally, layout and fabrication was undertaken by LSI. During this time, a test system for Sparcle was also implemented.

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 16 shows an Alewife node board with the Sparcle and floating point unit. Figure 17 shows a 16-node Alewife system package developed by the APT group at ISI in Los Angeles. The CMMU chip has been implemented and tested. It is being

Figure 16: A photograph of an Alewife node.

implemented in LSI Logic's LEA300K process, and we expect to begin its fabrication shortly.

# 7 Acknowledgements

The Sparcle project is funded in part by DARPA contract # N00014-87-K-0825 and in part by NSF grant # MIP-9012773. LSI Logic and SUN Microsystems helped implement Sparcle, and LSI Logic supported the fabrication of Sparcle. We would like to acknowledge the contributions of Dan Nussbaum, who was partly responsible for the processor simulator and run-time system and was the source of several ideas. Our design was influenced by Halstead's work on multithreaded processors. 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] Sarita V. Adve and Mark D. Hill. Weak Ordering - A New Definition. In *Proceedings 17th Annual International Symposium on Computer Architecture*, New York, June 1990.

[2] Anant Agarwal, David Chaiken, Godfrey D'Souza, Kirk Johnson, David Kranz, John Kubiatowicz, Kiyoshi Kurihara, Beng-Hong Lim, Gino Maa, Dan Nussbaum, Mike Parkin, and Donald Yeung. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In *Proceedings of Workshop on Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1991. An extended version of this paper has been submitted for publication, and appears as MIT/LCS Memo TM-454, 1991.

[3] Anant Agarwal, Beng-Hong Lim, David A. Kranz, and John Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *Proceedings 17th Annual International Symposium on Computer Architecture*, pages 104–114, New York, June 1990.

Figure 17: A photograph of the 16-node Alewife package.

[4] *Alpha Architecture Reference Manual.* Digital Press, 1992.

[5] Gail Alverson, Robert Alverson, and David Callahan. Exploiting Heterogeneous Parallelism on a Multithreaded Multiprocessor. In *Workshop on Multithreaded Computers, Proceedings of Supercomputing '91.* ACM Sigraph & IEEE, November 1991.

[6] Arvind, R. S. Nikhil, and K. K. Pingali. I-Structures: Data Structures for Parallel Computing. In *Proceedings of the Workshop on Graph Reduction, (Springer-Verlag Lecture Notes in Computer Science 279)*, September/October 1986.

[7] David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 224–234. ACM, April 1991.

[8] William J. Dally et al. The J-Machine: A Fine-Grain Concurrent Computer. In *IFIP Congress*, 1989.

[9] Michel Dubois, Christoph Scheurich, and Faye A. Briggs. Synchronization, coherence, and event ordering in multiprocessors. *IEEE Computer*, pages 9–21, February 1988.

[10] David Kranz, Kirk Johnson, Anant Agarwal, John Kubiatowicz, and Beng-Hong Lim. Integrating Message-Passing and Shared-Memory; Early Experience. In *Conference on Principles and Practice of Parallel Programming.* ACM, May 1993.

[11] John Kubiatowicz, David Chaiken, and Anant Agarwal. Closing the Window of Vulnerability in Multiphase Memory Transactions. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V).* ACM, October 1992.

[12] Kiyoshi Kurihara, David Chaiken, and Anant Agarwal. Latency Tolerance through Multithreading in Large-Scale Multiprocessors. In *Proceedings International Symposium on Shared Memory Multiprocessing*, April 1991.

[13] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings 17th Annual International Symposium on Computer Architecture*, pages 148–159, New York, June 1990.

[14] G. M. Papadopoulos and D.E. Culler. Monsoon: An Explicit Token-Store Architecture. In *Proceedings 17th Annual International Symposium on Computer Architecture*, New York, June 1990. IEEE.

[15] Charles L. Seitz. Mosaic Project Update, March 1991. Darpa VLSI woskshop.

[16] B.J. Smith. Architecture and Applications of the HEP Multiprocessor Computer System. *SPIE*, 298:241–248, 1981.

[17] SPARC Architecture Manual, 1988. SUN Microsystems, Mountain View, California.

[18] Wolf-Dietrich Weber and Anoop Gupta. Exploring the Benefits of Multiple Hardware Contexts in a Multiprocessor Architecture: Preliminary Results. In *Proceedings 16th Annual International Symposium on Computer Architecture*, pages 273–280, New York, June 1989.