# Firefly: A Multiprocessor Workstation

CHARLES P. THACKER, LAWRENCE C. STEWART, MEMBER, IEEE, AND
EDWIN H. SATTERTHWAITE, JR., MEMBER, IEEE

*Abstract*—**Firefly is a shared memory multiprocessor workstation developed at the Digital Equipment Corporation Systems Research Center (SRC). A Firefly system consists of from one to nine VLSI VAX processors, each with a floating point accelerator and a cache. The caches are coherent, so that all processors see a consistent view of main memory. The Firefly runs a software system that emulates the Ultrix system call interface, and in addition provides support for multiprocessing through multiple threads of control in a single address space. Communication is provided uniformly through the use of remote procedure call. We describe the goals, hardware, software system, and performance of the Firefly, and discuss the extent to which SRC has been successful in providing software to take advantage of multiprocessing.**

*Index Terms*—**Cache design, computer architecture, distributed systems, Modula-2, operating systems, performance analysis, shared memory multiprocessors, workstations.**

## I. Introduction

THE design and construction of the Firefly was the first hardware development project at SRC. It addressed the need for a personal workstation that would provide computing for SRC's research in programming technology and multiprocessing, within the context of distributed personal computing. To fill this need, the system had to satisfy three main requirements: it had to be a powerful workstation, it had to provide a useful degree of multiprocessing, and it had to be completed in a short time. These requirements led to a number of design choices and compromises, particularly in cache structure, input–output architecture, and memory capacity. Since the Firefly was to be a workstation, it had to operate in the user's office. This meant that it could not be too large, or too loud, or draw too much power. The limitation on physical size, combined with the short schedule, meant that as few module types as possible should be designed. We did not have the facilities to design high-density custom or semicustom chips, and this fact, coupled with the real-estate restriction, suggested the use of a single-chip CPU attached to a small, direct-mapped cache, a simple memory bus protocol, and a relatively low-performance bus. The MicroVAX 78032 [4] and its companion floating point accelerator, which provide the full VAX architecture in a two-chip implementation, filled the requirement for the CPU.

A small, direct-mapped cache is not usually consistent with high performance. The Firefly is unusual in that the primary purpose of its cache is not to reduce the average access time to main memory. Instead, its purpose is to reduce the per-processor load on the main storage system. This reduction in bus loading makes it possible for a relatively low-performance bus to support a number of processors. Another unusual aspect of the Firefly is that it provides a global shared memory in which data written by one processor are immediately available to the other processors in the system. The need for *coherent* main memory made it impossible to use a standard memory bus or storage modules, but this factor was outweighed by the simplification of the system software that coherent memory provides.

The fact that we wanted the Firefly finished quickly dictated that we use an existing input–output system and as many standard parts and subassemblies as we could. Fortunately, we were able to borrow the packaging and most of the input–output system from the DEC MicroVAX II computer system, which was introduced in the spring of 1985. The only input–output device that has been designed specifically for the Firefly is the display controller, since a standard controller with the requisite properties did not exist at the time.

The software system for the Firefly, called Topaz, was designed and built in parallel with the hardware. To avoid the need to provide a large amount of applications software, Topaz provides an emulation of the Ultrix system call interface, which allows it to run VAX Ultrix binaries unchanged [11]. Since Ultrix does not include provisions for multiprocessing, these facilities are provided by the Topaz *Threads* module [3]. The interface provided by the threads module allows the creation and management of a number of concurrent threads of control within a single address space. Remote procedure call (RPC) is used extensively in the Firefly, for communication both between address spaces within a single machine and between multiple machines on a network.

Most Firefly software, including Topaz, is written in Modula-2+ [12], an extension of the Modula-2 [16] language that includes support for garbage-collected storage, exception handling, and concurrency. Modula-2 provides strong type checking and the concept of separation of interface and implementation. These features make it much easier for a number of programmers to work cooperatively on large and complex software projects.

In Section II, we describe the hardware of the Firefly, with emphasis on the technique used to provide global shared memory. We also discuss the performance estimates that were made as part of the design process. In Section III, we discuss the software system, and in Section IV, some performance

Manuscript received November 2, 1987; revised February 25, 1988.
The authors are with Digital Equipment Corporation Systems Research Center, Palo Alto, CA 94301.
IEEE Log Number 8821807.

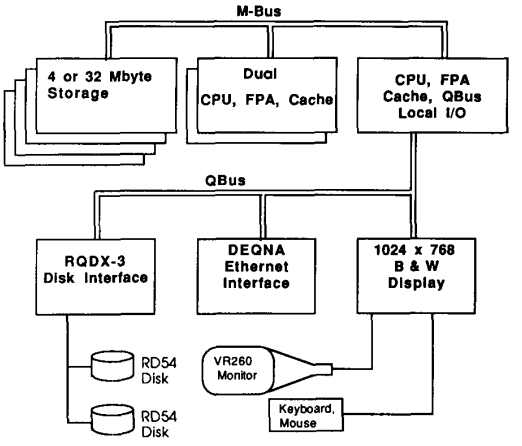



Fig. 1. Firefly system. A system consists of five CPU's, each with an attached snoopy cache. Caches communicate with storage over a dedicated bus, while input-output is done over a standard QBus.

results are presented. In Section V, we discuss the extent to which the Firefly has met its goals.

## II. Hardware Description

The principal components of the Firefly are shown in Fig. 1. A typical five-processor system consists of the ten printed circuit boards shown in the figure. The modules are approximately 8 by 10 inches in size (the input-output controllers are half this size), and are housed in a standard DEC deskside cabinet that provides power and space for the electronics and for a number of disk or tape drives.

The system makes use of a standard DEC Q22 bus for input-output. The use of a standard bus makes it easy to add additional or specialized input-output controllers. The QBus does asynchronous transfers and has a bandwidth of approximately 3 Mbytes/s. When fully loaded, the QBus could consume about 30 percent of the main memory bandwidth, but the average load is much lower. A number of standard input-output controllers exist for the QBus. The Firefly uses a DEQNA Ethernet controller and an RQDX3 disk controller. Both controllers are direct-memory-access (DMA) devices, and do data transfers directly to Firefly memory. The standard system uses two RD54 disk drives, which provide a total of 280 Mbytes of storage.

The nonstandard parts of the Firefly are the CPU modules, the storage modules, and the display controller. The CPU modules communicate among themselves and with storage over a synchronous bus, the MBus. The MBus is quite simple, consisting of five request lines, 32 lines that carry data or addresses, three control lines, and a 10 MHz clock signal that synchronizes the processors and storage modules. The MBus can do one 4-byte transfer every 400 ns for an aggregate bandwidth of 10 Mbytes per second. In addition to providing access to memory, it also provides an interprocessor interrupt facility, and the signals used to reset and initialize the system. It is carried between modules on flat-ribbon cable, so that a standard backplane can be used. The MBus is driven and received at each module by TTL components. Since it is not possible to provide proper electrical termination for TTL signals because of the limited current available from drivers, the MBus must be short, and it can support only a few attached modules. These electrical constraints limit the Firefly to about nine processors. This is acceptable because the bandwidth of the bus also introduces a similar limit, so the system is fairly well balanced.

Firefly processors are packaged on two different types of modules. The first type is the input-output processor (IOP), of which there is one per system. This module contains a single MicroVAX 78032 CPU, a floating point accelerator (FPA), and a 16 kbyte cache. These components occupy about half the available module area. The rest of this module is occupied by logic that operates the QBus, a time-of-year clock, the bootstrap read-only memory, and a console terminal interface (used primarily for low-level debugging). Since DMA devices on the QBus can only address a memory of 4 Mbytes, a translation table is necessary to provide the extra bits needed to address the main memory of the Firefly. This table is managed by the IOP. QBus DMA transfers are made through the IOP's cache (although QBus references that miss in the cache bypass it, rather that allocating space that would not be referenced again), so DMA devices see the same coherent view of memory that the CPU does. The IOP receives all interrupts from devices on the QBus, and it is the only processor in the system that can do programmed input-output transfers to QBus controllers. This means that the portions of the operating system that deal directly with input-output devices *must* run on the IOP, but the IOP can also run ordinary processes.

The second module type contains two CPU's and FPA's, and two caches. The standard five-processor system contains two of these modules. Two versions of this module have been built. The first version uses the MicroVAX 78032 CPU and has 16 kbyte caches. This module is essentially two copies of the CPU, FPA, and cache contained on the IOP module. The second version uses the newer CVAX 78034 [2] processor and its companion FPA and has 64 kbyte caches. The two cache designs are similar, except that the CVAX module takes advantage of the fourfold increase in static RAM density that took place between the times the two designs were done. The caches are direct mapped (i.e., a particular memory location must occupy a specific location in the cache if it is present in the cache at all). They are addressed with real addresses, since both the CVAX and the MicroVAX contain on-chip virtual address translation hardware. Each cache in the MicroVAX module consists of a 4096 entry *data store* and a *tag store* with the same number of entries. The tag store contains the bits of an address not used to index the cache, as well as 2 state bits used by the coherence protocol. The CVAX module's cache is four times as large, and the CVAX CPU also contains a 1024 byte on-chip cache. Due to the difficulty of maintaining memory coherence in a two-level cache, the on-chip cache is only used to store instructions, which are essentially read-only. The external cache is fast enough that a CPU reference that hits in it does not incur any wait states (i.e., a faster cache would not improve the performance). In the MicroVAX version, the access time is 400 ns, while it is twice as fast in the CVAX version. In addition to being larger, the RAM's

used to implement the CVAX cache are considerably faster than their older counterparts.

The storage modules have also benefited from rapid improvements in memory density. In the original version of the system, the maximum amount of memory that could be attached was 16 Mbytes, contained on four modules of 4 Mbytes each using 256 kbit RAM chips. This was a serious mistake. Minicomputer physical memories have been doubling in size approximately every two years, with 16–18 bits of address being common in 1970 [14, p. 7]. By this measure, the 1986 Firefly should have had 64 Mbytes of physical memory. The severity of this problem was realized shortly after the Firefly was placed into service, and a new module was designed that uses 1 Mbit chips to provide 32 Mbytes of storage per module, or 128 Mbytes total. The CVAX processors can address all of this memory, but we have no current plans to upgrade the IOP modules. Thus, the IOP's CPU and the DMA devices can access only the first 16 Mbytes of physical memory. This restriction has made it necessary for the operating system to copy data when an input–output device needs to transfer to or from a location above the 16 Mbyte limit, but this performance degradation is more than offset by the benefits of the additional memory.

The monochrome display controller (MDC) is packaged on a 5 by 8 inch module. It contains an 8 MHz 29116 16-bit microprocessor with 2048 40-bit words of microinstruction memory and a one-megapixel frame buffer constructed with video RAM's. Three-quarters of the frame buffer holds the display bit map, while the rest is available to the display management software. The MDC periodically polls a work queue kept in Firefly main memory, and executes commands from the queue. Commands are provided to do BitBlt [6] operations within the internal frame buffer or between main memory and the buffer. An optimized version of BitBlt is provided to paint characters from a font in off-screen memory. The MDC can paint a large area of the screen at 16 megapixels per second, and can paint approximately 20 000 ten-point characters per second. The MDC also supports a keyboard and mouse. Sixty times per second, the controller deposits the current mouse position and an unencoded bit map representing the current state of the keyboard in Firefly memory.

The MDC provides an example of implementing a well-understood abstraction in specialized hardware to increase its performance. We have had a decade of experience with the use of BitBlt as a display primitive, and therefore felt comfortable in providing a rigid interface. Since they are less generally useful, the MDC provides no facilities for more complex drawing primitives such as splines or conics. Designing the Firefly display controller as an input–output device has brought an unexpected result. It is easy to plug multiple display controllers into a single Firefly, and the marginal cost is dominated by the cost of the extra monitor. Many SRC researchers now have multiple displays and find the increase in display area valuable.

The task of designing the Firefly hardware was straightforward, but we did make several serious mistakes during the project. The primary mistake was to depend on careful design and cross-checking, rather than simulation, to detect design errors. When building hardware with off-the-shelf components, it is easy to argue, but rarely true, that simulation is too time-consuming. Although we used a strict design methodology, all the modules in the MicroVAX Firefly underwent at least one revision, and we estimate that we would have saved from four to six months had we simulated the design carefully before building it. The CVAX version of the module was simulated, but timing analysis was again done by hand and several minor errors were not detected until the module was built. The second mistake was to depend on vendor specifications to describe accurately the properties of components. In several cases, standard components simply did not work according to the vendor's specifications. We spent a great deal of time hunting down obscure bugs that were caused by bad design on the part of the component vendors, rather than ourselves. Simulation would not have helped in this area, except to bolster our confidence in the design.

### *A. Memory System Details*

The most important feature of the Firefly caches is that they provide a global shared memory in which data written by one processor are immediately available to other processors. This is accomplished using coherent, or *snoopy,* caches, which monitor the memory bus traffic and take or supply data as necessary to maintain coherence.

The usual reason for including a cache in a computer system is to reduce the average memory access time seen by CPU references. The cache is usually much smaller than main memory, but is built with components that are significantly faster. The cache is able to supply a large fraction of the processor requests because of two properties of programs; spatial and temporal locality. Spatial locality is the tendency for a program to reference clustered locations in preference to locations distributed randomly, while temporal locality is the tendency for a program to reference the same location several times during brief portions of its execution. The Firefly cache takes advantage of the temporal locality of programs, but since its line size is the same as the CPU's unit of transfer (4 bytes), it cannot take advantage of spatial locality. This is not as harmful as it might seem, since cache misses that acquire the bus immediately add only one cycle to a MicroVAX CPU access. The miss penalty is relatively higher in the CVAX version of the system. MBus cycles take the same time to complete, but the processor cycles are twice as fast. Cache misses add four CVAX cycles to the access time.

The purpose of the cache in the Firefly is *not* the usual one of reducing access time. Instead, the cache is used to shield the memory bus from the majority of references made by the CPU, so that a main memory with modest performance can service a large number of processors.

A number of snoopy cache designs have appeared in the literature. A report by Archibald and Baer is a survey of several protocols [1]. The simplest protocol is *write-though* with invalidation, in which all writes done by a CPU are sent to the main memory bus. Whenever a cache observes a write directed to a location it contains, it invalidates its copy of the location. This is not a practical protocol for more than a few processors, because the substantial write traffic will rapidly

saturate the bus, and extra misses will be required to reload invalidated lines.

Rather than using write-through, most published protocols make use of *write-back,* in which the contents of a cache line are written to main memory only when the line is needed for another CPU reference and its contents have been modified locally. A modified line is *dirty,* a line selected for replacement is a *victim,* and the write-back operation is a *victim write.* Write-back is much more efficient than write-through, since only dirty victims are written back to memory. Unfortunately, write-back exacerbates the problem of providing a coherent memory, because CPU writes are not directly visible to other caches. This problem is frequently solved by having caches that wish to write a particular location acquire permission to do so, or *ownership* of the location. Ownership implies permission to write the location, as well as the responsibility for updating main memory when the cache line containing the location is victimized. A read-only location can be in several caches simultaneously, but when a cache acquires ownership of a line in order to write it, other caches invalidate their contents. Berkeley Ownership [8] is an example of an ownership protocol.

The coherence protocol used in the Firefly differs from most others in that multiple caches are allowed to contain a writeable datum simultaneously, and no prearrangement is required for a processor to write the shared location. The Xerox Dragon [10] uses a similar scheme. The key idea in this protocol is that a cache can detect when another cache shares a particular location. For nonshared lines, a write-back strategy it used. Processor reads and writes of nonshared locations go only to the cache, and writes to main storage are needed only when a dirty line is victimized to satisfy a miss. For locations that are shared, processor reads are serviced from the cache, but when a processor write is done, the cache does write-through, and other caches that share the datum are updated, as is main storage.

To implement the protocol, each cache maintains 2 state bits for each cache line, dirty and shared. The dirty bit indicates that the cached copy of a datum has been modified with respect to main memory and must therefore by written back when the line is victimized. The shared bit indicates that some other cache *may* also contain the line, and that write-through must be done when the attached CPU does a write. The MBus contains a signal, MShared, that is asserted during an MBus operation if another cache contains the requested location.

The four possible states of a cache line and the transitions between states caused by processor ($P$) and memory bus ($M$) operations are shown in Fig. 2. The initial reference that brings a location into the cache leaves the line clean and either shared or not shared, depending on the response that the initiator receives from other caches on the MShared line (responses are shown in parentheses in the figure). Subsequent transitions between states occur when a processor write to a shared line results in a write-through (the state transition depends on the response), or when an MRead or MWrite done by another cache hits in this one. In the latter case, the cache updates its data if the operation is MWrite, or supplies its data if the operation is MRead.

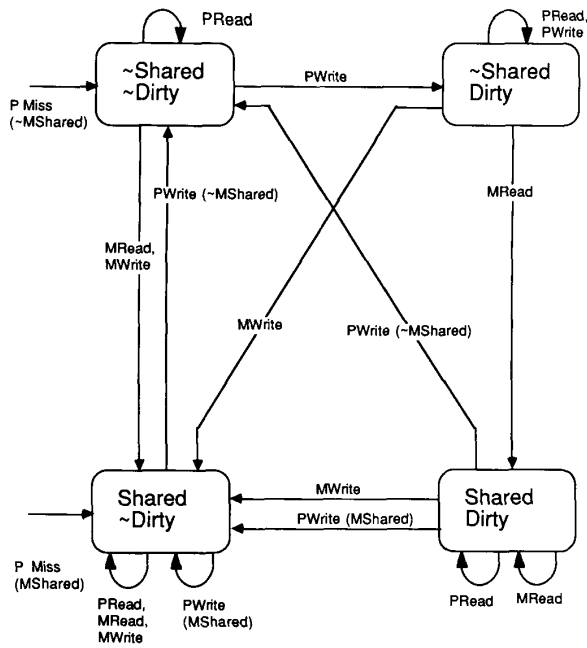



Fig. 2. Cache line states. Each cache line may be in one of four states. State transitions occur when a CPU write is directed to a shared line, causing write-through. In this case, the transition depends on whether or not another cache shares the line. Transitions also occur when an MBus operation done by another cache hits in this one.

Cache lines do not require an invalid state. When each processor in the system is reset, its cache is placed in *miss mode,* in which all CPU reads are forced to miss, and all victims are made to appear clean so that they will not be written. The bootstrap program does a sequence of reads that touch every location in the cache, making the contents consistent with memory. The program then disables miss mode, and normal operation begins. During normal operation, a location can be displaced from a cache if the line is needed to satisfy a miss, but the protocol never invalidates lines, so no invalid state is needed.

A CPU read can cause zero, one, or two MBus operations. If the read hits in the cache, no bus traffic is needed. If the read misses and the line is clean, an MBus read is issued to load the line with the requested data. If the line is dirty, its contents must be written back to main memory before the read is done. When the read is done, the line's shared bit is set to the value of MShared returned by other caches.

A CPU write that hits in a nonshared line requires no MBus traffic. The line is marked dirty so that it will be written back to memory when it becomes a victim. If the line is shared, the cache does write-through to maintain coherence and update main memory. In this case, the line is marked clean. A VAX CPU can issue writes to bytes, to 16-bit words, or to 32-bit long words. When a byte or word write misses, the cache first reads the long word containing the byte or word, then it modifies only that portion of the line that was supplied by the CPU. If the line is shared, the modification is not done until the cache acquires the bus to do write-through, so writes to


Fig. 3. MBus timing.

two different bytes of the same long word done by different CPU's work properly. Since partial long word writes are treated as read misses followed by write hits, the bus, most of the cache logic, and the storage modules have straightforward implementations, since they never deal with partial-long word quantities.

Most writes done by VAX programs are to aligned long words. Since Firefly cache lines are only one long word in size, long word write misses are optimized. Instead of doing a read, then overwriting the line with write data, the cache simply does write-through, leaving the line clean. The final state of the shared bit is determined by the value on the MShared line received from other caches.

The Firefly coherence protocol has the advantage that write-through is done only when it is logically necessary to support sharing. When a location ceases to be shared, only one extra write-through is done by the last cache that contains the location. This write does not receive MShared from another cache, so the shared bit is cleared and the cache reverts to doing write-back. The disadvantage of this *conditional write-through* strategy is that write-through continues as long as a datum resides in more than one cache, even though only one processor may be using it. If processes are allowed to move freely between processors, the number of unnecessary writes could be significant, since most of the writeable data for a process will be in both the old and the new cache until the data are displaced by the misses of other processes. For this reason, the operating system scheduler goes to some effort to avoid process migration. A coherence protocol that invalidates the contents of other caches when shared locations are written avoids this problem, but performs poorly when actual sharing occurs, since the invalidated information must be reloaded when it is next needed.

The timing of MBus operations is shown in Fig. 3. There are only two operations, MRead and MWrite. Each operation requires four 100 ns bus cycles. During the first cycle, caches arbitrate for the bus, using a set of wires provided for the purpose. The highest priority requester places the address and a bit specifying the operation onto the bus during the latter part of the first cycle. If the operation is MWrite, the initiator sends write data during the second cycle. All caches other than the initiator determine whether they contain the requested address during the second cycle, and if one or more of them contain the address, they assert the MShared signal during the third cycle. During the fourth cycle, read data are supplied. If no cache asserted MShared during the third cycle, the data come from the memory. If MShared was asserted, the caches that contain the line supply the data, and the memory is inhibited. More than one cache may supply data, but since the protocol ensures coherence, the values will be identical. Since the bus timing is fixed, MBus operations have priority for access to the cache tag stores. When a conflict occurs, the CPU operation is delayed for one cycle. Such conflicts are infrequent, since an average instruction requires about ten cycles to execute, but requires only two tag store accesses, each of which requires one cycle.

The VAX architecture provides a number of *interlocked* instructions that support atomic access to shared variables. The CPU implements these operations with two primitive operations: a read-lock that acquires a global lock before doing the read, and a write-unlock that releases the lock when the write is finished. In the original Firefly, the global lock is implemented by a daisy-chained arbiter carried between the CPU modules on the backplane. In the CVAX version, lock requests are carried in parallel on the Mbus and resolved combinationally. When the CPU issues a read-lock, the read is not processed by the cache until the CPU has requested and acquired the lock. A write-unlock is treated as a normal write, except that the global lock is released when the write is finished. The global lock itself is never held for more than a few cycles. Thus, it is almost always available and does not represent a significant bottleneck. Time to acquire the lock is important for such applications as distributed storage management; the change to the CVAX scheme increased the speed of a reference-counted assignment, a central operation in our garbage-collection algorithm, by a factor of nearly four.

Although the description would seem to imply a complex implementation, surprisingly little logic is needed. The cache data paths in the original dual-processor module consist of 11 4K $\times$ 4 static RAM's and about 20 other TTL components. The control logic is implemented in about 15 parts, mostly PAL's. The complexity of the CVAX version of this module is similar, except that 16K $\times$ 4 static RAM's are used in the caches, and the address data paths are several bits wider.

### *B. Hardware Performance Estimate*

In a multiprocessor in which a number of caches communicate with memory over a shared bus, the system performance is ultimately limited by the speed of that bus. As more processors are added, their activity introduces delays which slow all processors. Eventually, the bus saturates. Several researchers have used trace-driven simulation to analyze the effects of cache organization and choice of bus protocol on system performance [15].

We did a similar analysis for the Firefly, although the reason was not to predict the outcome of benchmarking, which we felt was better done on a real system, but to increase our confidence in the efficiency of the simple, direct-mapped cache and the choice of the coherence protocol. Our analysis uses simulation to obtain the characteristics of a single processor and its cache, then uses a simple queueing model to predict the performance of the full system as a function of the number of processors. The estimate was done for the MicroVAX version of the Firefly. The CVAX version includes an on-chip cache that further complicates the analysis. When designing the CVAX processor module, we assumed that this cache, as well as the larger off-chip cache, would decrease the miss rates by an amount that would make

up for the increased speed of the processor and allow us to continue to use the 10 Mbyte/s MBus of the original Firefly.

To carry out the analysis, we need to know the memory reference behavior of the processor, and the performance of the cache. Measurements made on the VAX [5] show that a typical instruction does 0.95 (Rir) instruction reads per instruction, 0.78 (Rdr) data reads, and 0.40 (Rw) data writes, for a total of 2.13 (Rt) references per instruction. This is an architectural property averaged over a number of applications, and does not depend on the particular CPU implementation.

Trace-driven simulation of the MicroVAX CPU, carried out for us by D. Zukowski of the DEC Eastern Research Laboratory, showed it to be an 11.9 tick-per-instruction (TPI)[1] implementation of the VAX architecture when operating with a memory that introduces no wait states. These simulations also showed that the Firefly cache achieves a miss rate $M$ of 0.2,[2] and that the fraction Dv of cache entries that are dirty is 0.25. The remaining fact that we need for the analysis is the degree of data sharing. Since multiprocessor traces were not available, this parameter was estimated. We arbitrarily assumed that a fraction $S = 0.1$ of the processor's writes are to shared data.

To predict the amount by which each processor in the Firefly is slowed due to contention for the bus, we model the bus and memory as a closed queueing network with a single server and a number of customers equal to the number of processors. This view of the system is analogous to models of time sharing systems with terminal workloads in which users spend a fixed amount of time thinking, then submit service requests to a CPU. In our analysis, think time corresponds to the time between MBus operations in an unloaded system, while the service demand is the time required for a memory reference, two ticks. Algorithm 6.2 in [9] provides an exact solution to this problem. Given the think time, the service demand $D$, and the number of customers $n$, it yields the residence time $R$ (the time a request spends waiting for the bus and doing a reference), the throughput of the bus, the number of processors waiting for the bus, and the bus utilization $U$ (the fraction of nonidle bus cycles). The throughput and queue length are not important for our estimate, but the residence time, when combined with the cache miss rate, determines the speed of an individual processor (TPI). The bus utilization is easily measured in the real system, and can therefore be used to validate the model. This model ignores the fact that the caches have fixed priority for access to the MBus. This reduces the delays incurred by high-priority caches at the expense of those with lower priority.

There are three effects that increase the number of ticks per instruction above 11.9 TPI.

- Misses: One cycle of the last reference that occurs during a miss is counted in the think time (reflecting the fact that a clean miss in an unloaded system adds only one tick to the access time). Victim writes are not overlapped, so the MBus write takes the full $R$ ticks. The slowdown due to misses (added ticks per instruction) is, therefore,

$$\mathrm{Sm} = \mathrm{Rt} * M * (R-1) + \mathrm{Rt} * M * \mathrm{Dv} * R \text{ (ticks per instruction).}$$

- Write-through: This is the product of the write hits per instruction, the fraction of writes to shared data, and the number of ticks per MBus write:

$$\mathrm{Sw} = \mathrm{Rw} * (1-M) \text{ (write hits per instruction)} * S \text{ (fraction of shared writeable data)} * (R-1) \text{ (ticks per MBus reference).}$$

- Tag store probes by other caches: Each CPU cache reference that hits without doing a write-through will be slowed by one tick if an MBus operation initiated by another cache needs to access the tag store during the same cycle as the CPU. The number of ticks per instruction lost to this effect is the product of the number of cache hits per instruction and the probability that a particular tick will be needed by an MBus operation:

$$\mathrm{Sp} = (\mathrm{Rdr} + \mathrm{Rir} + \mathrm{Rw} * (1-S)) \text{ (references that do not require the MBus)} * (1-M) \text{ (fraction of references that hit)} * U/D \text{ (probability of one-tick slowdown).}$$

- This time must also be included in the calculation of the think time,[3] since it increases the time between memory references. The utilization used to calculate Sp is that for a system containing one less CPU than is actually present, since we are calculating the effect on the $n$th processor due to the $(n-1)$ other processors in the system. Note also that CPU operations that miss in the cache are not counted in this effect, since the cycle lost in this case is fully overlapped with the MBus operation that caused the delay.

These effects are additive, so the total number of ticks per instruction for a single processor is

$$\mathrm{TPI} = 11.9 + \mathrm{Sm} + \mathrm{Sw} + \mathrm{Sp}.$$

The performance of a single processor in an $n$-processor system relative to a processor in a one-processor system, RP, is the ratio of the TPI's of the two processors. The performance of an $n$-processor system relative to a single-processor system, SRP, is just $n$ times this value.

[1] We use the term *tick* rather than the more familiar *cycle* to emphasize that in the MicroVAX CPU, the basic time quantum is *two* 100 ns Firefly clock cycles.

[2] This is an abnormally large miss rate for a 16 kbyte cache. We attribute it to the small line size (4 bytes). A larger line would probably have reduced the miss rate considerably, but it would have complicated the design of the cache, the MBus, and the storage modules.

[3] The think time is $(11.9 + \mathrm{Sp})/((\mathrm{Rt} * M * (1 + \mathrm{Dv})) + (\mathrm{Rw} * S))$, the interval in ticks between memory references due to read misses and write through.

TABLE I
FIREFLY ESTIMATED PERFORMANCE

| n | R | U | Sm | Sw | Sp | TPI | RP | SRP |
|---|---|---|---|---|---|---|---|---|
| 1 | 2.00 | 0.09 | 0.64 | 0.00 | 0.00 | 12.54 | 1.00 | 1.00 |
| 2 | 2.18 | 0.17 | 0.73 | 0.04 | 0.07 | 12.74 | 0.98 | 1.97 |
| 3 | 2.38 | 0.26 | 0.84 | 0.04 | 0.15 | 12.93 | 0.97 | 2.91 |
| 4 | 2.61 | 0.34 | 0.97 | 0.05 | 0.22 | 13.13 | 0.95 | 3.82 |
| 5 | 2.88 | 0.42 | 1.11 | 0.06 | 0.28 | 13.35 | 0.94 | 4.70 |
| 6 | 3.20 | 0.49 | 1.28 | 0.07 | 0.35 | 13.60 | 0.92 | 5.53 |
| 7 | 3.57 | 0.56 | 1.47 | 0.08 | 0.41 | 13.87 | 0.90 | 6.33 |
| 8 | 4.00 | 0.63 | 1.70 | 0.10 | 0.47 | 14.17 | 0.88 | 7.08 |
| 9 | 4.51 | 0.69 | 1.98 | 0.11 | 0.52 | 14.51 | 0.86 | 7.78 |
| 10 | 5.11 | 0.75 | 2.29 | 0.13 | 0.58 | 14.90 | 0.84 | 8.41 |
| 11 | 5.81 | 0.80 | 2.67 | 0.15 | 0.62 | 15.35 | 0.82 | 8.99 |
| 12 | 6.63 | 0.84 | 3.11 | 0.18 | 0.67 | 15.85 | 0.79 | 9.49 |
| 13 | 7.58 | 0.88 | 3.61 | 0.21 | 0.70 | 16.43 | 0.76 | 9.92 |
| 14 | 8.68 | 0.91 | 4.19 | 0.25 | 0.74 | 17.08 | 0.73 | 10.28 |
| 15 | 9.92 | 0.94 | 4.85 | 0.29 | 0.76 | 17.80 | 0.70 | 10.57 |
| 16 | 11.30 | 0.96 | 5.59 | 0.33 | 0.78 | 18.60 | 0.67 | 10.78 |

Table I summarizes the results of the Firefly performance estimate for systems containing from 1 to 16 processors. The effects of bus saturation are clearly visible. The fifth processor adds almost 90 percent of the performance of a single CPU, while the tenth processor contributes only 63 percent. The cost of the conditional write-through protocol is small—even with the 10 percent sharing assumed, write-through accounts for less than one percent of the TPI in the five-processor configuration. The cost of providing the snoopy cache is somewhat larger, since about 2 percent of the TPI is due to tag store contention. In a system with a high level of sharing or with a CPU that referenced memory more frequently, it might be appropriate to duplicate the cache tag store and service MBus requests from the second store, but this is clearly unnecessary in the Firefly.

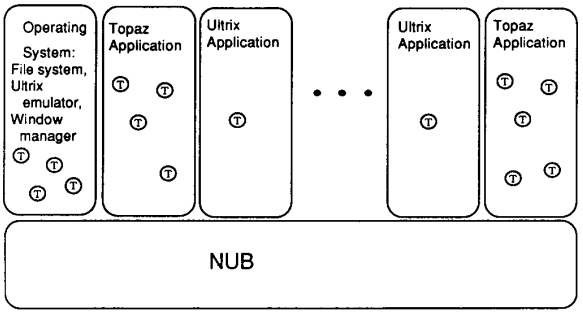



Fig. 4. Internal structure of Topaz. The nub runs in VAX kernel mode and provides low-level services. Other operating system services are provided by a user-mode process running in a separate address space. Programs in different address spaces communicate using remote procedure call. Ultrix address spaces contain only one thread of control, while Topaz address spaces may have many threads.

## III. Software

The Firefly's operating system is called Topaz. The key facilities of Topaz that are visible to users are execution of existing Ultrix binaries, a remote file system, and a display manager called trestle that provides both tiled and overlapping windows. The principal facilities of Topaz that are visible to programs are multiple threads of control in the same or different address spaces, and pervasive support for remote procedure calls.

### *A. Topaz Structure*

The components of Topaz are shown schematically in Fig. 4. The nub runs in VAX kernel mode and provides low-level services including virtual memory, thread scheduling, simple device drivers, and the remote procedure call transport mechanism. All other system software, including the operating system facilities not provided by the nub, a debugging server, and the window manager, runs in VAX user mode in an ordinary address space.

There are two kinds of user address space. An Ultrix address space contains a single thread of control, and provides an environment in which most VAX Ultrix binaries can run unchanged. A Topaz address space provides the same facilities, but in addition permits multiple threads of control to execute simultaneously. The operating system is multithreaded, so even single-threaded Ultrix applications are able to take some advantage of the concurrency provided by the Firefly. A typical Firefly operates with tens of address space and hundreds of threads.

In many operating systems, the notion of a process includes a thread of control, a complete virtual address space, and a great deal of operating system state such as a suite of open files, user identification, current working directory, a collection of event handlers, and so forth. This complexity leads to unwieldy and slow process creation. The Topaz notion of thread is restricted to the thread of control. The creation of a new address space, forking of a new thread, and the manipulation of other operating system state are all independent.

Remote procedure call provides communication between address spaces on a single machine and between machines on the SRC network. Since procedure call semantics are very familiar to programmers, using this paradigm permits the construction of complex distributed applications by people without extensive communications experience. RPC, together with inexpensive threads, permits all input–output and communications services to have synchronous interfaces. If asynchronous behavior is desired, one simply forks a new thread to make the synchronous call.

The trestle window manager handles allocation of display real estate and multiplexing of the keyboard and mouse among applications. No particular style of user interface is mandated, but applications are encouraged towards cooperation and consistency through the easy availability of a set of trestle services. Applications programs communicate with trestle by remote procedure call, and can make use of displays attached to remote machines if desired. A protocol conversion utility is available that allows an *X* window system [13] application running on a mainframe to use a trestle window as an *X* server. Trestle also takes advantage of multiprocessing, since once a client program has transmitted a painting request to trestle, the client is free to proceed, while other threads in the trestle server complete the request.

Firefly workstations are not independent entities. A user's Firefly participates as a component in a distributed computing environment which also includes larger mainframe computers and a variety of servers. Accurate time, network naming, internetwork communications, print, compute, and file servers are available. Many of these servers are implemented on Fireflies. The Firefly contributes to high file server performance in two ways: requests from independent clients can be processed in parallel, and RPC network file transfers use multiple threads to achieve high throughput. Workstations, when idle, act as compute servers which are available to the active users. This facility has been used for such projects as factoring very large numbers and rebuilding large software systems.

### *B. Programming Language and Environment*

Both Topaz and new applications software are largely written in Modula-2+. Modula-2+ is a version of Modula-2 augmented by language support for garbage-collected storage, exception handling, and concurrency. Modula-2 was chosen as the base for SRC's systems programming language because it, among several alternatives examined, came closest to meeting our needs. The primary features that were considered essential, and which Modula-2 provided, were strong typing with compile-time checking, and separate specification of interfaces and their implementations. The latter feature was seen as particularly important, since it provides a precise specification of the behavior of a software abstraction, while allowing the implementation of that abstraction to change. This is particularly valuable when a number of people cooperate on large programming tasks.

The most controversial change made to the language was probably the provision of garbage-collected storage. Garbage collection has a run-time cost, but it relieves the programmer of the responsibility for properly deallocating storage, which can be a significant source of errors in complex systems. The changes to the language for garbage collection include the addition of REF types. REF's are similar to pointers, except that the compiler and the run-time system keep track of the number of extant copies of a REF, and when this count becomes zero, the referent is automatically deallocated. The collector runs as a separate thread, concurrently with the application.

Modula-2+ facilities for concurrency are provided by a combination of the compiler and the run-time system's threads module. The threads module provides fork and join operations on threads, and wait and signal operations on condition variables. The compiler provides a LOCK statement which acquires a mutex, or mutual exclusion variable, for the duration of its scope. Essentially, the LOCK statement declares a critical section.

Exceptions allow a precise specification of the ways in which a software abstraction can fail, and allow failures to be passed through intervening layers of abstraction (i.e., up the call stack) to the layer that is prepared for the failure. Finalization complements exceptions by providing a place for a block of code to restore its invariants (i.e., clean up its state) both on normal and exceptional exit. Finalization is provided by a TRY ··· FINALLY block. Statements within the block are executed, and regardless of the detailed flow of control, the code in the FINALLY clause is executed before control leaves the dynamic scope of the block. Exceptions are provided by the standard procedure RAISE, and a TRY ··· EXCEPT block. The RAISE procedure initiates an exception, and the EXCEPT clause provides a place to declare exception handlers which will be activated if necessary while control is still within the dynamic scope of the block. Modula-2+ exceptions are nonlocal gotos with dynamic binding, and have termination semantics. When an exception is raised, the call stack is searched for the nearest handler for that exception. Before control passes to the handler, the call stack is unwound. If there is no handler, the complete state is made available to the debugger.

Modula-2+ programs are debugged using a source-language debugger known as *Loupe*. Loupe can be used to debug programs running on the same machine, or it can be used remotely over the network. No prearrangement is required to use Loupe. Given a machine name and a process ID, it can connect to the machine and bring the process and all its threads to an orderly halt in preparation for debugging. It is then possible to examine variables and data structures by name, call procedures in the program being examined, and set breakpoints at source statement boundaries. When the debugging session is finished, the target program may be restarted from the point of interruption. Loupe has been invaluable in the development of software for the Firefly, since it can be used to debug any system component, from operating system device drivers to application programs.

## IV. Hardware Performance Measurements

The choice of benchmarks is always difficult. For the purposes of this paper, we examine how the Firefly architecture and implementation react under load.

The first performance measurements were made in 1986, quite early in the evolution of the Firefly. We wanted to get some early insight into the behavior of the Firefly memory system. The measurements show that the hardware performs about as well as expected. Table II contains the results of one measurement. The reference rates are measured using a counter connected to the hardware, and span several minutes of execution of the target program.

The program used in this example is an exerciser for the

TABLE II
FIREFLY MEASURED PERFORMANCE (K REFS/S)

| | One-CPU system | | Five-CPU system | |
|---|---|---|---|---|
| | Expected | Actual | Expected | Actual |
| Per CPU: | | | | |
| Reads | 688 | 1125 | 609 | 850 |
| Writes | 161 | 240 | 143 | 225 |
| Total | 849 | 1365 | 752 | 1075 |
| MBus Total References: | | | | |
| | 440 (U=.18) | | 1350 (U=.54) | |
| MBus References, Per CPU: | | | | |
| Reads: | 340 (M=.3) | | 145 (M=.17) | |
| Writes | | | | |
| That received MShared: | | | | |
| | 0 | | 75 | |
| That did not receive MShared: | | | | |
| | 50 | | 20 | |
| Victims: | | | | |
| | 50 | | 10 | |

Topaz threads package. The program forks a number of threads, each of which then executes and checks the results of threads package primitive operations. There is a great deal of synchronization and process migration, since the threads deliberately block and reschedule themselves.

The most surprising thing about this program is that it makes references at a much greater rate than the suite of programs used in the simulations. We would expect a one-CPU system to make about 850K references per second when connected to a Firefly cache that adds one tick to every operation that misses, plus two ticks for every dirty victim write. Instead, we see 1350K references. Part of the discrepancy can be explained by the fact that the CPU does instruction prefetching, which was not simulated. If the prefetching were perfect, instruction fetches would occur, but they would be overlapped with the execution of earlier instructions, and would not affect the rate of instruction issue. The instruction rate would increase to 476K instructions/s (10.5 TPI) and the reference rate would be 1014K references/s. Prefetching should not be perfect, however, since branches and memory-limited instructions reduce the amount of overlap the prefetcher can achieve. On the other hand, instructions that are prefetched but not executed increase the reference rate without increasing the issue rate, so perhaps the results are not as surprising as they seem initially. Note that the *ratio* of reads to writes is different in the one-CPU(4.7:1) and the five-CPU (3.8:1) cases, indicating that prefetches occur less frequently when bus loading slows nonprefetch references.

Our measurement method can distinguish three categories of MBus write: nonvictim writes that receive MShared from other caches, nonvictim writes that do not receive MShared, and victim writes. For this program, the estimate of 10 percent sharing is clearly too low, since 75K of the 225K writes done by one CPU (33 percent) were write-throughs that received MShared. The difference between the estimated and actual sharing is not too surprising, since the program is designed to test operations that depend heavily on sharing. The number of victim writes is much lower than predicted by our simple model, since write-throughs leave cache lines clean.

The most important cache parameters in the Firefly are the cache miss rate and the bus loading. In the five-processor system, the cache miss rate is quite close to the prediction, and the bus load is slightly higher than predicted, as would be expected from the higher reference rate of a single processor. The miss rate of a single processor system is much higher than expected, possibly due to cold-start effects caused by rapid context switching.

More recently, with a more complete software environment at our disposal, we have made some additional measurements of both the MicroVAX Firefly, now in general use, and the CVAX Firefly, now entering service.

We first report some measurements of Regsim, which is a register transfer level hardware simulator. Regsim is a single-threaded, compute bound program. On a uniprocessor, running multiple copies of such an application will not increase throughput, but on a multiprocessor, it makes sense to run multiple compute bound applications.

MBus utilization, user-mode run time, and wall-clock run time were measured for varying numbers of copies of Regsim (using the same input data) running on both the MicroVAX Firefly with five processors and the CVAX Firefly with four CVAX's. For each machine, two sets of runs were made, with processes that were made free to run anywhere, or constrained to particular processors by the scheduler. The CVAX data also include a partial breakdown of MBus traffic into MBus read cycles and MBus write cycles in which MShared was asserted. asserted.

Table III shows the number of seconds spent computing (user mode) and the number of seconds to complete the job (wall-clock seconds). The speedup figures present aggregate throughput as the ratio of $N$ times the wall-clock time for one copy on the MicroVAX (42 s) to that for $N$ copies.

Wall-clock time is the measure of system throughput perceived by a user, but it includes many effects besides those of the hardware, such as the compression of workstation overhead activity and the limiting behavior of the file system. The user-mode execution times are more useful for evaluating how increased memory system activity slows an application, since the application follows approximately the same sequence of instructions in each case.

The MicroVAX figures track our performance estimates fairly well. When four copies of the application are running, the user-mode running time increases from 42 to 45 s, indicating a relative processor performance of 0.93 at an MBus utilization of just over 50 percent. The reference rate per processor is higher than anticipated, as discussed earlier, but the processor performance as a function of bus utilization matches.

The measurements of the CVAX system are much more interesting. First, when "idle," the CVAX system displays an extremely low cache miss rate, indicating that the working set of the various idle jobs fits within the 64 kbyte caches. However, the idle system also exhibits a large number of write-through cycles, indicating that essentially all written variables reside in multiple caches, requiring updates. This same body of writeable data is responsible for the *decrease* in write-through traffic with increasing load, when the Regsim processes are constrained to particular processors. In these cases, the Regsim processes flush the writeable data out of several caches, reducing the required update traffic.

TABLE III
FIREFLY PERFORMANCE IN A COMPUTE BOUND APPLICATION

| Number of copies: | idle | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| **MicroVAX, unconstrained processes:** | | | | | | |
| User-mode seconds | | 42 | 43 | 45 | 48 | 51 |
| Wall-clock seconds | | 42 | 43 | 46 | 51 | 63 |
| Wall-clock speedup | | 1.0 | 2.0 | 2.7 | 3.3 | 3.3 |
| MBus Utilization | .15 | .21 | .26 | .40 | .52 | .64 |
| **MicroVAX, constrained processes:** | | | | | | |
| User-mode seconds | | 42 | 43 | 44 | 45 | 49 |
| Wall-clock seconds | | 42 | 43 | 46 | 51 | 58 |
| Wall-clock speedup | | 1.0 | 2.0 | 2.7 | 3.3 | 3.6 |
| MBus Utilization | .15 | .18 | .30 | .35 | .44 | .50 |
| **CVAX, unconstrained processes:** | | | | | | |
| User-mode seconds | | 19 | 20 | 22 | 23 | -- |
| Wall-clock seconds | | 19 | 20 | 22 | 26 | -- |
| Wall-clock speedup | | 2.2 | 4.2 | 5.7 | 6.5 | -- |
| MBus Utilization | .28 | .28 | .40 | .62 | .72 | -- |
| MBus Reads | .01 | .07 | .14 | .23 | .28 | -- |
| MBus Shared Writes | .27 | .20 | .24 | .36 | .44 | -- |
| **CVAX, constrained processes:** | | | | | | |
| User-mode seconds | | 19 | 19 | 19 | 20 | -- |
| Wall-clock seconds | | 20 | 20 | 21 | 23 | -- |
| Wall-clock speedup | | 2.1 | 4.2 | 6.0 | 7.3 | -- |
| MBus Utilization | .28 | .32 | .28 | .32 | .37 | -- |
| MBus Reads | .01 | .07 | .14 | .20 | .25 | -- |
| MBus Shared Writes | .27 | .16 | .08 | .07 | .03 | -- |

Note: MBus operations are stated as the fraction of the 2.5 million operations per second that are available.

TABLE IV
FFT SPEEDUP

| Active CPUs | Average CPUs busy | Speedup | |
|---|---|---|---|
| 1 | 1.25 | 1 | (reference) |
| 2 | 2.1 | 2.2 | |
| 3 | 3.0 | 3.4 | |
| 4 | 3.7 | 4.2 | |
| 5 | 4.8 | 5.1 | |

Most interesting of all is the difference between the constrained and unconstrained cases for the two systems. In the unconstrained cases, the Regsim processes are frequently rescheduled on different processors because of the activities of other threads. This process migration results in increased write-through traffic on the bus. The loss in performance due to this effect on the MicroVAX system (3 user-mode seconds out of 45) is about 6 percent. On the CVAX system, however, the performance loss is 15 percent due to the much larger caches. The 1.1 million MBus shared writes in the 4 process unconstrained case on the CVAX system correspond to about 50 percent of CPU writes directed to shared cache entries. Careful scheduler design will be necessary to minimize process migration as caches become still larger. Alternatively, advanced coherence protocols, such as those discussed in [7], can minimize the effects of worst case sequences of operations.

These preliminary measurements of the CVAX Firefly confirm our expectation that the combination of a faster processor and larger cache results in approximately the same bus load per processor. On our benchmarks, the upgrade has improved execution speeds by factors of 2.0–2.5. This is less than the 2.5–3.2 speedup reported for other systems that use the new CVAX processor. We have sacrificed some potential performance by choosing not to use the on-chip cache for data and by retaining the original MBus timing.

Regsim illustrates the performance of the Firefly for large programs that make heavy use of the MBus. Applications with small working sets can achieve still more efficient parallelism. J. Saxe has implemented a parallel FFT for signal processing which achieves an apparent super-linear speedup with parallelism on the MicroVAX Firefly (Table IV). Super-linear effects occur when overhead activities are spread out over a large number of processors rather than competing with the application for one or a few processors. This effect is only apparent when wall-clock times are measured, rather than user-mode execution time.

The trestle window system also makes use of the parallelism available in the Firefly. Client programs make painting requests to trestle, which delivers the request to a three-stage

TABLE V
TRESTLE SPEEDUP

| Active CPUs | User-time | Real-time | Speedup | MBus busy |
|---|---|---|---|---|
| 1 uVAX | 144 | 600 | 1.0 | .25 |
| 2 uVAX | 149 | 290 | 2.1 | .40 |
| 3 uVAX | 142 | 199 | 3.0 | .55 |
| 4 uVAX | 138 | 156 | 3.8 | .65 |
| 5 uVAX | 136 | 143 | 4.2 | .70 |
| 4 (CVAX) | 55 | 66 | 9.1 | .85 |

TABLE VI
PARALLEL REGSIM SPEEDUP

| Active Threads | Real-time (seconds) | Objects evaluated | Speedup (real time) | Speedup (evals/sec) | MBus busy |
|---|---|---|---|---|---|
| 1 | 313 | 1.43M | 1.0 | 1.0 | .19 |
| 2 | 191 | 1.64M | 1.6 | 1.9 | .25 |
| 3 | 146 | 1.75M | 2.1 | 2.6 | .35 |
| 4 | 127 | 1.83M | 2.5 | 3.2 | .46 |
| 5 | 121 | 1.96M | 2.6 | 3.6 | .48 |

concurrent pipeline and quickly returns to the application. Table V shows the user-mode execution time and wall-clock execution time in microseconds for a simple painting request.

Some parallel applications do not speed up as well as others. We have implemented a parallel version of the Regsim simulator that demonstrates some of the difficulties of building parallel applications. This program uses an object-oriented representation of the circuit being simulated. Each object represents a register, logic element, or memory. During each simulated clock cycle, an object is called upon to evaluate itself if its value is needed by another object. The objects that must be evaluated each cycle (those which have internal state or whose outputs are to be inspected) are divided equally among the threads available to the evaluator. The number of evaluation threads is a parameter of the program, and two additional threads are used: one to read ahead in the input file, and one to write the results of the simulation to another file. This method has the advantage that the evaluator threads only need to synchronize at simulated clock-cycle boundaries. Very little speedup would be achieved if the evaluation of every object were a critical section, since the time to evaluate an object is comparable to the time needed to synchronize. Since most objects are functional (i.e., their outputs depend only on their inputs, not on internal state), it is not necessary to ensure that an object is evaluated only once per cycle, and multiple evaluations occur more frequently as the number of active threads increases. This effect limits the wall-clock speedup that can be obtained (which is the quantity of interest to a user), but the speedup measured on the basis of evaluations per second is still respectable. Table VI shows the speedup and the MBus utilization for this program running on a five-processor Firefly as the number of evaluation threads is varied between one and five. Speedup ceases when the number of threads approaches the number of processors. This program does not attempt to constrain threads to particular processors, so process migration and operating system overhead activities also limit the speedup.

As will be apparent by now, the performance of a shared memory multiprocessor which uses caches to reduce the load on the main storage bus depends critically on the cache performance. For example, a single Firefly processor, executing a block copy instruction, can achieve a 100 percent miss rate and use two thirds of the MBus bandwidth. Two processors executing block copy instructions at once will completely load the bus.

## V. CONCLUSIONS

The Firefly hardware is now complete, and five-processor systems are being used by all members of the SRC staff as their primary source of computing. We have built a few seven- and nine-processor systems for experimentation. Our initial belief that it would be possible to build an interesting and useful system by limiting our aspirations and using available technology wherever possible has been verified. The system has met its performance goals, and the hardware reliability has been more than adequate.

The Firefly software is still evolving rapidly. We have had considerable success in utilizing the coarse-grained parallelism that the multiprocessor provides. When a user carries out a few unrelated activities simultaneously, the performance of the system is much more predictable than that of a time-shared uniprocessor. Topaz and a few applications have been able to exploit medium-gained parallelism in useful ways. Attempts to redesign other applications to use multiprocessors effectively have met with mixed success. We still have much to learn about whether or how we can do this routinely for a wide range of applications. If we are successful, shared-memory multiprocessors using snoopy caches can provide a cost-effective way of providing high-performance workstations using modest-performance technologies.

## ACKNOWLEDGMENT

They entire staff of the Systems Research Center has contributed to the Firefly, so it is difficult to list everyone. The authors were responsible for the caches and memory system of the Firefly. J. Dillon designed the new storage modules, and B. McNamara, C. Peters, and P. Petit made substantial contributions to the hardware. P. Rovner, V. Cavalli-Sforza,

and J. Horning built the Modula-2+ system, starting from M. Powell's Modula-2 compiler. R. Levin, M. Schroeder, G. Swart, P. McJones, A. Birrell, and B. Lampson developed much of Topaz. The trestle window manager was designed and built by G. Nelson, M. Manasse, and M. Brown. We would also like to thank the anonymous reviewers, whose comments on an earlier draft improved the presentation considerably.

## References

[1] J. Archibald and J. L. Baer, "Cache coherence protocols: Evaluation using a multiprocessor simulation model," *ACM Trans. Comput. Syst.*, vol. 4, pp.273–298, Nov. 1986.

[2] D. W. Archer, D. R. Deverell, T. F. Fox, P. E. Gronowski, A. K. Jain, M. Leary, D. G. Miner, A. Olesin, S. D. Persels, P. I. Rubinfeld, and R. M. Supnik, "A CMOS VAX microprocessor with on-chip cache and memory management," *IEEE J. Solid-State Circuits,* vol. SC-22, pp. 849–852, Oct. 1987.

[3] A. D. Birrell, J. V. Guttag, J. J. Horning, and R. Levin, "Synchronization primitives for a multiprocessor: A formal specification," Tech. Rep. 20, DEC Syst. Res. Center, Aug. 1987.

[4] D. W. Dobbernpuhl, R. M. Supnik, and R. T. Witek, "The MicroVAX 78032 chip, a 32 bit microprocessor," *Dig. Tech. J.*, Mar. 1986.

[5] J. S. Emer and D. W. Clark, "A characterization of processor performance in the VAX-11/780," in *Proc. 10th Annu. Symp. Comput. Architecture,* IEEE, June 1984, pp. 301–310.

[6] D. Ingalls, "The Smalltalk graphics kernel," *Byte,* vol. 6, pp. 168–194, Aug. 1981.

[7] A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator, "Competitive snoopy caching," in *Proc. 27th Annu. Symp. Foundations Comput. Sci.*, IEEE, Oct. 27–29, 1986, pp. 214–254.

[8] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon, "Implementing a cache consistency protocol," in *Proc. 12th Int. Symp. Comput. Architecture,* IEEE, 1985.

[9] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative System Performance: Computer System Analysis Using Queueing Network Models.* Englewood Cliffs, NJ: Prentice-Hall, 1984.

[10] E. M. McCreight, "The Dragon computer system, an early overview," in *NATO Adv. Study Instit. Microarchitecture VLSI Comput.*, July, 1984.

[11] P. R. McJones and G. F. Swart, "Evolving the UNIX system interface to support multithreaded programs," Tech. Rep. 21, DEC Syst. Res. Center, Sept. 1987.

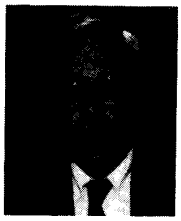
[12] P. Rovner, "Extending Modula-2 to build large integrated systems," *IEEE Software,* vol. 6, pp. 46–57, Nov. 1986.

[13] R. W. Scheifler and J. Gettys, "The X window system," *ACM Trans. Graphics,* vol. 5, pp. 79–109, Apr. 1986.

[14] D. P. Siewiorek, C. Gordon Bell, and A. Newell, *Computer Structures, Principles and Examples.* New York: McGraw-Hill, 1982.

[15] A. J. Smith, "Cache memories," *ACM Comput. Surveys,* vol. 14, pp. 473–530, Sept. 1982.

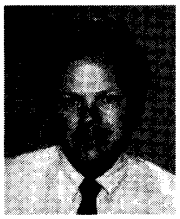
[16] N. Wirth, *Programming in Modula-2,* 3rd ed. Berlin, Germany: Springer-Verlag, 1985.



**Charles P. Thacker** received the A.B. degree in physics from the University of California, Berkeley, in 1967.

He has been with the Systems Research Center of Digital Equipment Corporation, Palo Alto, CA, since 1983. Before joining Digital, he was a Senior Research Fellow at the Xerox Palo Alto Research Center, which he joined in 1970. While at Xerox, he was responsible for the design of a number of computer systems, including the first personal workstation, the Alto. He holds several patents in the area of computer organization, and is a coinventor of the Ethernet local-area network. His research interests include computer architecture, computer networking, and computer-aided design.

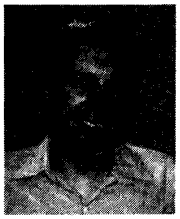
Mr. Thacker is a member of the Association for Computing Machinery.



**Lawrence C. Stewart** (S'74–M'80) received the S.B. degree from the Massachusetts Institute of Technology, Cambridge, in 1976, and the M.S. and Ph.D. degrees from Stanford University, in 1977 and 1981, respectively, all in electrical engineering.

From 1981 to 1984, he was with the Xerox Palo Alto Research Center Computer Science Laboratory. Since 1984, he has been with the Digital Equipment Corporation Systems Research Center, Palo Alto, CA. His current interests are in computer hardware, architecture, and systems, data compression, voice, and computer communications.

Dr. Stewart is a member of Eta Kappa Nu, Tau Beta Pi, and the Association for Computing Machinery.



**Edwin H. Satterthwaite, Jr.** (M'85) received the B.S. degree in chemistry from the University of Delaware, Newark, and the M.S. and Ph.D. degrees in computer science from Stanford University, Stanford, CA.

Before joining the Systems Research Center of Digital Equipment Corporation, Palo Alto, CA, in 1986, he was a Senior Member of the Research Staff at the Xerox Palo Alto Research Center, where he was involved in the design and implementation of compilers and integrated programming systems. He has previously published in the areas of symbolic debugging, parsing, design of programming languages, and modular software construction. His current research interests include computer architecture, communications networks, and the design and implementation of programming languages.