23

Firefly: A Multiprocessor Workstation

by Charles P. Thacker, Lawrence C. Stewart, and Edwin H. Satterthwaite Jr.

December 30, 1987

digital

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

Systems Research Center

DEC's business and technology objectives require a strong research program. The Systems Research Center (SRC) and three other research laboratories are committed to filling that need.

SRC began recruiting its first research scientists in 1984 — their charter, to advance the state of knowledge in all aspects of computer systems research. Our current work includes exploring high-performance personal computing, distributed computing, programming environments, system modelling techniques, specification technology, and tightly-coupled multiprocessors.

Our approach to both hardware and software research is to create and use real systems so that we can investigate their properties fully. Complex systems cannot be evaluated solely in the abstract. Based on this belief, our strategy is to demonstrate the technical and practical feasibility of our ideas by building prototypes and using them as daily tools. The experience we gain is useful in the short term in enabling us to refine our designs, and invaluable in the long term in helping us to advance the state of knowledge about those systems. Most of the major advances in information systems have come through this strategy, including time-sharing, the ArpaNet, and distributed personal computing.

SRC also performs work of a more mathematical flavor which complements our systems research. Some of this work is in established fields of theoretical computer science, such as the analysis of algorithms, computational geometry, and logics of programming. The rest of this work explores new ground motivated by problems that arise in our systems research.

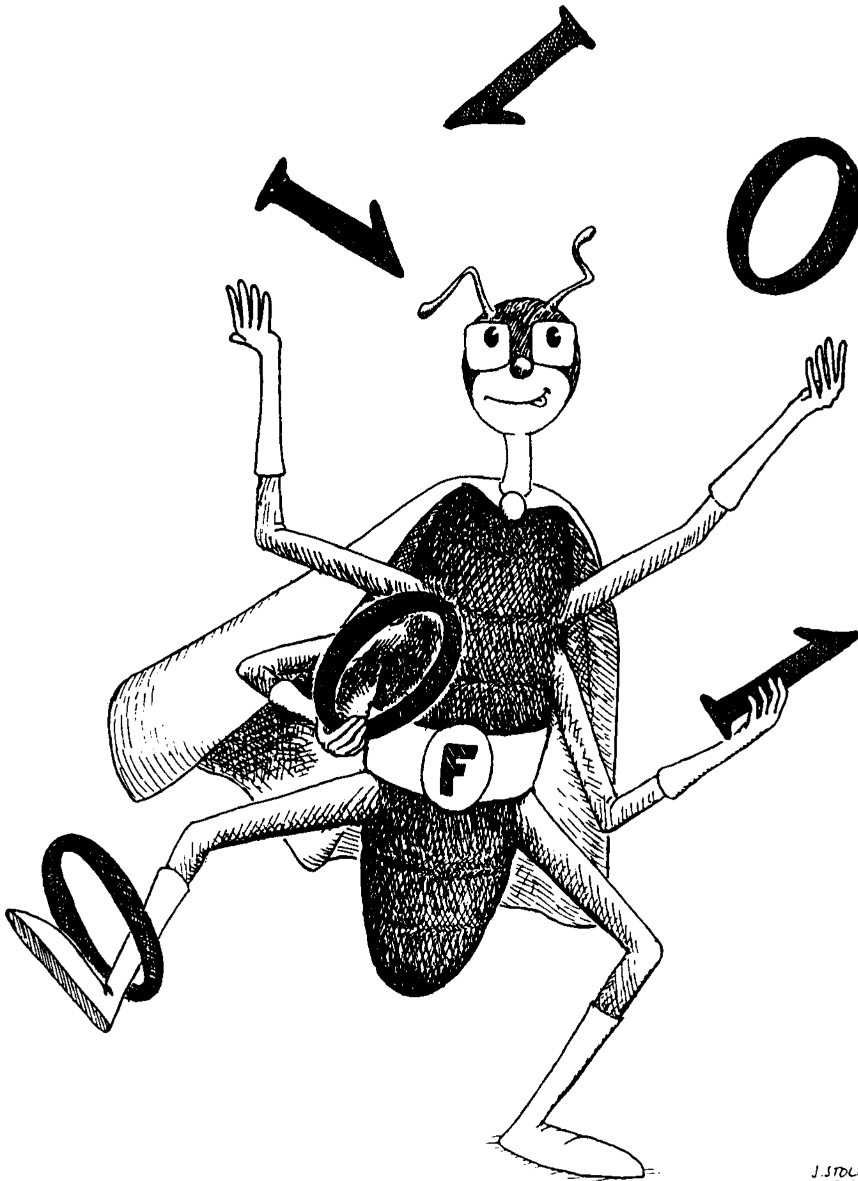
DEC has a strong commitment to communicating the results and experience gained through pursuing these activities. The Company values the improved understanding that comes with exposing and testing our ideas within the research community. SRC will therefore report results in conferences, in professional journals, and in our research report series. We will seek users for our prototype systems among those with whom we have common research interests, and we will encourage collaboration with university researchers.

Robert W. Taylor, Director

Firefly: A Multiprocessor Workstation

Charles P. Thacker, Lawrence C. Stewart, and Edwin H. Satterthwaite Jr.

December 30, 1987



Publication history

This paper has appeared in the *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, October 5-8, 1987, in Palo Alto.

The proceedings were published jointly by IEEE and ACM as:

Operating Systems Review 21:4, October 1987 (IEEE), *Computer Architecture News* 15:5, October 1987 (ACM), and *Sigplan Notices* 22:10, October 1987 (ACM).

Copyright 1987 Association for Computing Machinery Inc. Reprinted by permission.

©Digital Equipment Corporation 1987

This work may not be copied or reproduced in whole or in part for any commercial purpose. Permission to copy in whole or in part without payment of fee is granted for nonprofit educational and research purposes provided that all such whole or partial copies include the following: a notice that such copying is by permission of the Systems Research Center of Digital Equipment Corporation in Palo Alto, California; an acknowledgment of the authors and individual contributors to the work; and all applicable portions of the copyright notice. Copying, reproducing, or republishing for any other purpose shall require a license with payment of fee to the Systems Research Center. All rights reserved.

Authors' abstract

The Firefly is a shared-memory multiprocessor workstation that is used as the primary source of computing at the Digital Equipment Corporation, Systems Research Center (SRC). Two versions of the Firefly have been built. The first version contains from one to seven MicroVAX 78032 processors, each with a floating point unit and a sixteen kilobyte cache. The caches are coherent, so that all processors see a consistent view of main memory. A system may contain from four to sixteen megabytes of storage. Input-output is done via a standard DEC QBus. Input-output devices are an Ethernet controller, fixed disks, and a monochrome 1024 x 768 display with keyboard and mouse. Optional hardware includes a high resolution color display and a controller for high capacity disks. The second version of the Firefly contains faster CVAX 78034 processors, sixty-four kilobyte caches, and a main memory of up to 128 megabytes.

The Firefly runs a software system that emulates the Ultrix system call interface. It also supports medium and coarse-grained multiprocessing through multiple threads of control in a single address space. Communication is implemented uniformly through the use of remote procedure calls.

This report describes the goals, architecture, implementation, and performance analysis of the Firefly. It then presents some measurements of hardware performance, and concludes with some brief remarks on the evolution of the software.

Charles P. Thacker, Lawrence C. Stewart, and Edwin H. Satterthwaite Jr.

Capsule review

Multiprocessors have become a commonplace. Most multiprocessor designs are heroic attempts to get mainframe performance from microcomputers. The Firefly is different: it is a straightforward and economical design. This paper describes the essentials of the Firefly's architecture and implementation. Readers interested in the fundamentals of multiprocessor design will find this paper—uncluttered by heroics—a valuable addition to the literature.

Neil C. Wilhelm

Contents

1 Introduction 1
2 Requirements 1
3 Compromises 1
4 Software 3
4.1 Topaz Structure 3
4.2 Programming Language and Environment 4
5 Hardware Description 5
5.1 Memory System Details 6
5.2 Hardware Performance Estimate 9
5.3 Hardware Performance Measurements 11
6 Conclusions 12
7 Acknowledgements 13
8 References 15
9 Index 17

List of Figures:

Figure 1: Firefly System 2
Figure 2: Internal Structure of Topaz 4
Figure 3: Cache Line States 8
Figure 4: MBus Timing 9

List of Tables:

Table 1: Firefly Estimated Performance 11
Table 2: Firefly Measured Performance (K refs/sec) 12

1. Introduction

Building the Firefly was the first hardware development project at SRC. It addressed the need for a personal workstation that would provide the computing engine for SRC's research in programming technology and multiprocessing, within the context of distributed personal computing. The Firefly therefore had to fulfill 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.[1] This report describes the goals, architecture, implementation and performance analysis of the Firefly. It then presents some measurements of hardware performance, and concludes with some brief remarks on the evolution of the software.

2. Requirements

With the availability of inexpensive, single-chip processors with the power of super-minicomputers, multiprocessors have become less difficult to design and build than uniprocessors with similar raw performance. A system composed of ten one-MIPS processors is a much less formidable engineering effort than a ten-MIPS uniprocessor and its associated memory system. It seemed to us that if we wanted a powerful workstation quickly, it would have to be a multiprocessor.

Hardware alone does not make a useful system. However, by using only medium and coarse-grained parallelism, we were confident that we could provide software to make effective use of a system with a small number of processors. At the coarsest level, workstation users like to keep several activities running at once -- profiling an application while compiling a module while reading mail. Pipelined execution is another form of coarse-grained concurrency. Experienced Ultrix users, for example, often use pipelines of applications such as the text processing utilities awk, grep, and sed. At the next level, we knew that some important applications could be modified to take advantage of parallelism and that a number of operating system support activities could proceed in parallel with applications. Finally, and more speculatively, the machine would provide a proving ground for applications making use of fine-grained parallelism.

This thinking led us to the Firefly (Figure 1), which in its original version consisted of a number of MicroVAX 78032 [3] processors sharing a single primary memory. While the individual processors had performance comparable with that of a VAX 11/780, the system as a whole was more like a VAX 8600. A more recent version of the system makes use of CVAX 78034 [2] processors. It provides approximately 2.5 times the performance and eight times as much memory as the original version.

3. Compromises

Fulfilling our three requirements caused us to make a number of design choices and compromises, particularly in cache structure, I/O architecture, and memory capacity. For ease of programming, we wanted to make the Firefly as symmetrical as we could. Our desire to use the Firefly as a workstation meant that the machine had to operate in the user's office. That in turn meant that the Firefly could not be too large, or too loud, or draw too much power.

These limitations on physical size, combined with our short schedule, meant that we should design as few board types as we could. In particular, we wanted to fit two processors on an 8 by 10 inch board. We did not want to design high-density custom or semi-custom chips, and this fact, coupled with the real-estate restriction, dictated that we use a small, direct-mapped cache with a simple memory bus protocol, on a relatively low performance bus.

---

[1] It is better to have a working computer than a long running development project. As Wes Clark once said, "Sometimes it is better to have twenty million instructions by Friday than twenty million instructions per second."

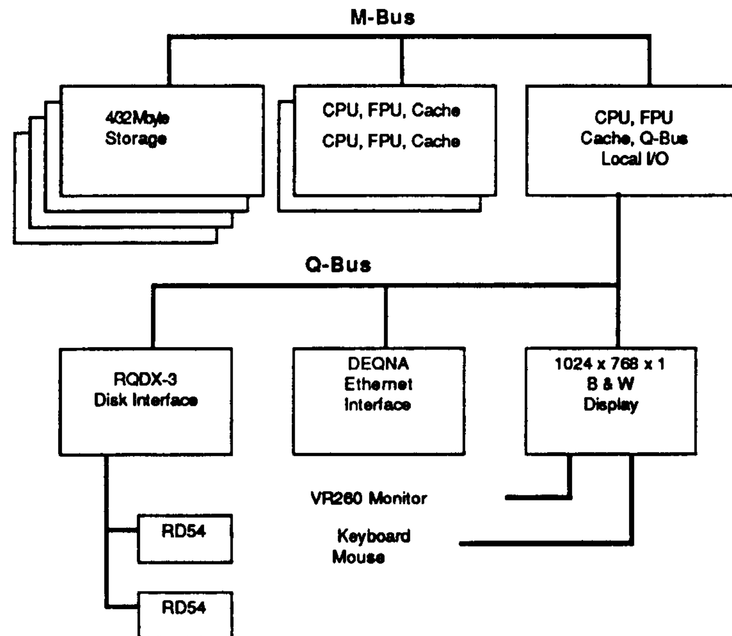


Figure 1: Firefly System

A small, direct-mapped cache is not usually consistent with high performance. The Firefly cache is unusual in that it is not employed to reduce the average access time to main memory. Rather, it is used to reduce the per-processor load on the main storage system. This reduction in bus loading allows us to attach several processors.

The fact that we wanted the machine finished quickly dictated that we use an existing I/O system and as many standard parts and subassemblies as we could. The availability in the spring of 1985 of the DEC MicroVAX CPU chip and associated computer system gave us a considerable boost, as we were able to borrow the entire I/O system. This shortened our development time, but left us with an asymmetric I/O arrangement in which only one of the processors had direct access to the QBus. This might seem to be a major obstacle to programming, but there is no difficulty with an asymmetric hardware implementation, provided that the *abstraction* presented by the I/O system is symmetric. We have achieved this symmetry for all devices except the disks.

The Firefly has three main I/O device types: disk, network, and display. For the disk and network interfaces, we chose to use standard DEC devices, a buffered controller for rigid and floppy disks (RQDX3), and an Ethernet controller (DEQNA). Both controllers are direct memory access (DMA) devices, and do data transfers directly to Firefly memory through the I/O processor's cache. The 22-bit address space of the QBus is mapped into the 24-bit space of the Firefly by mapping registers that are controlled by the IO processor.

Since no high-performance display controllers were available at the time, we decided to design our own. Both the monochrome controller (MDC) and color controller operate by periodically polling a work queue in main memory using DMA. This design provides fully symmetric access to the displays by any processor. The disk and network controllers are more traditional DMA designs -- at least a few programmed I/O instructions must execute on the I/O processor to initiate a transfer. For these devices, only the software interface to the device is fully symmetric[2].

[2]In fact, for the network device driver the abstraction is very similar to the implementation. Any processor can enqueue work for the network and then initiate the transfer by a specialized interprocessor interrupt to the I/O processor. The few instructions necessary to start the network controller are coded directly in the I/O processor's interprocessor interrupt service routine. We have not found it necessary to code the disk device driver similarly. Since the disk is buffered from applications by a large read cache and a large write buffer, only a small performance improvement would be gained by optimizing the actual initiation of disk transfers.

The use of existing semicustom parts in the I/O system and limits placed on the size of the cache data paths by the available board area led to a problem potentially more serious than that resulting from asymmetric I/O. The Firefly was limited to 16 megabytes of physical memory. Minicomputer physical memories have been doubling in size approximately every two years, with 16 to 18 bits of address being common in 1970 [10, p. 7]. By this measure, the 1986 Firefly should have had 64 megabytes of physical memory. The severity of this problem was realized shortly after the Firefly was placed into service. The second version of the system provides for up to 128 megabytes of physical memory, but at some sacrifice in the symmetry of the I/O abstraction.

4. Software

The software system for the Firefly is called Topaz. The operating system component of Topaz is called Taos. 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.

4.1. Topaz Structure

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. In particular, multiple threads can coexist in a single Topaz address space.

Inter-address-space and inter-machine communications in Topaz are handled by remote procedure calls. Since procedure call semantics are very familiar to programmers, using this familiar paradigm permits the construction of complex distributed applications by people without extensive communications experience. RPC, together with inexpensive Threads, permits all I/O and communications services to have synchronous interfaces. If asynchronous behavior is desired, one simply forks a new Thread to make the synchronous call.

The internal structure of Topaz is shown in Figure 2. The Nub runs in VAX kernel mode and provides virtual memory, thread scheduling, simple device drivers, and the transport mechanism for inter-address-space RPC. All other software, including the operating system, a debugging server, and the window manager, runs in user mode in various address spaces.

There are two kinds of application address spaces. Ultrix address spaces provide an environment in which most MicroVAX Ultrix binaries can run unchanged[3]. An Ultrix address space can support only one thread of control. A Topaz address space provides the same facilities, but in addition, permits multiple threads of control to execute simultaneously. Topaz applications use an operating system interface that communicates with Taos through RPC and permits concurrent access to the OS facilities. There are no particular limits to either the number of address spaces or the number of threads, but we expect the Firefly typically to operate with tens of address spaces and hundreds of threads.

The debugging address space is known as UserTTD (Topaz Tele-Debug). A debugger on the same or a different machine communicates with UserTTD to examine and debug user address spaces, including Taos. There is a

[3]Because of the pervasive use of RPC, we have not implemented the Ultrix socket abstraction.

smaller version of this debugging server, called NubTTD, that is used to debug the Nub.

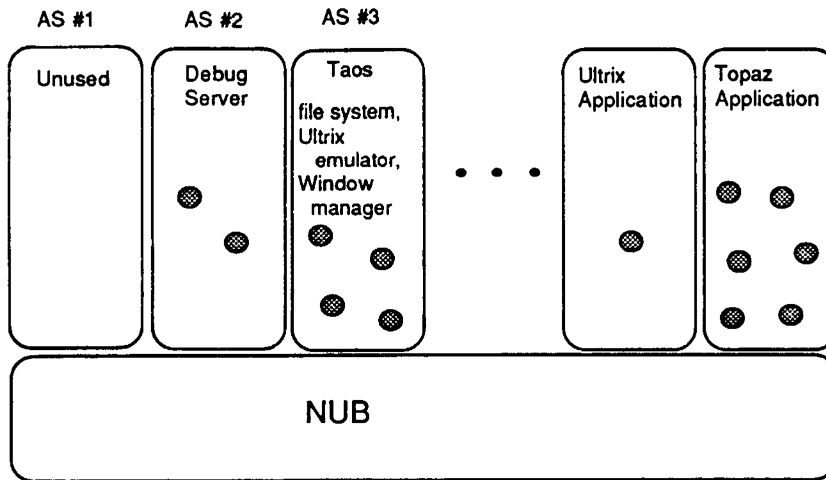


Figure 2: Internal Structure of Topaz

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 calls.

4.2. Programming Language and Environment

Both Topaz and new applications software are largely written in Modula-2+. Modula-2+ [9] is a version of Modula-2 [12] augmented by language support for garbage collected storage, exception handling, and concurrency.

The changes to the language for garbage collected storage include the addition of REF types. REFs are similar to POINTERs, except that the compiler and the runtime system keep track of the number of extant copies of a REF. When this number becomes zero, the referent is safely and automatically deallocated. The reference counts are kept in the objects themselves. Assignments to parameters and local variables on the stack are not reference counted. Only REFs in heap storage are counted. REFs on the stack are identified by a conservative scan. The collector runs concurrently with the application, which reduces the execution time penalty of collection to the main line computation. A separate trace and sweep collector handles the reclamation of circular or self-referential structures.

Modula-2+ provides facilities for finalization and exception handling. Finalization is provided through a TRY ... FINALLY block. Statements within the TRY block are executed, and regardless of the detailed flow of control, the finalization code in the FINALLY block is executed before control leaves the scope of the overall block. Exceptions are provided by a RAISE built-in procedure and a TRY ... EXCEPT block. The EXCEPT block provides a place to declare exception handlers. Modula-2+ exceptions are non-local gotos with dynamic binding, and have termination semantics. When an exception is raised, the call stack is searched for the nearest handler. Before control passes to the handler, the call stack is unwound. If there is no handler, control passes to the debugger, which has access to the complete state before stack unwinding.

Modula-2+ facilities for concurrency are provided by a combination of the compiler and the runtime system 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 is applied to a DO END block. LOCK

acquires a Mutex, or mutual exclusion variable, for the duration of the block. Essentially, the LOCK statement declares a critical section.

Strong typing and the separation of interface and implementation in Modula-2 have made possible a large and growing collection of software packages that provide a wide range of runtime services. The well-defined abstractions, or interfaces, provided by packages make it easier for a large number of programmers to work together on large systems.

5. Hardware Description

Firefly processors communicate with primary memory through individual cache memories and a dedicated bus, the MBus. Each cache is direct mapped, and in the original version of the system, contained 4096 four-byte lines. To run at full speed, the MicroVAX processors in the Firefly required "no-wait-state" memory with a 400 ns cycle time. Each processor averaged about 400,000 VAX instructions per second and kept its local memory interface busy about 40% of the time.

The original main memory system was packaged as one master four-megabyte module, and up to three slave modules of the same size. The MBus supports one four-byte transfer every 400 ns, for an aggregate bandwidth of 10 megabytes per second. The MBus also provides facilities for system initialization and interprocessor interrupts. It is carried between modules on flat-ribbon cable, so that a standard backplane can be used.

Each Firefly system consists of a single primary processor board, plus a number of secondary processor boards. The primary board contains a MicroVAX CPU chip with its companion floating point unit and cache, plus the logic necessary to control the QBus, which is used only for I/O. This board makes use of a number of semicustom components originally developed for the MicroVAX II system. The secondary processor boards in the original Firefly contained two identical MicroVAX CPUs, their associated FPUs, and caches. The second version of the Firefly uses faster processors and allows a larger main memory. The new secondary processor boards use a pair of CVAX 78034 processors, matching floating point units, and an upgraded cache. The cache is again direct mapped, with 16384 four-byte lines, and is fast enough so that memory cycles that hit in the cache complete in 200 ns with no wait states. The CVAX processor itself includes a 1024 byte on-chip cache. To simplify the problem of maintaining memory coherence, we have chosen to configure that cache to store only instruction references, not data.

Each new memory module provides 32 megabytes of storage, for a maximum of 128 megabytes of physical memory per Firefly system. The new secondary processor boards can address all of this memory, but we have no current plans to upgrade the primary processor boards. Thus the CPU serving as the I/O processor and the DMA devices can access only the first 16 megabytes of physical memory. This restriction further increases the asymmetry of the I/O system at the hardware level, but most higher level interfaces can still present a satisfactory abstraction to clients.

The Firefly disk, ethernet, and display controllers connect to a DEC QBus. DMA references to main memory are made through the I/O processor's cache (although DMA misses do not allocate). When fully loaded, the QBus consumes about 30% of the main memory bandwidth. The average I/O load is much lower.

The monochrome display controller (MDC) is packaged on a board half as large as the processor boards. It contains a 10 MHz 29116 16-bit microprocessor with 2048 40-bit words of microinstruction memory and a one-megapixel frame buffer constructed with video RAMs. Three-quarters of the frame buffer holds the display bitmap, while the rest is available to the display manager.

The MDC periodically polls a work queue kept in Firefly main memory, and executes commands from the queue. Commands are provided to do BitBlt [5] 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 cache 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 10-point characters per second. The MDC also supports a keyboard and mouse. Sixty times per second, the controller deposits in Firefly memory the current mouse position and an unencoded bitmap representing the current state of the keyboard.

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. Because 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 I/O 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 real-estate valuable.

5.1. 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 is 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 only four bytes, it cannot take advantage of spatial locality. This is not as harmful as it might seem, since misses add only one cycle to a MicroVAX CPU access. The 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-through* with invalidation, in which all writes are sent to the main memory bus. Whenever a cache observes a write directed to a line it contains, it invalidates its copy. 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. 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 usually 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. When a cache acquires ownership of a line, other caches invalidate their contents. Berkeley Ownership [7] 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 datum simultaneously, and no prearrangement is required for a processor to write a shared location. The Xerox Dragon [8] uses a similar scheme. The key idea in this protocol is that a cache can detect when another cache shares a particular location. For non-shared lines, a write-back strategy is used. Processor reads and writes 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 two tag bits for each cache line: Dirty and Shared. Dirty indicates that the cached copy of a datum has been modified with respect to main memory, and must therefore be written back when the line is victimized. Shared 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 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 Figure 3. When a processor operation causes an MBus operation, the transition is determined by the response that the initiator receives from the other caches in the system on the MShared line. In the figure, this response is shown in parentheses.

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. When the read is done, the Shared tag is set to the value of MShared returned by other caches. If the line is dirty, its contents must be written back to main memory before the read is done.

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 and shared. A write miss is treated as a read miss followed immediately by a write hit.

In the VAX, most writes are to aligned (32-bit) longwords. Since Firefly cache lines are only one longword in size, longword write misses can be optimized. Instead of doing a read, then overwriting the line with write data, the cache simply does write-through, leaving the line clean. The state of the shared tag 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 tag 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 is displaced by

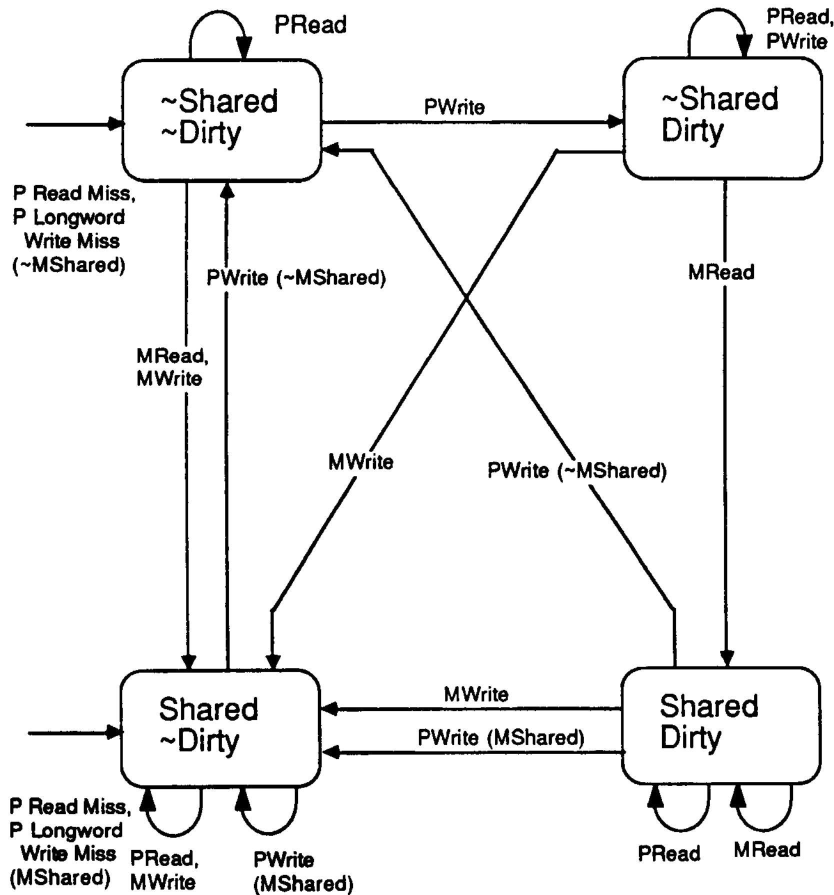


Figure 3: Cache Line States

the activity of another process. For this reason, the Topaz 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 the CPU next references it.

The timing of MBus operations is shown in Figure 4. There are only two operations, MRead and MWrite. Each 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 is supplied. If no cache asserted MShared during cycle 3, the data comes 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 read data, but since the protocol ensures coherence, the values will be identical.

Although the description would seem to imply a complex implementation, surprisingly little logic is needed. The cache data paths in the original dual-processor board consist of eleven 4K X 4 static RAMs and about twenty other TTL components. The control logic is implemented in about 15 parts, mostly PALs. The complexity of the CVAX version of this board is similar, except that 16K x 4 static RAMs are used in the caches.

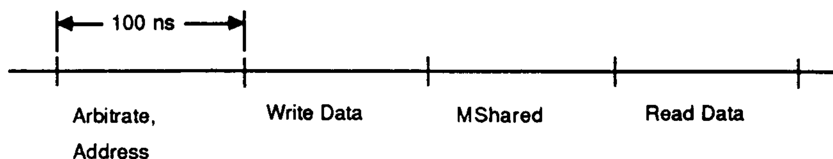


Figure 4: MBus Timing

5.2. 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 [11]. Our analysis uses simulation to obtain the characteristics of a single processor and its cache, then uses a simple queuing model to predict the performance of the full system as a function of the bus load (i.e., the fraction of non-idle bus cycles). We then calculate the number of processors required to achieve the given load. The overall system performance is then simply the product of the performance of a single processor and the number of processors. This is a back-of-the-envelope approach, but for our purposes, slide-rule accuracy is more than adequate.

The analysis presented here 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 board, 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 use the 10 Mbyte/second 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 [4] show that a typical instruction does .95 (=IR) instruction reads per instruction, .78 (=DR) data reads, and .40 (=DW) data writes, for a total of 2.13 (=TR) references per instruction. This is an architectural property valid across a wide range of applications, and does not depend on the particular CPU implementation.

Trace-driven simulation of the MicroVAX CPU, carried out for us by Deborrah Zukowski of the DEC Eastern Research Laboratory, showed it to be an 11.9 tick-per-instruction (TPI) implementation of the VAX architecture when operating with a memory that introduces no wait states. In the Firefly, ticks are 200 ns. each. These simulations also showed that a single processor Firefly cache achieves a miss rate M of 0.2[4], and that the fraction D 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.

Each MBus operation in the Firefly requires N=2 ticks. If we model the bus and storage as an open queueing network, the number of ticks spent waiting for the bus and doing the operation is N/(1-L) ticks, where L is the bus load. This is not accurate at high loads, since the number of caches requesting service is bounded, but it is fairly accurate at the moderate loads at which the system actually operates.

---

[4]This is an abnormally large miss rate for a 16 kilobyte 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. Since the penalty for a miss is only one tick if the MBus is available, while the penalty for delaying the project or making the system larger would be much greater, we did not pursue a larger line.

We are also ignoring 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: Each miss requires one MBus read to service the miss, plus D (the fraction of dirty entries in the cache) MBus victim writes. The number of added ticks per instruction is therefore:

```
SM = TR (refs per instruction)
   * M (misses per ref)
   * (1+D) (MBus ops per miss)
   * N/(1-L) (ticks per M op)

   = 1.065/(1-L)
```

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

```
SW = DW * S * N/(1-L)

   = .08/(1-L)
```

- Tag store probes by other caches: Each CPU cache access that hits will be slowed by one tick if an MBus operation 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:

```
SP = TR * (1-M) * 1/N * L
   = .85L
```

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

```
TPI = 11.9 + SM + SW + SP
    = 11.9 + 1.145/(1-L) + .85L
```

The relative performance of a single processor, RP, is simply 11.9/TPI.

We can determine the number of processors, NP, needed to obtain a given load by dividing the total number of MBus operations per tick at that load, L/N, by the number of MBus operations per tick used by a single processor. The number of MBus operations used per tick is the product of the number of MBus operations per instruction, and the number of instructions per tick, 1/TPI.

```
NP = (L/N) /    (MBus ops per tick)
   / ((1/TPI)   (Instructions per tick)
   * (M*TR*(1+D)  (M ops per instruction
                   due to misses)
   +  DW*S)       (M ops per instruction
                   due to write-through)
   )

   = L*TPI / 1.145
```

The total system performance TP, relative to a single processor with no-wait-state memory is just RP * NP.

```
TP = 11.9 * NP / TPI
```

These equations can be manipulated algebraically to find L, TPI, RP, and TP as a function of NP. The results of these calculations are summarized in Table 1. It is clear that the Firefly MBus can support perhaps nine processors before the marginal improvement achieved by adding another processor becomes unattractive. The standard five-

processor configuration delivers somewhat more than four times the performance of a single processor with no-wait-state memory. The average bus load on the standard machine is 0.4 and each processor runs at about 85% of a no-wait-state system.

The design of the Firefly memory system was heavily influenced by pragmatic considerations: we implemented what would fit, using available components. Nevertheless, the system is fairly well balanced. The cache hit rates are low, but misses are not very expensive. If the Firefly processors were significantly faster relative to main memory, then it would be necessary to push down the miss rate either by increasing the cache size or by increasing the cache block size. In the CVAX version of the system, we chose to quadruple the cache size.

Table 1: Firefly Estimated Performance

NP (number of processors):	2	4	6	8	10	12
L (bus loading):	.17	.33	.47	.60	.70	.78
TPI (ticks per instruction):	13.4	13.9	14.5	15.3	16.3	17.7
RP (relative performance) :	.89	.85	.82	.78	.72	.67
TP (total performance):	1.77	3.43	4.93	6.23	7.29	8.07

5.3. Hardware Performance Measurements

Measurements of the MicroVAX Firefly show that the hardware performs about as well as expected. Table 2 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 Topaz Threads package. The program forks a number of threads, each of which then executes and checks the results of Threads package primitives. 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 chip 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/sec (10.5 TPI) and the reference rate would be 1014K references/sec. 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 non-prefetch references.

Our measurement method can distinguish three categories of MBus write: Non-victim writes that receive MShared from other caches, non-victim writes that do not receive MShared, and victim writes. For this program, the estimate of 10% sharing is clearly too low, since 75K of the 225K writes done by one CPU (33%) were write-throughs that received MShared. The difference between the estimated and actual sharing is not too surpris-

ing, 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.

Table 2: Firefly Measured Performance (K refs/sec)

	One-CPU system Expected	One-CPU system Actual	Five-CPU system Expected	Five-CPU system Actual
Per CPU:				
Reads	688	1125	609	850
Writes	161	240	143	225
Total	849	1350	752	1075

	One-CPU system	Five-CPU system
Actual MBus Total References:	440 (L=.18)	1350 (L=.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

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 to 2.5. This is less than the 2.5 to 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.

6. Conclusions

The Firefly hardware is now complete, and five-processor systems are being used as workstations by all members of the SRC staff. We have built a few seven-processor systems for experimentation, but these are not in everyday use. 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. We did, however, make several serious errors 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 board was simulated, but timing analysis was again done by hand and several minor errors were not detected until the board was built. The second major problem was that several 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 our design.

The Firefly software is still evolving rapidly. We have had considerable success in utilizing the coarse-grained parallelism that the multiprocessor provides, and Topaz and a few applications have been able to exploit medium-grained parallelism in useful ways. 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.

Standard applications also benefit from multiprocessing. The file system uses multiple threads to do read-ahead and write-behind, and the window manager does much of its work in parallel with the user's computation. Single threaded applications that use garbage collection also benefit. The application must pay the in-line cost of reference counted assignments, but the collector itself runs as a separate thread on another processor.

The use of parallelism at the lowest levels of the system helps to compensate for the fact that Ultrix system calls are emulated, and are therefore somewhat slower in Topaz than they would have been had we simply ported Ultrix[5].

New software will obtain the greatest benefit from multiprocessing. For example, we have implemented a parallel version of the Unix *make* utility, which forks multiple compilations in parallel when possible. An experimental version of the Modula-2+ compiler quickly reads in the source file and then compiles each procedure body in parallel. Another example is the implementation of remote servers. We have found that our RPC data transfer protocol, with multiple outstanding calls, achieves very high performance. The remote server can sustain a bandwidth of 4.6 megabits per second using an average of three concurrent threads.

Advantage is sometimes measured in years rather than MIPS. Bill Joy asserts [6] that the performance of integrated CPUs is doubling each year. An eight processor system therefore holds a three year advantage over a uniprocessor built with similar technology. Snoopy caches permit the construction of multiprocessor computer systems with modest numbers of processors without requiring a memory system much more complex or much faster than that required for a uniprocessor. We are working towards system and application software to make effective use of such a machine. Once software can use the multiprocessor well, that three year advantage can be held indefinitely by building multiprocessors with the fastest available components.

7. Acknowledgements

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. John Dillon designed the new storage boards, and Bob McNamara, Carol Peters, and Phil Petit made substantial contributions to the hardware. Paul Rovner, Violetta Cavalli-Sforza, and Jim Horning built the Modula-2+ system, starting from Mike Powell's Modula-2 compiler. Roy Levin, Mike Schroeder, Garret Swart, Paul McJones, Andrew Birrell, and Butler Lampson developed much of Topaz. The Trestle window manager was designed and built by Greg Nelson, Mark Manasse and Mark Brown. Eric Roberts implemented the parallel version of *make*.

[5]Most of the speed difference in simple system calls is due to the context switch necessary because Taos runs as a user mode address space. Longer-running system services do not suffer as much from this effect. It is also true that our software has not had the benefit of years of careful tuning.

References

[1] James Archibald and Jean-Loup Baer.
Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model.
ACM Transactions on Computer Systems 4(4):273-298, November, 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 Journal of Solid State Circuits SC-22(5): 849-852, October, 1987.

[3] D.W. Dobberpuhl, R.M. Supnik, and R.T. Witek.
The MicroVAX 78032 Chip, a 32 Bit Microprocessor.
Digital Technical Journal , March, 1986.

[4] J.S. Emer and D.W. Clark.
A Characterization of Processor Performance in the VAX-11/780.
In *Proceedings of the 10th Annual Symposium on Computer Architecture*, pages 301-310. IEEE, June, 1984.

[5] Dan Ingalls.
The Smalltalk Graphics Kernel.
Byte 6(8):168-194, August, 1981.

[6] William Joy.
Computer Workstation Architecture: 1982-1992.
In *Proceedings of the Conference on New Frontiers in Computer Architecture.* Citicorp/TTI, 3100 Ocean Park Blvd., Santa Monica, California, 90405, March, 1986.

[7] R.H. Katz, S.J. Eggers, D.A. Wood, C.L. Pekins, and R.G. Sheldon.
Implementing a Cache Consistency Protocol.
In *Proceedings of the 12th International Symposium on Computer Architecture.* IEEE, 1985.

[8] E.M. McCreight.
The Dragon Computer System, an Early Overview.
In *NATO Advanced Study Institute on Microarchitecture of VLSI Computers.* July, 1984.

[9] Paul Rovner, Roy Levin, and John Wick.
On Extending Modula-2 For Building Large, Integrated Systems.
Technical Report 3, DEC Systems Research Center, January, 1985.

[10] Daniel P. Siewiorek, C. Gordon Bell, and Allen Newell.
Computer Structures, Principles and Examples.
McGraw-Hill, 1982.

[11] A.J. Smith.
Cache Memories.
ACM Computing Surveys 14(3): 473-530, September, 1982.

[12] Niklaus Wirth.
Programming in Modula-2.
Springer-Verlag, third edition 1985.

Index

address spaces
- Topaz 3
- Ultrix 3

Archibald, James 6

Baer, Jean-Loup 6
Berkeley Ownership 7
BitBlt display primitive 5, 6

cache (see also, performance) 1, 2, 5, 6, 7, 8
- on-chip ~ 5, 9, 12
- parameters in Firefly 12
- role of ~, in Firefly 6
- snoopy ~ 6, 13
- state transitions 7

color controller 2
concurrency 1, 4, 13
- facilities for, in Modula-2+ 4

conditional write-through 7
CVAX 78034 processor 1, 5, 12

debugging address space
- See instead: UserTTD

Dirty tag, introduced 7

exception handling, in Modula-2+ 4

finalization, in Modula-2+ 4

garbage collection in Modula-2+ 4

I/O system 3, 6
- abstraction in 3
- symmetry in 3, 5

make utility 13
MBus 5, 7, 9, 10
- timing of ~ operations 8

memory system 5, 6, 7, 8
MicroVAX 78032 processor 1, 2, 5, 6, 9
Modula-2 4
Modula-2+ 4, 13
monochrome display controller (MDC) 2, 5, 6
MShared signal, introduced 7

Nub 3
NubTTD 3

performance, of hardware
- estimates 9, 10, 11
- measurements 11, 12

pipelined execution 1
POINTER 4

QBus 2, 5

REF type
- compared with POINTER 4

remote procedure call 3

Shared tag 7
snoopy cache (see instead, cache) 6

Taos 3
Topaz 3, 4, 11, 13
- introduced 3
- notion of Thread in 3
- scheduler 7

Trestle 3

UserTTD (Topaz Tele-Debug) 3

write-back protocol 6
write-through protocol 6, 10

Xerox Dragon 7

Zukowski, Deborrah 9

SRC Reports

"A Kernel Language for Modules and Abstract Data Types."
R. Burstall and B. Lampson.
Research Report 1, September 1, 1984.

"Optimal Point Location in a Monotone Subdivision."
Herbert Edelsbrunner, Leo J. Guibas, and Jorge Stolfi.
Research Report 2, October 25, 1984.

"On Extending Modula-2 for Building Large, Integrated Systems."
Paul Rovner, Roy Levin, John Wick.
Research Report 3, January 11, 1985.

"Eliminating **go to**'s while Preserving Program Structure."
Lyle Ramshaw.
Research Report 4, July 15, 1985.

"Larch in Five Easy Pieces."
J. V. Guttag, J. J. Horning, and J. M. Wing.
Research Report 5, July 24, 1985.

"A Caching File System for a Programmer's Workstation."
Michael D. Schroeder, David K. Gifford, and Roger M. Needham.
Research Report 6, October 19, 1985.

"A Fast Mutual Exclusion Algorithm."
Leslie Lamport.
Research Report 7, November 14, 1985.

"On Interprocess Communication."
Leslie Lamport.
Research Report 8, December 25, 1985.

"Topologically Sweeping an Arrangement."
Herbert Edelsbrunner and Leonidas J. Guibas.
Research Report 9, April 1, 1986.

"A Polymorphic λ-calculus with Type:Type."
Luca Cardelli.
Research Report 10, May 1, 1986.

"Control Predicates Are Better Than Dummy Variables For Reasoning About Program Control."
Leslie Lamport.
Research Report 11, May 5, 1986.

"Fractional Cascading."
Bernard Chazelle and Leonidas J. Guibas.
Research Report 12, June 23, 1986.

"Retiming Synchronous Circuitry."
Charles E. Leiserson and James B. Saxe.
Research Report 13, August 20, 1986.

"An $O(n^2)$ Shortest Path Algorithm for a Non-Rotating Convex Body."
John Hershberger and Leonidas J. Guibas.
Research Report 14, November 27, 1986.

"A Simple Approach to Specifying Concurrent Systems."
Leslie Lamport.
Research Report 15, December 25, 1986.

"A Generalization of Dijkstra's Calculus."
Greg Nelson.
Research Report 16, April 2, 1987.

"*win* and *sin*: Predicate Transformers for Concurrency."
Leslie Lamport.
Research Report 17, May 1, 1987.

"Synchronizing Time Servers."
Leslie Lamport.
Research Report 18, June 1, 1987.

"Blossoming: A Connect-the-Dots Approach to Splines."
Lyle Ramshaw.
Research Report 19, June 21, 1987.

"Synchronization Primitives for a Multiprocessor: A Formal Specification."
A. D. Birrell, J. V. Guttag, J. J. Horning, R. Levin.
Research Report 20, August 20, 1987.

"Evolving the UNIX System Interface to Support Multithreaded Programs."
Paul R. McJones and Garret F. Swart.
Research Report 21, September 28, 1987.

"Building User Interfaces by Direct Manipulation."
Luca Cardelli.
Research Report 22, October 2, 1987.

by Charles P. Thacker, Lawrence C. Stewart, and Edwin H. Satterthwaite Jr.

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301