Proceedings of the First Workshop on Experiences with Building Distributed and Multiprocessor Systems, Ft. Lauderdale, FL, October 1989, pp 227-236.

Implementation Issues for the Psyche Multiprocessor Operating System

Michael L. Scott
Thomas J. LeBlanc
Brian D. Marsh

University of Rochester
Department of Computer Science
Rochester, NY 14627
{scott,leblanc,marsh}@cs.rochester.edu

ABSTRACT

Psyche is a parallel operating system under development at the University of Rochester. The Psyche user interface is designed to allow programs with widely differing concepts of process, sharing, protection, and communication to run efficiently on the same machine, and to interact in meaningful ways. In addition, the Psyche implementation effort is addressing a host of systems issues for large-scale shared-memory multiprocessors, including remote access to kernel data structures; the organization of kernel address maps; the design of appropriate synchronization and scheduling mechanisms; user-level device drivers, loaders, and pagers; remote source-level kernel debugging; rapid turn-around for the kernel debugging cycle; resource reservation mechanisms for real-time applications; and page migration and replication to maximize locality.

1. Introduction

Parallel processing is in the midst of a transition from special purpose to general purpose use. Part of the impetus for this transition has been the development of practical, large-scale, shared-memory multiprocessors. To make the most effective use of these machines, an operating system must address two fundamental issues that do not arise on uniprocessors. First, the kernel interface must provide the user with greater control over parallel processing abstractions than is customary in a traditional operating system. Second, the kernel must be structured to take advantage of the parallelism and sharing available in the hardware.

If shared-memory multiprocessors are to be used for day-to-day computing, it is important that users be able to program them with whatever style of parallelism is most appropriate for each particular application. To do so they must be able to exercise control over concepts traditionally reserved to the kernel of the operating system, including processes, communication, scheduling, protection, and the grain size of memory sharing. If shared-memory multiprocessors are to be used efficiently, it is also important that the kernel not define abstractions that hide a significant portion of the hardware's functionality.

This work was supported in part by NSF CER grant number DCR-8320136, DARPA ETL contract number DACA76-85-C-0001, ONR Contract number N00014-87-K-0548, and an IBM Faculty Development Award.

The Psyche project is an attempt to design and prototype a high-performance, general-purpose operating system for large-scale shared-memory multiprocessors. The fundamental kernel abstraction, an abstract data object called a *realm*, can be used to implement such diverse mechanisms as monitors, remote procedure calls, buffered message passing, and unconstrained shared memory. Sharing is the default in Psyche; protection is provided only when the user specifically indicates a willingness to sacrifice performance in order to obtain it. Sharing also occurs between the user and the kernel, and helps to enable explicit, user-level control of process structure and scheduling.

Details of the Psyche kernel interface and its rationale have been presented elsewhere [5, 6]. The purpose of the current paper is to outline the kernel structuring issues that we are addressing in our prototype implementation. This is a work-in-progress paper; we have been writing code for about a year, and have recently completed the initial version of the kernel. We are not at a point where we can make definitive statements based on performance measurements or production-quality applications, but we can draw a few conclusions from our implementation experience and from previous projects [2, 3].

Psyche is intended to be portable to a wide range of shared-memory multiprocessors. Our initial implementation is written in C++ and runs on the BBN Butterfly Plus multiprocessor (the hardware base of the GP1000 product line). We have completed the major portions of the kernel and are experimenting with user-level software. In concert with members of the computer vision and planning groups within the department, we have undertaken a major integrated effort in the area of real-time active vision and robotics. The first "toy" program ran in user mode on Psyche in December of 1988. Our first robotics application is expected to be running soon.

Our robotics laboratory includes a custom binocular "head" on the end of a PUMA robot "neck." Images from the robot's "eyes" feed into a special-purpose pipelined image processor. Higher-level vision, planning, and robot control have been implemented on a uniprocessor Sun. Real-time response, however, will require extensive parallelization of these functions. The Butterfly implementation of Psyche provides the platform for this work. Effective implementation of the full range of robot functions will require several different models of parallelism, for which Psyche is ideally suited. In addition, practical experience in the vision lab will provide feedback on the Psyche design.

2. Synopsis of Psyche Abstractions

The Psyche programming model [6] is based on passive data abstractions called *realms*, which include both code and data. The code constitutes a protocol for manipulating the data and for scheduling threads of control. Invocation of protocol operations is the principal mechanism for accessing shared memory, thereby implementing interprocess communication.

Depending on the degree of protection desired, an invocation of a realm operation can be as fast as an ordinary procedure call, termed *optimized* invocation, or as safe as a remote procedure call between heavyweight processes, termed *protected* invocation. Unless the caller insists on protection (by performing an explicit kernel call), both forms of invocation are initiated by an ordinary jump-to-subroutine instruction. In the case of a protected invocation the instruction causes a page fault which allows the kernel to intervene.

To permit sharing of arbitrary realms at run time, Psyche arranges for all realms to reside in a uniform address space. The use of uniform addressing allows processes to share data structures and pointers without the need to translate between address spaces. Realms that are known to contain only private data can overlap, as can realms that are only accessed using protected invocations, so normal operating system workloads will fit within the Psyche address space.

At any moment in time, only a small portion of the Psyche uniform address space is accessible to a given process. Every Psyche process executes within a *protection domain*, an execution

environment that denotes the set of available rights. A protection domain's view of the Psyche address space, embodied by a hardware page table, contains those realms for which the right to perform optimized invocations has been demonstrated to the kernel. A process moves between protection domains, inheriting a new view of the address space and the corresponding set of rights, by performing protected invocations.

In order to execute processes inside a given protection domain, the user must ask the kernel to create a collection of *virtual processors* to be associated with that domain. The kernel keeps track of which processes have moved from one protection domain to another, but aside from this it deals only with virtual processors, leaving the job of process management to user-level code. Users, for their part, need not worry about virtual processors. Above the level of the kernel interface Psyche behaves as if there were one physical processor for each virtual processor. We refer to a virtual processor as an *activation* of a protection domain.

On each node of the physical machine, the kernel time-slices between activations currently located on its node. A data structure shared between the kernel and the user contains an indication of which process is being served by the current activation. This indication can be changed in user code, so it is entirely possible (in fact likely) that when execution enters the kernel the currently running process will be different from the one that was running when execution last returned to user space. The kernel's help is not required to create or destroy processes within a single protection domain, or to perform context switches between those processes.

Communication from the kernel to the activations takes the form of signals, or *upcalls*, that resemble software interrupts. Upcalls occur when a process moves to a new protection domain, when it returns, and whenever an error occurs. In addition, user-level code can establish upcall handlers for wall time and interval timers, and can arrange to receive a warning in advance of activation preemption.

3. Kernel Organization

3.1. Basic Kernel Structure

The Psyche kernel interface is designed to take maximum advantage of shared-memory architectures. Since we are interested in concepts that scale, we assume that Psyche will be implemented on NUMA (non-uniform memory access) machines. A NUMA host is modeled as a collection of *clusters*, each of which comprises processors and memories with identical locality characteristics. A Sequent or Encore machine consists of a single cluster. On a Butterfly, each node is a cluster unto itself. The proposed Encore Ultramax [7] would consist of non-trivial clusters.

Our most basic kernel design decisions have been adopted with an eye toward efficient use of very large NUMA machines.

(1) The kernel is *symmetric*. Each cluster contains a separate copy of the bulk of the kernel code, and each processor executes this code independently. Scheduling and memory-management data structures are allocated in the kernel on a per-cluster basis. Kernel functions are performed locally whenever possible. The only exceptions are interrupt handlers (which must be located where the interrupts occur) and some virtual memory daemons which consume fewer resources when run on a global basis.

(2) The kernel makes extensive use of shared memory to communicate between processors, both within and between clusters. Ready lists, for example, are manipulated remotely in order to implement protected invocations. The alternative, a message-passing scheme in which instances of the kernel would be asked to perform the manipulations themselves, was rejected as overly expensive. Most modifications to remote data structures can be performed asynchronously; the remote kernel will notice them the next time the data is read.

Synchronous inter-kernel interrupts are used for I/O, remote TLB invalidation, and insertion of high-priority processes in ready queues.

(3) The kernel operates in two separate but overlapping address spaces. Since each instance of the kernel must be able to interact with each other instance, scalability dictates that a large amount of address space be devoted to kernel data structures. Since the kernel also shares data structures with the user, the entire Psyche uniform address space must be visible to the kernel as well. No available machine provides enough virtual address space for both of these needs. We have therefore designed a two-address-space kernel organization (see figure 1). The code and data of the local kernel instance are mapped into the same locations in both address spaces, making switches between those spaces easy. The user/kernel address space also contains all of user space, and the kernel/kernel address space contains the data of every kernel instance. Local data appear at two different locations in the kernel/kernel space.

As in most modern O. S. implementations, little distinction is made between parallelism in user space and parallelism in the kernel. Kernel resources are represented by parallel-access data structures, not by active processes. An activation that traps into the kernel enters a privileged hardware state ("supervisor mode") and begins to execute trusted code, but continues to be the same active entity that it was in user space.

When executing in user space, the activations of separate protection domains must have separate page tables. The kernel is included in each of these page tables (accessible only in supervisor mode), so that there is in fact a separate user/kernel address space for each protection domain. A disadvantage of this scheme is that address space switches are required not only to access data in the kernel/kernel address space, but also in order for the kernel to examine user data in more than one protection domain (as for example, when invoking a protected realm operation). An alternative would be to provide a single, universal user/kernel address space used on

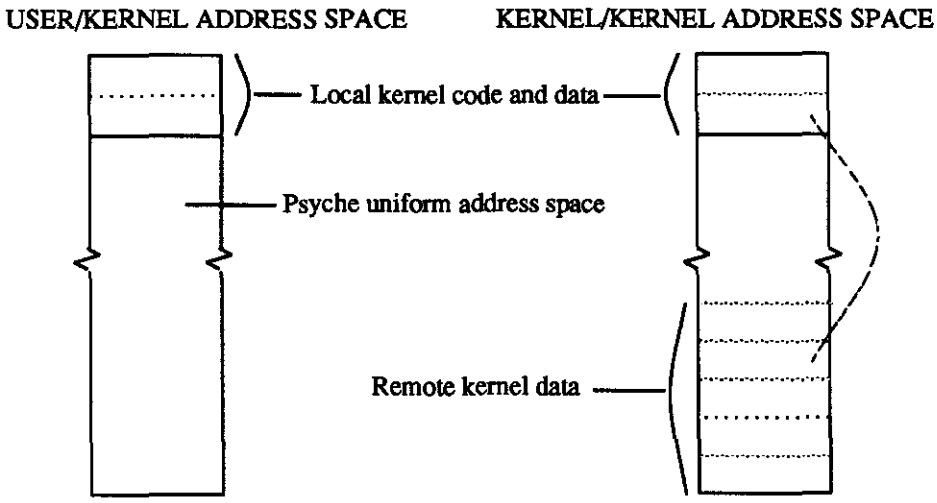

Figure 1: Kernel address spaces

kernel entry by every activation. It is not yet clear whether the resulting savings in address space switches would justify the additional cost of maintaining consistency with user-level page tables.

3.2. Synchronization

A traditional uniprocessor operating system often obtains mutual exclusion for kernel data structures by disabling preemption in the kernel. In a shared-memory multiprocessor, this simple technique no longer works. Data structures can be modified remotely. In fact, an early inventory of kernel data structures in Psyche revealed that almost none (other than those local to a subroutine) was private to a single processor. As a result, explicit synchronization is almost always required when accessing kernel data. We have therefore opted *not* to prohibit preemption in the kernel; there seems to be no point in doing so. The overhead incurred by explicit locking remains to be measured. We expect it to be significant, but our intuition is that it will still be less than the cost of message-passing between kernels to avoid the need for locking.

We have found a need in the kernel for four major types of synchronization. (We also have a facility for all-processor barrier synchronization, but this is used only for kernel initialization.)

disabled preemption
: Those few data structures that *are* processor-local (buffers for the per-processor console, for example) can be protected by disabling preemption. To allow nesting of locks, the kernel maintains a "preemption level" that is incremented when entering a critical section and decremented when leaving. At the end of a quantum, the clock handler forces a context switch only if the counter is zero. If the counter is positive, the handler sets a flag. The code that decrements the preemption level counter causes a context switch on behalf of the clock handler if the flag is set and the level has returned to zero.

locked-out interrupts
: Interrupt masking is used solely to synchronize with device handlers. Data structures shared with devices are never accessed remotely.

spin locks
: Spin locks are the most frequently-used locks in the kernel. There are separate EREW and CREW locks,[1] though the former are much more common. Spin locks are used only to protect critical sections of small, bounded length. The spin lock implementation disables preemption to ensure that the bound is not violated by an inopportune context switch.

scheduler locks
: For those situations in which an activation must wait for a condition that may not happen soon, we provide a simple mechanism to interact with the time-slicing scheduler. Every activation contains in its context block a flag that indicates whether its state has been "saved successfully." To block itself, an activation (1) disables preemption, (2) writes its name down where some other activation will find and resume it at an appropriate time, and (3) invokes the activation scheduler. The scheduler sets the flag of the old activation, clears the flag of the new activation, and re-enables preemption. Anyone who wants to resume an activation must spin until the state-saved flag is set. This mechanism suffices to implement semaphores or monitors and is also used by the clock handler to insert the current activation on the ready list and force preemption. The scheduler always assumes that preemption is disabled and that its caller has done something appropriate with the formerly-running activation.

---

[1] Exclusive read, exclusive write; concurrent read, exclusive write. EREW locks have lower overhead; they can be acquired and released more quickly than CREW locks in the absence of contention. CREW locks are appropriate when contention is high and reading is much more common than writing.

4. Resource Management

4.1. Devices

On the Butterfly Plus, Psyche supports three classes of devices:

(1) A pair of serial lines connects the "king node" of the Butterfly to a Unix host machine. One of these lines is used by Psyche as a console; the other is used for debugging (see section 5). Both of these uses are encapsulated in the kernel; serial line I/O is not meant to be exported to user-level programs.

(2) Our hardware also allows a Multibus cage to be connected to an individual node. The only Multibus device we support is an Ethernet interface. This is currently used to provide a simple remote file system (for development purposes) and Unix-style standard I/O.

(3) Non-network I/O travels over a VME bus. BBN hardware attaches the bus directly to the Butterfly communication switch, in place of one or two processor nodes. This connection method is superior to that of the Multibus both in terms of potential throughput and in its independence of any particular managing node. In our robotics lab the VME bus is used to communicate with the low-level image processor and the robot eye controllers.

Consistent with the Psyche philosophy of user-level flexibility and kernel minimality, we have developed an interface for memory-mapped I/O devices that limits the kernel's role to basic initialization and forwarding of interrupts. The `make_realm` system call allows the user (with appropriate access rights) to create a realm at a specified virtual or physical address. On the Butterfly, Multibus devices are accessed at special virtual addresses (decoded off the virtual address bus) and VME devices are accessed at the physical addresses corresponding to the VME adapter's location on the communication switch. By creating a memory-mapped realm, a user-level program obtains the ability to read and write device registers without the assistance of the kernel. A second kernel call allows the program to request that device interrupts be translated into upcalls into a user-level activation. We expect these upcalls to be generated with an acceptably small amount of overhead (though obviously more than a simple kernel-level interrupt handler). Again, the actual performance figures have yet to be obtained.

4.2. Virtual Memory

The largest and most sophisticated portion of the kernel is devoted to memory management [1], comprising four distinct abstraction layers. The lowest (NUMA) layer provides an encapsulation of physical page frames and tables. The second (UMA) layer provides the illusion of uniform memory access times through page replication and migration. The third (VUMA) layer provides a default pager for backing store and a mechanism for user-level pagers. The final (PUMA) layer implements Psyche protection domains and upcalls. Page faults may indicate events of interest to any of the layers; they percolate upward until handled.

The PUMA layer maintains a mapping that allows it to identify the realm that contains a given virtual address. This mapping is consulted when a page fault propagates to the PUMA layer, and allows the kernel to determine whether an attempt to touch an inaccessible realm constitutes an error, a protected invocation, or an initial use of something that should be mapped in for optimized access. The UMA layer is strictly divided between policy and mechanism. It is not yet clear how best to decide when to replicate and migrate pages, and this division facilitates experiments. There is no notion of location attached to a realm; the placement of its pages is under the complete control of UMA-layer policies. High-quality policies are likely to depend on the judicious use of hints from user-level software.

The Psyche external pager mechanism is similar in spirit to that provided in Mach [8], but with an interface based on shared memory instead of message passing. The `make_realm` kernel call allows the user to specify a pager activation capable of providing missing pages and

disposing of pages replaced by the kernel. Rather than send the pages in messages, as in Mach, Psyche simply provides the pager with optimized access to the data and code of the realm to be paged. Page-out and page-in requests are provided to the pager as upcalls. Page-out and page-in completion are indicated by the pager with kernel calls. Attempts by the pager to write into non-resident pages of the realm result in page faults that the kernel interprets as anticipatory page-in (pre-paging).

To bootstrap Psyche, the kernel creates a single primordial realm in a single protection domain, containing a single user-level process. This process executes code to create additional realms. Program loaders are outside the kernel, and may be integrated with external pagers. To run a user program, a shell (1) reads the header of the executable file to determine program size, (2) executes kernel calls to create an empty realm and one or more activations, (3) invokes a linker to relocate the executable into the virtual address of the newly-created realm, (4) copies the code and data into the realm, and (5) performs a protected invocation to start an activation running. More realistically, steps (3) and (4) can be replaced by communication with the default or user-provided pager to associate the realm with its executable file and relocation information. Pages can then be supplied on demand, and need not be written (with great amounts of unnecessary paging traffic) at start-up time.

4.3. Support for Real-Time Applications

Though the principal goal of Psyche is to support general-purpose parallel computing, we interpret this goal to include applications for which real-time support is important. We are interested in real-time computing as a research area, and are in the early stages of design work on a real-time subset of Psyche. We are well aware of the difficulties of adding real-time support to a pre-existing operating system, but have several mitigating factors on our side in our attempt to do this in Psyche. First, we are free to change the kernel or its interface when necessary. Second, we are able on a multiprocessor to segregate real-time and non-real-time portions of our workload onto different processors, where they can be managed with different policies. Third, we have in Psyche a kernel that is unusually small and easily adaptable to the needs of user programs.

The segregation of real-time and non-real time processes is facilitated by the already-existing mechanism for creating realms at specified physical locations. We can use this mechanism to dedicate the physical memory of a processor or cluster (without paging) to a particular application. An additional kernel call allows us to dedicate the computational resources of those processors to the activations of the application.

5. Experience with Tools

The Psyche kernel is written in C++ and compiled with the GNU (Free Software Foundation) g++ compiler. We have found the disciplined use of C++ abstractions to be useful in organizing our code. In addition, though this is difficult to quantify, we believe that it has helped to reduce the number of bugs in the code significantly. Unfortunately, the GNU compiler is still undergoing development, and has evolved considerably over the past year. It is unclear whether the advantages of C++ (versus C) have saved us as much time as we have lost to compiler bugs. As stable, high-quality C++ compilers become available, we expect to see them used more and more for operating system development.

One particularly useful language extension provided by the GNU compiler is the ability to redefine the built-in `new` operator and provide it with additional arguments (other than simply the size of the object desired). We have used this extension to provide multiple classes of dynamically-allocated memory. Separate allocators are used for (1) a processor-local heap, (2) the globally-accessible heap, (3) physically-contiguous, non-paged memory for page tables, and (4) physical "page zero" memory required by Butterfly microcoded atomic operations. For cases (2) and (3), an additional argument allows the programmer to specify a preferred node on

which to allocate the memory.

We have also defined a memory pseudo-class called "static" that can be used in conjunction with `new` to specify the time at which the constructor is called for a statically-allocated object. Since the kernel boots in an uninitialized environment (without page tables, interrupt vectors, etc.), we cannot in general allow these constructors to be called immediately upon startup. An additional user-provided argument specifies the virtual address of the object to be "allocated."

When power-cycled, the Butterfly Plus executes a serial-line loader in ROM. For many months we used this ROM directly to load Psyche at 9600 baud. As the kernel grew, this became increasingly painful. To speed the development cycle, we devised a small bootstrap program that initializes the Ethernet interface and then loads the bulk of the kernel using a naive (busy-wait) implementation of UDP. This bootstrap loader was surprisingly easy to write; we wish we had built it sooner.

To facilitate re-execution (for cyclic debugging, for example) we implemented a mechanism to restore the initial state of the kernel on demand. Immediately upon startup, we save a copy of the initialized data segment, and compute a checksum of the code. Upon receipt of a special character sequence, the console line interrupt handler restores the data, verifies that the code is uncorrupted, resets the hardware, and branches to the beginning of the kernel. The cost of this mechanism is small enough, both in complexity and space overhead, to recommend as a general practice for other kernel developers.

The most important tool we have constructed for Psyche is a mechanism for remote, source-level debugging, in the style of the Topaz TeleDebug facility developed at DEC SRC [4]. An interactive front end runs on a Sun workstation using the GNU gdb debugger. Gdb comes with a remote debugging facility; relatively minor modifications were required to get it to work with Psyche. The debugger communicates via UDP with a multiplexor running on the Butterfly's host machine. The multiplexor in turn communicates with a low-level debugging stub (lld) that underlies the Psyche kernel

The multiplexor allows many different debugging sessions to be underway simultaneously, each of them talking to a different Psyche node. It communicates with lld via one of the serial lines connected to the Butterfly king node. The interrupt handler for the debugging line accumulates input until it recognizes a special debugger packet termination character. It looks inside the packet to determine the node for which the packet is intended, and either wakes up the instance of lld on its own node or causes a remote interrupt to effect the same result on another node.

The protocol between gdb and lld is strictly request-reply, and does not require reliable communication. Lld is stateless, or as close to stateless as possible. A debugger can be attached to any instance of the kernel at any time. Lld is also very simple, by design. It was the first portion of the kernel to be written, and has proven extremely useful. With it we are able, for example, to single-step through interrupt drivers using all the facilities of a high-quality source-level debugger.

One question that arises in the design of a remote debugging facility is where to keep track of the instructions that underlie breakpoints. If breakpoint information is kept on the host machine the target system becomes unusable if the debugger crashes. Topaz therefore maintains its breakpoint information in the debugging stub on the target. The guiding philosophy behind this decision is that it should always be possible to debug, so long as the debugging stub remains intact. For the sake of simplicity, we initially kept our breakpoints on the Sun. Lld tended to break more often than gdb anyway, and only infrequently did we find ourselves unable to continue debugging because of lost information. As the kernel has become more stable and our debugging needs more sophisticated, this situation has begun to change. Particularly annoying is the fact that the kernel cannot be restarted if its code has been corrupted by breakpoint trap

instructions. We are now in the process of moving breakpoint data into lld. Only the underlying instructions will be maintained; associated conditions, commands, enable status, etc. will still be kept in gdb.

We are also currently working on mechanisms to extend the benefits of remote debugging up into user-level programs. Of particular interest as a research issue is the appropriate *focus* of debugging. In Psyche, a human user may wish to debug a process, a realm, or (more nebulously) an entire application. We expect special facilities to be needed to address these different views. When debugging a process, for example, breakpoint traps should be ignored when encountered by a process not currently at the focus of attention. Since breakpoints manifest themselves as kernel traps, the kernel will need to share more semantic information with user-level debuggers than is customary in traditional debugging systems.

6. Conclusion

An evaluation of the Psyche user interface from the programmer's point of view will depend on experience with applications. An evaluation of its implications for kernel performance will require more tuning and measurement than we have been able to undertake to date. We intend to focus in particular on the cost of protected procedure calls, page fault handing (which subsumes communication as well as virtual memory in Psyche), and the generation of upcalls for events such as I/O and timer expiration.

In the implementation, we have been happy with the modularity and structure afforded by the symmetric, shared memory organization of the kernel and the use of C++. We have also found that the layering of the VM system makes it relatively easy to understand and modify. Remote debugging at the lowest levels of the kernel has been extremely valuable, as have the mechanisms for Ethernet loading and software kernel restart.

Some of the costs of our implementation decisions have yet to be fully evaluated. One potential source of overhead is the frequent use of locks for synchronization of access to data structures shared between nodes. Another is the memory management context switches induced by the two-address-space structure of the kernel. A third is the propagation of page faults through an explicitly layered VM system. Each of these will be the focus of study as the kernel matures.

Partly as a result of our experience with previous versions of the BBN Butterfly [3] and partly as a result of our work to date on Psyche, we are able to say a number of things about the design of the Butterfly Plus. We are pleased with the machine in most regards. It is the only commercially-available shared-memory MIMD multiprocessor that will scale to large numbers of nodes. In our estimation, this makes it the most attractive machine on the market for research in parallel operating systems.

The Butterfly displays no noticeable switch contention, though memory hot spots of course present a problem. The I/O potential of the VME adapter is very good — much better than that of the processor-local Multibus adapter. It would be useful to be able to perform DMA directly from the VME bus into the memory of individual processors. As currently designed, data must be copied out of the VME adapter explicitly.

The Motorola 68851 memory management unit is extremely flexible, but suffers from a few annoying problems. Its hierarchical page tables provide very good support for Psyche-style sparse address spaces, but its use of physical addresses for page table pointers makes it inconvenient to walk the tree manually, and makes paging of page tables essentially impossible. Another serious problem for Psyche is that memory cannot be made readable in user mode and both readable and writable in supervisor mode without duplicating page table entries. Finally, a deficiency in the handling of TLB misses during read-modify-write cycles can lead to bus errors when performing 68020 atomic operations. It is cumbersome to handle these in software.

The Butterfly does not support remote invocation of 68020 atomic operations. It provides its own collection of remote atomic fetch-and-phi operations in microcode, and these have proven very useful, though incomplete. A few of the more useful primitives (32-bit fetch-and-store, for example) are missing. Special functions such as the atomic operations and Multibus I/O are invoked by reading and writing special *virtual* locations. This mechanism allows the entire physical address space to be reserved for genuine memory, but introduces a level of memory management complexity that we would have been glad to avoid.

Recent developments in Psyche include the implementation of a simple command shell, remote file access via Ethernet, the VME driver, and a linker/loader for user programs. We expect soon to demonstrate a fully-functional kernel by executing our first robotics application, a balloon juggling program that uses VME and Ethernet communication to control our robot eyes and arm. At that point, we plan to suspend the development of new facilities for a short time while we reorganize and evaluate our current implementation.

References

[1] LeBlanc, T. J., B. D. Marsh, and M. L. Scott, "Memory Management for Large-Scale NUMA Multiprocessors," Technical Report, Computer Science Department, University of Rochester, 1989.

[2] LeBlanc, T. J., J. M. Mellor-Crummey, N. M. Gafter, L. A. Crowl, and P. C. Dibble, "Experiences in the Development of a Multiprocessor Operating System," *Software — Practice and Experience*, to appear, 1989.

[3] LeBlanc, T. J., M. L. Scott, and C. M. Brown, "Large-Scale Parallel Programming: Experience with the BBN Butterfly Parallel Processor," *Proceedings of the ACM SIGPLAN PPEALS 1988 — Parallel Programming: Experience with Applications, Languages, and Systems*, 19-21 July 1988, pp. 161-172.

[4] Redell, D., "Experience with Topaz TeleDebugging," *Proceedings, ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, 5-6 May 1988, pp. 35-44. In *ACM SIGPLAN Notices 24*:1 (January 1989).

[5] Scott, M. L., T. J. LeBlanc, and B. D. Marsh, "Design Rationale for Psyche, a General-Purpose Multiprocessor Operating System," *Proceedings of the 1988 International Conference on Parallel Processing*, 15-19 August 1988, pp. 255-262, vol. II – Software.

[6] Scott, M. L., T. J. LeBlanc, and B. D. Marsh, "Evolution of an Operating System for Large-Scale Shared-Memory Multiprocessors," Technical Report, Computer Science Department, University of Rochester, 1989.

[7] Wilson, A. W. Jr., "Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors," *Fourteenth Annual International Symposium on Computer Architecture*, 2-5 June 1987, pp. 244-252.

[8] Young, M., A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron, "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System," *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, 8-11 November 1987, pp. 63-76. In *ACM Operating Systems Review 21*:5.