A Caching Model of Operating System Kernel Functionality

David R. Cheriton and Kenneth J. Duda
Computer Science Department
Stanford University
Stanford, CA 94025
{cheriton,kjd}@cs.stanford.edu

Abstract

Operating system research has endeavored to develop micro-kernels that provide modularity, reliability and security improvements over conventional monolithic kernels. However, the resulting kernels have been slower, larger and more error-prone than desired. These efforts have also failed to provide sufficient application control of resource management required by sophisticated applications.

This paper describes a caching model of operating system functionality as implemented in the *Cache Kernel,* the supervisor-mode component of the V++ operating system. The Cache Kernel caches operating system objects such as threads and address spaces just as conventional hardware caches memory data. User-mode *application kernels* handle the loading and writeback of these objects, implementing application-specific management policies and mechanisms. Experience with implementing the Cache Kernel and measurements of its performance on a multiprocessor suggest that the caching model can provide competitive performance with conventional monolithic operating systems, yet provides application-level control of system resources, better modularity, better scalability, smaller size and a basis for fault containment.

1 Introduction

Micro-kernels to date have not provided compelling advantages over the conventional monolithic operating system kernel for several reasons.

First, micro-kernels are larger than desired because of the complications of a modern virtual memory system (such as the copy-on-write facility), the need to support many different hardware devices, and complex optimizations in communication facilities, all of which have been handled inside most micro-kernels. Moreover, performance problems have tended to force services originally implemented on top of a micro-kernel back into the kernel, increasing its size. For example, the Mach inter-machine network server has been added back into some versions of Mach for this reason.

Second, micro-kernels do not support domain-specific resource allocation policies any better than monolithic kernels, an increasingly important issue with sophisticated applications and application systems. For example, the standard page-replacement policies of UNIX-like operating systems perform poorly for applications with random or sequential access [17]. Placement of conventional operating system kernel services in a micro-kernel-based server does not generally give the applications any more control because the server is a fixed protected system service. Adding a variety of resource management policies to the micro-kernel fails to achieve the efficiency that application-specific knowledge allows and increases the kernel size and complexity.

Finally, micro-kernels are bloated with exception-handling mechanisms for the failure and unusual cases that can arise with the hardware and with other server and application modules. For example, the potential page-in exception conditions with external pagers introduces complications into Mach.

In this paper, we present an alternative approach to kernel design based on a caching model, as realized in the V++ Cache Kernel. The *V++ Cache Kernel* caches the active objects associated with the basic operating system facilities, namely the address spaces and threads associated with virtual memory, scheduling and IPC. In contrast to conventional micro-kernel design, it does not fully implement all the functionality associated with address spaces and threads. Instead, it relies on higher-level *application kernels* to provide the management functions required for a complete implementation, including the loading and writeback of these objects to and from the Cache Kernel. For example, on a page fault, the application kernel associated with the faulting thread loads a new page mapping descriptor into the Cache Kernel as part of a cached address space object. This new descriptor may cause another page mapping descriptor to be written back to another application kernel to make space for the new descriptor. Because the application kernel selects the physical page frame to use, it

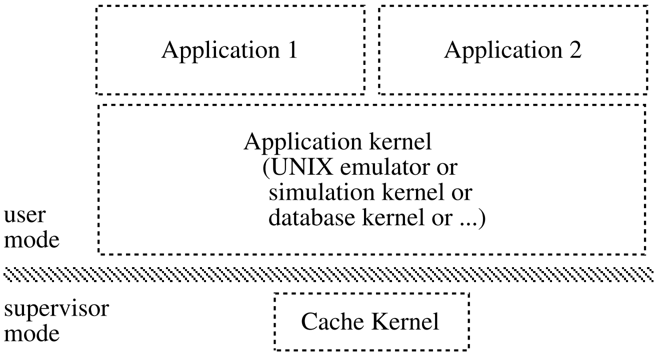


Figure 1: Overall System Architecture in V++

fully controls physical page selection, the page replacement policy and paging I/O.

The following sections argue that this caching model reduces supervisor-level complexity, provides application control of resource management and provides application control over exception conditions and recovery, addressing the problems with micro-kernel designs to date (including a micro-kernel that we developed previously [4]).

The next section describes the Cache Kernel programming interface, illustrating its use by describing how an emulator application kernel would use this interface to implement standard UNIX-like services. Section 3 describes how sophisticated applications can use this interface directly by executing as part of their own application kernel. Section 3 describes how resources are allocated among competing applications. Section 4 describes our Cache Kernel implementation, and Section 5 describes its performance, which appears to provide competitive performance with conventional monolithic kernels. Section 6 describes previous research we see as relevant to this work. We close with a summary of the work, our conclusions and some indication of future directions.

2 The Cache Kernel Interface

In a Cache-Kernel-based system, one or more application kernels execute in user mode on top of the supervisor-mode Cache Kernel, as illustrated in Figure 1. Applications execute on top of the application kernel, either in separate address spaces or the same address space as the application kernel. For example, application 1 and 2 in the figure may be executing on top of, but in separate address spaces from, a UNIX kernel emulator.

The Cache Kernel acts as a cache for three types of operating system objects: address spaces, threads and kernels. It holds the descriptors for the active subset of these objects, executing the performance-critical actions on these objects. The rest of the service functionality typical in a modern operating system (*e.g.,* virtual memory and scheduling) is implemented in application kernels. The application kernel also provides backing store for the object state when it is unloaded from the Cache Kernel, just as data in a conventional cache has a backing memory area. For example, each application kernel maintains a descriptor for each of its threads, loads a thread descriptor into the Cache Kernel to make the thread a candidate for execution, and saves the updated state of that thread when the thread is written back to it.

The primary interface to the Cache Kernel consists of operations to load and unload these objects, signals from the Cache Kernel to application kernels that a particular object is missing, and writeback communication to the application kernel when an object is displaced from the Cache Kernel by the loading of another object.

Each loaded object is identified by an object identifier, returned when the object is loaded. This identifier is used to specify the object when another object is loaded that depends on it. For example, when a thread is loaded, its address space is specified by the identifier returned from the Cache Kernel when the corresponding address space object was loaded. If this identifier fails to identify a valid address space, such as can arise if the address space object is written back concurrently with the thread being loaded, the thread load operation fails, and the application kernel retries the thread load after reloading the address space object. Application kernels do not use the Cache Kernel object identifiers except across this interface because a new identifier is assigned each time an object is loaded. For example, the UNIX emulator provides a "stable" UNIX-like process identifier that is independent of the Cache Kernel address space and thread identifiers which may change several times over the lifetime of the UNIX process.

A small number of objects can also be *locked* in the Cache Kernel, protected from writeback. Locked objects are used to ensure that the application page fault handlers, schedulers and trap handlers execute and do not themselves incur page faults.

The following subsections describe this interface in more detail, illustrating its use by describing how an emulator application kernel would use this interface to implement UNIX-like operating system kernel services.

2.1 Address Space Objects

The Cache Kernel caches an address space object for each active address space. The address space state is stored as a root object and a collection of per-page virtual-to-physical memory mappings. The page mappings, one per mapped page, specify some access flags, a virtual address and the corresponding physical address.

An address space object is loaded by its application kernel with minimal state (currently, just the lock bit), returning a Cache Kernel identifier for the address space object. This identifier is used to specify this object for unloads, references and various query/modify operations. As an illustration of use, the UNIX emulation kernel executes a new process by loading an address space object into the Cache Kernel for the new process to run in and a new thread descriptor to execute this program. Its own data structures for the process record the Cache Kernel identifiers for the address space and thread objects as well as management information associated with the process, such as the bindings of virtual addresses to the program's code and data, which are typically contained in a file. The emulator may then explicitly load some per-page memory mappings for the new process or simply load them on demand, as described below.

When a new address space object is loaded, the Cache Kernel may write back another address space object to make space available for the new object. Before an address space object is written back, all the page mappings in the address space and all the associated threads are written back. For example, in response to address space writeback, the UNIX emulator (application kernel) marks the corresponding address space object as "unloaded," indicating that it must be loaded before the process it contains can be run again.

The page mappings associated with an address space object are normally loaded on demand in response to page faults. When a thread accesses a virtual address for which no mapping is cached, the Cache Kernel delivers a mapping fault to the kernel that owns the address space (and thread(s) contained therein), following the steps illustrated in Figure 2. In step 1, the hardware traps to the Cache Kernel access error handler. The handler stores the state of the faulting thread in its thread descriptor, switches the thread's address space to the thread's application kernel's address space, switches the thread's stack pointer to an exception stack provided by the application kernel, and switches the program counter to the address of the application kernel's page fault handler, which is specified as an attribute of the kernel object corresponding to the application kernel. In step 2, the access error handler causes the thread to start executing the application-kernel-level page fault handler. The faulting address and the form of access (read or write) are communicated as parameters to the page fault handler. In step 3, the application kernel page fault handler navigates its virtual memory data structures, possibly locating a free page frame and reading the page from backing store. It constructs a page mapping descriptor and loads it into the Cache Kernel in step 4. (Alternatively, it may send a UNIX-style SEGV signal to the process. In this latter case, it resumes the thread at the address corresponding to the user-specified UNIX signal handler.) The loading of a new page descriptor may cause another page descriptor to be written back to the associated application kernel in order to make space for the new descriptor, the same as previously described for address space descriptors. In step 5, the faulting thread informs the Cache Kernel that exception processing is complete. The Cache Kernel then restores the stack pointer, program counter, and a few other registers, and resumes the thread in step 6. As an optimization, there is a special Cache Kernel call that both loads a new mapping and returns from the exception handler. To provide protection, the physical address and the access that the application kernel can specify in a new mapping are restricted by its authorized access to physical memory, as recorded in its corresponding kernel object loaded in the Cache Kernel.

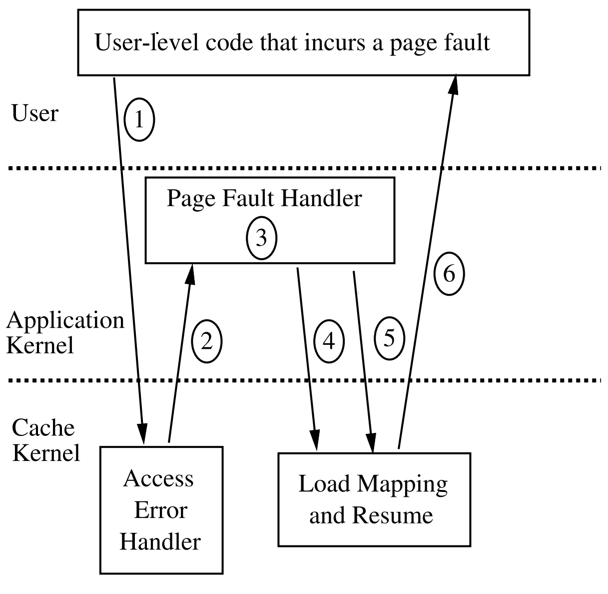


Figure 2: Page Fault Handling

Other exceptions are forwarded to the application kernel by the same mecahnism. In particular, exceptions arise from writing to a read-only page (protection fault), attempting to execute a privileged-mode instruction (privilege violation), and accessing a main-memory cache line that is held on a remote node (consistency fault)[1]. The application kernel has complete control of the faulting thread while handling the fault, just as a conventional operating system would. This approach allows the application kernel to handle these exceptions without complicating the Cache Kernel.

A page mapping is written back to the managing application kernel in response to an explicit request, such as when a page frame is reclaimed, as well as in response to another mapping being loaded. The writeback provides current state bits associated with the mapping including the "referenced" and "modified" bits. The application kernel uses this writeback information to update its records about the state of this page in the address space. In particular, it uses the "modified" bit to know whether the page contents need to be written to backing store before the page frame is reused. The page faulting and writeback mechanisms allow the Cache Kernel to cache only the active set of mappings, relying on the application kernel to store the other mappings.

[1]The *consistency fault* mechanism is used to implement a consistency protocol on a cache-line basis for distributed shared memory, providing a finer-grain consistency unit than pages. A consistency trap also occurs if a reference is made to a memory module that has failed.

The application kernel can explicitly unload inactive mappings, reducing the replacement interference on active mappings. For instance, the UNIX emulator may unload an address space descriptor (and thus all its page mappings) when the process is swapped to disk and no longer executing. In expected use, the Cache Kernel provides enough address space descriptors so that replacement interference in the Cache Kernel is primarily on the page mappings, not address space objects.

Page mappings are identified by address space and virtual address or virtual address range. This identification is adequate for mappings and avoids the space overhead of the general object identification scheme, which would require a separate field per page mapping descriptor. The size of page mapping descriptor is minimized because space for these descriptors dominates the space requirements for the Cache Kernel (see Section 5).

2.2 Interprocess Communication

All interprocess and device communication is provided in the caching model by implementing it as an extension of the virtual memory system using *memory-based messaging* [7]. With memory-based messaging, threads communicate through the memory system by mapping a shared region of physical memory into the sender and receiver address spaces, as illustrated in Figure 3. The sending thread

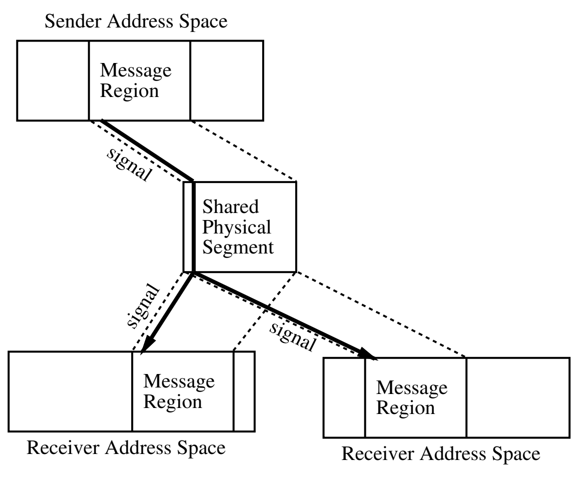


Figure 3: Memory-based Messaging

writes a message into this region and then delivers the address of the new message to the receiving threads as an *address-valued signal*. That is, the virtual address corresponding to the location of the new message is passed to the receiving threads' signal function, translated from the virtual address of the sending thread (using the normal inverted page table support). On receiving the address-valued signal, the receiving thread reads the message at the designated location in the virtual memory region. While the thread is running in its signal function, additional signals are queued within the Cache Kernel.

To support memory-based messaging, the page mappings described in the previous section are extended to optionally specify a *signal thread* and also to specify that the page is in *message mode.* An application kernel interested in receiving signals for a given page specifies a signal thread in the mapping for the page. The signaling uses the same mapping data structures as the rest of the virtual memory system. This extension is simpler than the separate messaging mechanism for interprocess communication that arises with other micro-kernels. Also, the Cache Kernel is only involved in communication setup. The performance-critical data transfer aspect of interprocess communication is performed directly through the memory system. Moreover, with suitable hardware support, there is no software intervention even for signal delivery[2]. Thus, communication performance is limited primarily by the raw performance of the memory system, not the software overhead of copying, queuing and delivering messages, as arises with other micro-kernels.

Memory-based messaging is used for accessing devices controlled by the Cache Kernel. For example, the Ethernet device in our implementation is provided as memory-mapped transmission and reception memory regions. The client thread sends a signal to the Ethernet driver in the Cache Kernel to transmit a packet with the signal address indicating the packet buffer to transmit. On reception, a signal is generated to the receiving thread with the signal address indicating the buffer holding the new packet. This thread demultiplexes the data to the appropriate input stream, similar to conventional network protocol implementations.

Devices that fit into the memory-based messaging model directly require minimal driver code complexity of the Cache Kernel. They also provide the best performance. For example, our own network interface for a 266 Mb Fiber Channel interconnect is designed to fit into this memory-mapped model, and so requires relatively few (276) lines of code for the Cache Kernel driver. In particular, the driver only needs to support memory mapping the special device

[2] The ParaDiGM hardware [7] provides *automatic signal-on-write* to memory in message mode, delivering an *address-valued signal* to each processor managing a signal thread for the page when a thread writes a cache line in message mode. It also provides *message-oriented consistency* on pages in message mode, allowing a processor (presumably the sender) to write a cache line without requiring any "ownership" of the line. When the write completes, the cache controller updates other cached copies of the line as necessary. This specialized consistency minimizes the cache consistency overhead for memory used purely for messaging. The design also calls for hardware delivery of the signal using a per-processor reverse-TLB that maps physical addresses to the corresponding virtual address and signal handler function pairs so there is no software intervention on a reverse-TLB hit. At present, our hardware supports automatic signal generation but not delivery, so the missing portion is emulated in a tightly coded part of the Cache Kernel. Conventional hardware requires software support for generating the signal as well as signal delivery, but this software is a minor extension of the current Cache Kernel mapping mechanisms.

address space corresponding to the network interface. Data transfer and signaling is then handled using the general Cache Kernel memory-based messaging mechanism. The clock is also designed to fit this memory-based messaging model. In contrast, the Ethernet device requires a non-trivial Cache Kernel driver to implement the memory-based messaging interface because the Ethernet chip itself provides a conventional DMA interface.

An object-oriented RPC facility implemented on top of the memory-based messaging as a user-space communication library allows applications and services to use a conventional procedural communication interface to services. For instance, object writeback from the Cache Kernel to the owning application kernel uses a *writeback channel* implemented using this facility. This RPC facility is also used for high-performance communication between distributed application kernels, as described in Section 3. Memory-based messaging supports direct marshaling and demarshaling to and from communication channels with minimal copying and no protection boundary crossing in software. The implementation in user space allows application kernels control over communication resource management and exception handling, by, for example, overriding functions in the communication library.

2.3 Thread Objects

The Cache Kernel caches a collection of thread objects, one for each application kernel thread that should be considered for execution. The thread object is loaded with the values for all the registers and the location of the kernel stack to be used by this thread if it takes an exception (as described in Section 2.1). Other process state variables, such as signal masks and an open file table, are not supported by the Cache Kernel, and thus are stored only in the application kernel. As with address space objects, the Cache Kernel returns an object identifier when the thread is loaded which the application kernel can use later to unload the thread, to change its execution priority, or to force the thread to block. Each thread is associated with an address space which is specified (and must be already loaded) when loading the thread.

The Cache Kernel holds the set of active and response-sensitive threads by mechanisms similar to that used for page mappings. A thread is normally loaded when it is created, or unblocked and its priority makes it eligible to run. It is unloaded when the thread blocks on a long-term event, reducing the contention for thread descriptors in the Cache Kernel. For example, in the UNIX emulation kernel, a thread is unloaded when it begins to sleep with low priority waiting for user input. It is then reloaded when a "wakeup" call is issued on this event. (Reloading in response to user input does not introduce significant delay because the thread reload time (about 230 μs) is short compared to interactive response times.) A thread whose application has been swapped out is also unloaded until its application is reloaded into memory. In this swapped state, it consumes no Cache Kernel descriptors, in contrast to the memory-resident process descriptor records used by the conventional UNIX kernel. A thread being debugged is also unloaded when it hits a breakpoint. Its state can then be examined and reloaded on user request.

A thread that blocks waiting on a memory-based messaging signal can be unloaded by its application kernel after it adds mappings that redirect the signal to one of the application kernel's internal (real-time) threads. The application-kernel thread then reloads the thread when it receives a redirected signal for this unloaded thread. This technique provides on-demand loading of threads similar to the on-demand loading of page mappings that occurs with page faults. A thread can also remain loaded in the Cache Kernel when it suspends itself by waiting on a signal so it is resumed more quickly when the signal arrives. An application kernel can handle threads waiting on short-term events in this way. It can also lock a small number of real-time threads in the Cache Kernel to ensure they are not written back. Retaining a "working set" of loaded threads allows rapid context switching without application kernel intervention.

Using this caching model for threads, an application kernel can implement a wide range of scheduling algorithms, including traditional UNIX-style scheduling. Basically, the application kernel loads a thread to schedule it, unloads a thread to deschedule it, and relies on the Cache Kernel's fixed priority scheduling to designate preference for scheduling among the loaded threads. For example the UNIX emulator per-processor *scheduling thread* wakes up on each rescheduling interval, adjusts the priorities of other threads to enforce its policies, and goes back to sleep. A special Cache Kernel call is provided as an optimization, allowing the scheduling thread to modify the priority of a loaded thread (rather than first unloading the thread, modifying its priority and then reloading it.) The scheduling thread is assured of running because it is loaded at high-priority and locked in the Cache Kernel. Real-time scheduling is provided by running the processes at high priority, possibly adjusting the priority over time to meet deadlines. Co-scheduling of large parallel applications can be supported by assigning a thread per processor and raising all the threads to the appropriate priority at the same time, possibly across multiple Cache Kernel instances, using inter-application-kernel communication.

A thread executing in a separate address space from its application kernel makes "system calls" to its kernel using the standard processor trap instruction. When a thread issues a trap instruction, the processor traps to the Cache Kernel, which then forwards the thread to start executing a trap handler in its application kernel using the same approach as described for page fault handling. This trap forwarding uses similar techniques to those described for UNIX binary emulation [8, 19, 1]. A trap executed by a thread executing in its application kernel (address space) is handled as a Cache Kernel call. An application that is linked directly in the same address space with its application kernel calls its application kernel as a library using normal procedure calls, and invokes the Cache Kernel directly using trap instructions.

The trap, page-fault and exception forwarding mechanisms provide "vertical" communication between the applications and their application kernels, and between the application kernels and the Cache Kernel. That is, "vertical" refers to communication between different levels of protection in the same process or thread, namely supervisor mode, kernel mode and conventional user mode. "Horizontal" communication refers to communication between processes, such as between application kernels and communication with other services and devices. It uses memory-based messaging, as described in the previous subsection.

2.4 Kernel Objects

The Cache Kernel caches a collection of kernel objects, one for each active application kernel. A kernel object designates the application kernel address space, the trap and exception handlers for the kernel and the resources that the kernel has been allocated, including the physical pages the kernel can map, the percentage of each processor the kernel is allowed to use, and the number of locked objects of each type the kernel can load. The address spaces and threads loaded by an application kernel are owned and managed by that application kernel.

For example, the UNIX emulator is represented by a kernel object in the Cache Kernel. Each new address space and thread loaded into the Cache Kernel by the UNIX emulator is designated as owned and managed by the UNIX emulator. Consequently, all traps and exceptions by threads executing in address spaces created by the UNIX emulator are forwarded to the UNIX emulator for handling, as described earlier.

A kernel object is loaded into the Cache Kernel when a new application kernel is executed. Kernel objects are loaded by, and written back to, the first application kernel, which is normally the *system resource manager* described in Section 3. This first kernel is created, loaded and locked on boot. As with all Cache Kernel objects, loading a new kernel object can cause the writeback of another kernel object if there are no free kernel object descriptors in the Cache Kernel. Unloading a kernel object is an expensive operation because it requires unloading the associated address spaces, threads, and memory mappings. The Cache Kernel provides a special set of operations for modifying the resource attributes of a kernel object, as an optimization over unloading a kernel object, modifying the kernel object attributes and reloading it. Currently, there are only three such specialized operations. The use of these operations is discussed further in Section 3.

Writeback of kernel objects is expected to be, and needs to be, infrequent. It is provided because it is simple to do in the Cache Kernel framework, ensures that the system resource manager need runs out of kernel descriptors, such as for large swapped jobs with their own kernels, and provides a uniform model for handling Cache Kernel objects.

This description covers the key aspects of the Cache Kernel interface. Other conventional operating system services are provided at the application kernel level, as illustrated by the UNIX emulator.

A key benefit of the Cache Kernel is that it allows execution of multiple application kernels simultaneously, both operating system emulators as well as application-specialized kernels, as described in the next section. In this mode, it supports system-wide resource management between these separate kernels, as covered in Section 3.

3 Other Application Kernels

A variety of application kernels can be run (simultaneously) on the Cache Kernel. For example, a large-scale parallel scientific simulation can run directly on top of the Cache Kernel to allow application-specific management of physical memory [16] (to avoid random page faults), direct access to the memory-based messaging, and application-specific processor scheduling to match program parallelism to the number of available processors. For example, we have experimented with a hypersonic wind tunnel simulator, MP3D [6], implemented using the particle-in-cell technique. This program can use hundreds of megabytes of memory, parallel processing and significant communication bandwidth to move particles when executed across multiple nodes and can significantly benefit from careful management of its own resources. For example, it can identify the portion of its data set to page out to provide room for data it is about to process. Similarly, a database server can be implemented directly on top of the Cache Kernel to allow careful management of physical memory for caching, optimizing page replacement to minimize the query processing costs. Finally, a real-time embedded system can be realized as an application kernel, controlling the locking of threads, address spaces and mappings into the Cache Kernel, and managing resources to meet response requirements.

An application kernel is any program that is written to interface directly to the Cache Kernel, handling its own memory management, processing management and communication. That is, it must implement the basic system object types and handle loading these objects into, and processing writeback from, the Cache Kernel. Moreover, to be efficient, it must be able to specialize the handling of these resources to the application requirements and behavior.

A C++ class library has been developed for each of the resources, namely memory management, processing and communication. These libraries allow applications to start with a common base of functionality and then specialize, rather than provide all the required mechanism by itself. Application kernels can override general-purpose resource management routines in these libraries with more efficient application-specific ones. They can also override exception handling routines to provide application-specific recovery mechanisms.

The memory management library provides the abstraction of physical segments mapped into virtual memory regions, managed by a segment manager that assigns virtual addresses to physical memory, handling the loading of mapping descriptors on page faults. It bears some similarity to the library described by Anderson et al. [13]. The processing library is basically a thread library that schedules threads by loading them into the Cache Kernel rather than by using its own dispatcher and run queue. A communication library supports channels and channel management on top of the memory-based messaging, and interfaces to the stub routines of the object-oriented RPC facility mentioned earlier.

At the time of writing, we have implemented a simple subset of MP3D and a basic communication server using these libraries. In each of these cases, the application executes directly in the application kernel address space. We also have an initial design of a UNIX emulator, in which applications run in a separate address space from the application kernel for protection. We are also working to integrate a discrete-event simulation library we developed previously with these computational framework libraries. This simulation library provides temporal synchronization, virtual space decomposition of processing, load balancing and cache-architecture-sensitive memory management.

By allowing application control of resource management and exception handling, the Cache Kernel provides the basis for a highly scalable general-purpose parallel computer architecture that we have been developing in the ParaDiGM [5] project. The ParaDiGM architecture is illustrated in Figure 4. Each *multiprocessor module* (MPM) is a self-contained unit with a small number of processors, second-level cache and high-speed network interfaces, executing its own copy of the Cache Kernel out of its PROM and local memory. The high-speed network interfaces connect each MPM to other similarly configured processing nodes as well as to shared file servers. A shared bus connects the MPM to others in the same chassis and to memory modules.

The separate Cache Kernel per MPM limits the degree of parallelism that the Cache Kernel needs to support to the number of processors on one MPM, reducing contention for locks and eliminating the need for complex locking strategies. The MPM also provides a natural unit for resource management, further simplifying the Cache Kernel. Finally, the separate Cache Kernel per MPM provides a basis for fault-containment. A Cache Kernel error only disables its MPM and an MPM hardware failure only halts the local

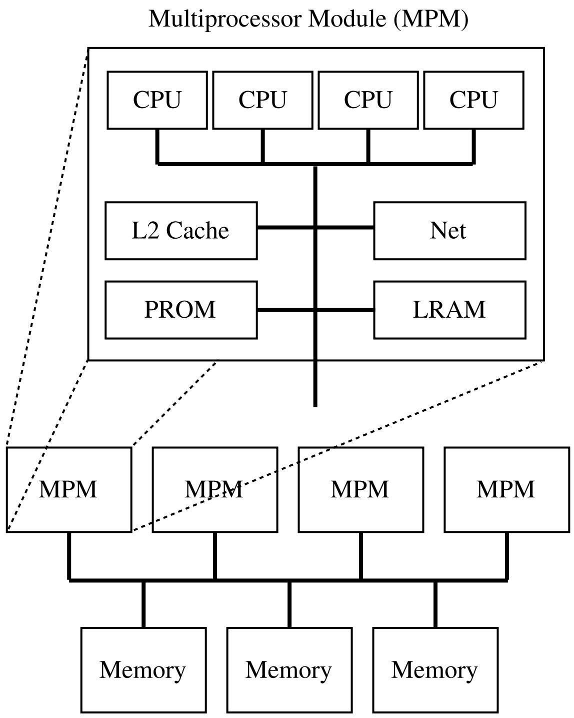


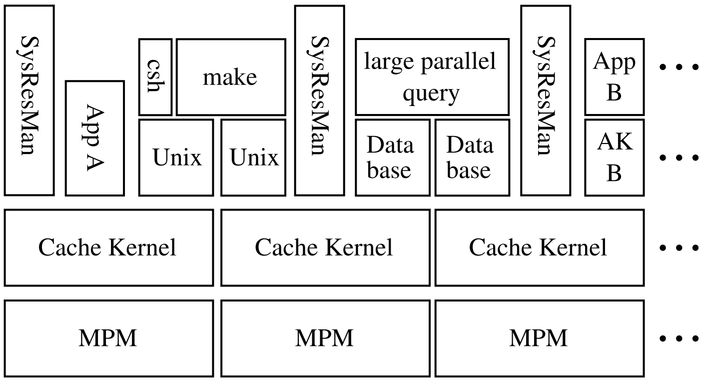
Figure 4: ParaDiGM Architecture

Cache Kernel instance and applications running on top of it, not the entire system. That is, a failure in one MPM does not need to impact other kernels. Explicit coordination between kernels, as required for distributed shared memory implementation, is provided by higher-level software.

The software architecture built on the ParaDiGM hardware architecture is illustrated in Figure 5. A sophisticated application can be distributed and replicated across several nodes, as suggested by the database query in the figure. The application can be programmed to recover from failures by restarting computations from a failed node on different nodes or on the original node after it recovers. One of our current challenges is extending the application kernel resource management class libraries to provide a framework for exception handling and recovery, facilitating the development of applications that achieve fault-tolerance on the basis provided by the Cache Kernel.

A variety of applications, server kernels and operating system emulators can be executing simultaneously on the same hardware as suggested in Figure 5. A special application kernel called the *system resource manager* (SRM), replicated one per Cache Kernel/MPM, manages the resource sharing between other application kernels so that they can share the same hardware simultaneously without unreasonable interference. For example, it prevents a rogue application kernel running a large simulation from disrupting the execution of a UNIX emulator providing timesharing services running on the same ParaDiGM configuration.

The SRM is instantiated when the Cache Kernel boots,



SysResMan	System resource manager
App A	Specialized application (scientific simulation)
Unix	Operating system emulator kernel
Database	Database server kernel
AK B	Kernel for specialized application B
App B	Specialized application B (user process)

Figure 5: Software Architecture

with its kernel descriptor specifying full permissions on all physical resources. It acts as the owning kernel for the other application kernel address spaces and threads as well as the application kernel objects themselves, handling writeback for these objects. The SRM initiates the execution of a new application kernel by creating a new kernel object, address space, and thread, granting an initial resource allocation, bringing the application's text and data into the address space, and loading these objects into the Cache Kernel. Later, it may swap the application kernel out, unloading its objects and saving its state on disk.

The SRM allocates processing capacity, memory pages and network capacity to application kernels. Resources are allocated in large units that the application kernel can then suballocate internally. Memory allocations are for periods of time from multiple seconds to minutes, chosen to amortize the cost of loading and unloading the memory from disk. Similarly, percentages of processors and percentages of network capacity are allocated over these extended periods of time rather than for individual time slices.

The SRM communicates with other instances of itself on other MPMs using the RPC facility, coordinating to provide distributed scheduling using techniques developed for distributed operating systems. In this sense, the SRM corresponds to the "first team" in V [4]. The SRM is replicated on each MPM for failure autonomy between MPMs, to simplify the SRM management, and to limit the degree of parallelism, as was discussed with other application kernels above. Our overall design calls for protection maps in the memory modules, so an MPM failure cannot corrupt memory beyond that managed by the SRM/Cache Kernel/MPM unit that failed. Application kernels that run across several MPMs can be programmed to recover from individual MPM failures, as mentioned earlier.

In contrast to the general-purpose computing configurations supported by the SRM, a single-application configuration, such as real-time embedded control, can use a single application kernel executed as the first kernel. This application kernel, with the authorization to control resources of the first kernel, then has full control over system resources.

4 Internal Design Issues

The Cache Kernel has been implemented in C++ and is running on our multiprocessor ParaDiGM hardware. This hardware prototype uses an MPM with four Motorola 68040 processors running at 25Mhz, two megabytes of local memory and 512 kilobytes of PROM. The Cache Kernel manages four to eight megabytes of high-speed software-controlled second-level cache per MPM that is shared by all four processors, connecting to third-level memory and other MPMs using VMEbus. Each MPM also has two 266 Mb fiber optic channel connections, providing high-speed communication to other MPMs not on the same VMEbus. Although this hardware is not the highest performance at this time, it does provide interesting architectural support for our operating system research, including hardware support for memory-based messaging, hierarchical software-controlled caching, local memory and PROM per MPM, direct connection of high-speed networking to the second-level cache through the memory-based messaging facility, and cache-based locking support.

The Cache Kernel code is burned into PROM on each MPM together with a conventional PROM monitor and network boot program. It executes in supervisor mode with all its data structures in the local RAM of the MPM. The memory mapping is set to protect the Cache Kernel against corruption from application programs.

This section describes three key design issues that we encountered in its implementation, namely efficient mapping support, the object cache replacement mechanism and resource allocation control.

4.1 Mapping Data Structures

The Cache Kernel must efficiently support a large number of memory mappings to allow application kernels to map large amounts of memory with minimal overhead. The mapping needs to be space-efficient because they are stored in memory local to each instance of the Cache Kernel. The mappings must also support specification of a signal process and copy-on-write, although these occur with only a small percentage of the mappings. To meet these requirements, the information from a page mapping is stored across several data structures when it is loaded into the Cache Kernel.

The virtual-to-physical mapping is stored in conventionally structured page tables, one set per address space and logically part of the address space object. The mapping's flags, such as the writable and cachable bits, are also stored in the page table entry. The current implementation uses Motorola 68040 page tables as dictated by the hardware. However, this data structure could be adapted for use with a processor that requires software handling of virtual-to-physical translation, such as the MIPS requires on a TLB miss.

The physical-to-virtual mapping is stored in a physical memory map, using 16-byte descriptors per page, specifying the physical address, the virtual address, the address space and a hash link pointer. The physical memory map is used to delete all mappings associated with a given physical page as part of page reclamation as well as to determine all the virtual addresses mapping to a given physical page as part of signal delivery. The specifications of signal thread and source page for a copy-on-write for a page, if present, are also stored as similar descriptors in this data structure. This data structure is viewed as recording dependencies between objects, the physical-to-virtual dependency being a special but dominant case. That is, the descriptor is viewed as specifying a key, the dependent object and the context, corresponding to the physical address, virtual address and address space in the case of the physical-to-virtual dependency. A signal thread is recorded as a dependency record with the address of the physical-to-virtual mapping as the key, a pointer to the signal thread as the dependent, and a special signal context value as the context. Locating the threads to which a signal on a given physical page should be delivered requires looking up the physical-to-virtual dependency records for the page, and then looking up the signal dependency records for each of these records. A similar approach is used to record copy-on-write mappings.

This approach to storing page mapping information minimizes the space overhead because the common case requires 16 bytes per page plus a small overhead for the page tables. However, it does impose some performance penalty on signal delivery, given the two lookups required in this approach.

To provide efficient signal delivery in the common case, a per-processor reverse-TLB is provided that maps physical addresses to the corresponding virtual address and signal handler function pairs. When the Cache Kernel receives a signal on a given physical address, each processor that receives the signal checks whether the physical address "reverse translates" according to this reverse TLB. If so, the signal is delivered immediately to the active thread. Otherwise, it uses the two-stage lookup described above. Thus, signal delivery to the active thread is fast and the overhead of signal delivery to the non-active thread is more, but is dominated by the rescheduling time to activate the thread (if it is now the highest priority). The reverse-TLB is currently implemented in software in the Cache Kernel but is feasible to implement in hardware with a modest extension to the processor, allowing dispatch of signal-handling to the active thread with no software intervention.

As mentioned earlier, the ParaDiGM hardware provides a number of extensions that the Cache Kernel takes advantage of for performance. However, the Cache Kernel is designed to be portable across conventional hardware. These extensions are relatively easy to omit or provide in software and have relatively little impact on performance, especially with uniprocessor configurations.

4.2 Object Replacement

The Cache Kernel requires a more complex replacement mechanism than a conventional data cache because the objects it caches have relationships among themselves, between themselves and the hardware, and internally to each object. For example, when an address space is replaced in the Cache Kernel and written back to its application kernel, all of its associated threads must also be unloaded and written back. (The alternative of allowing a loaded thread to refer to a missing address space was considered but was rejected as being too complicated, error-prone, and inefficient.) The relationships between objects and the hardware must also be managed carefully. For example, when unloading an address space, the mappings associated with that address space must be removed from the hardware TLB and/or page tables. Similarly, before writing back an executing thread, the processor must first save the thread context and context-switch to a different thread. Objects also have a more complex structure than the typical fixed-size cache line. For example, an address space is represented as a variable number of page table descriptors linked into a tree, providing efficient virtual-to-physical address mapping. Thus, loading and unloading these objects requires several actions and careful synchronization to ensure that the object is loaded and unloaded atomically with respect to other modules in the Cache Kernel and the application kernels.

Figure 6 shows the dependencies between Cache Kernel objects. The arrows in the figure indicate a reference, and therefore a caching dependency, from the object at the tail of the arrow to the object at the head. For example, a signal mapping in the physical memory map references a thread which references an address space which references its owning kernel object. Thus, the signal mapping must be unloaded when the thread, the address space or the kernel is unloaded.

When an object is unloaded, either in response to an explicit application kernel request or as required to free a descriptor in the Cache Kernel to handle a new load request, the object first unloads the objects that directly depend on it. These objects first unload the objects that depend on them, and so on. Locked dependent objects are unloaded the same as unlocked objects. Locking only prevents an object from being unloaded by the object reclamation mechanism when the object and the objects on which it depends are locked. For example, a locked mapping can be reclaimed unless its

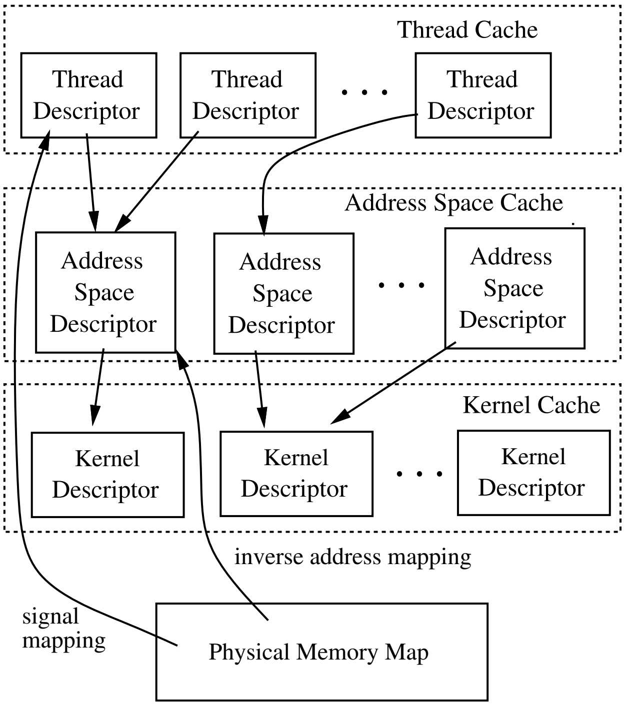


Figure 6: Cached Data Structures

address space, its kernel object and its signal thread (if any) are locked.

The Cache Kernel data structures use non-blocking synchronization techniques so that potentially long unload operations are performed without disabling interrupts or incurring long lock hold times. The version support that is used with the non-blocking synchronization also allows a processor to determine whether a data structure has been modified, perhaps by a unload, concurrently with its execution of a Cache Kernel operation. If it has been modified, the processor retries the operation. For example, a processor loading a new entry into the signal reverse TLB from the physical memory map can check that the version of the map has not changed before adding the entry, and can relookup the mapping if it has.

Memory-based messaging complicates the object replacement mechanism with the need for multi-mapping consistency. *Multi-mapping consistency* ensures that the sender's mapping for a message page is written back if any of the receivers' mappings are written back. This consistency avoids the situation of the sender signaling on the address and the receivers not being notified because their mappings are not loaded in the Cache Kernel. To enforce multi-mapping consistency, the Cache Kernel flushes all writable mappings associated with a physical page frame when it flushes any signal mapping for the page. Each application kernel is expected to load all the mappings for a message page when it loads any of the mappings. Thus, if the mappings are not loaded when the sender writes a message, it generates a mapping trap, causing all the mappings to be loaded. When communication is between threads on separate Cache Kernel instances, the application kernels must coordinate to ensure multi-mapping consistency. Locking of active mappings in the Cache Kernel can be used in this case as part of this coordination. As an alternative to unloading all the mappings, an application kernel can redirect signals to another thread as described in Section 2.2.

4.3 Resource Allocation

The Cache Kernel provides resource allocation enforcement mechanisms to allow mutually distrustful application kernels to execute using shared resources without undue interference.

An application kernel's access to memory is recorded as read and write permission on *page groups* of physical memory. A *page group* is a set of contiguous physical pages starting on a boundary that is aligned modulo the number of pages in the group (currently 128 4k pages). The page group as a large unit of allocation minimizes the space required in the Cache Kernel to record access rights to memory and minimizes overhead allocating memory to application kernels. Using two bits per page group to indicate access, a two-kilobyte memory access array in each kernel object records access to the current four-gigabyte physical address space. Each time a page mapping is loaded into the Cache Kernel, it checks that the access for the specified physical page is consistent with the memory access array associated with the loading kernel. Typically, each kernel has read and write access on a page group or else no access (meaning the memory is allocated to another application kernel). However, we are exploring the use of page groups that are shared between application kernels for communication and shared data, where one or more of the application kernels may only have read access to the memory. As described in Section 3, only the SRM can change the memory access array for a kernel.

The kernel object also specifies the processor allocation that the kernel's threads should receive in terms of a percentage for each processor's time and the maximum priority it can specify for its threads, *i.e.,* its quota. The Cache Kernel monitors the consumption of processor time by each thread and adds that to the total consumed by its kernel for that processor, charging a premium for higher priority execution and a discounted charge for lower priority execution. Over time, it calculates the percentage of each processor that the kernel is consuming. If a kernel exceeds its allocation for a given processor, the threads on that processor are reduced to a low priority so that they only run when the processor is otherwise idle. The graduated charging rate provides an incentive to run threads at lower priority. For example, the UNIX emulator degrades the priority of compute-bound programs to low priority to reduce the effect on its quota when running what are effectively batch, not interactive, programs.

The specification of a maximum priority for the kernel's threads allows the SRM to prevent an application kernel from interfering with real-time threads in another application kernel. For example, a compute-bound application kernel that is executing well under its quota might otherwise use the highest priorities to accelerate its completion time, catastrophically delaying real-time threads. The Cache Kernel also implements time-sliced scheduling of threads at each priority level, so that a real-time thread cannot excessively interfere with a real-time thread from another application executing at the same priority. That is, a thread at a given priority should run after all higher priority threads have blocked (or been unloaded), and after each thread of the same priority ahead of this thread in the queue has received one time slice. This scheme is relatively simple to implement and appears sufficient to allow real-time processing to co-exist with batch application kernels.

I/O capacity is another resource for which controlled sharing between application kernels is required. To date, we have only considered this issue for our high-speed network facility. These interfaces provide packet transmission and reception counts which can be used to calculate network transfer rates. The channel manager for this networking facility in the SRM calculates these I/O rates, and temporarily disconnects application kernels that exceed their quota, exploiting the connection-oriented nature of this networking facility. There is currently no I/O usage control in the Cache Kernel itself.

There is no accounting and charging for cache misses or memory-based message signaling even though these events make significant use of shared hardware resources. For example, a large number of cache misses can significantly load the memory system and the interconnecting bus, degrading the performance of other programs. It may be feasible to add counters to the second-level cache to recognize and charge for this overhead. The premium charged for high priority execution of threads is intended in part to cover the costs of increased context switching expected at high priority. However, to date we have not addressed this problem further.

We have also not implemented quotas on the number of Cache Kernel objects that an application kernel may have loaded at a given time, although there are limits on the number of locked objects. Application kernels simply contend for cached entries, just like independent programs running on a shared physically mapped memory cache. Further experience is required to see if quotas on cache objects are necessary.

5 Evaluation

The Cache Kernel is evaluated from three major standpoints: code size, caching performance, and microbenchmarks of trapping, signaling and page fault handling. The code size measurements indicate the reduction in size is a benefit of the caching approach while the other measurements indicate that the caching model has not detracted from performance over a conventional kernel structure.

5.1 Code Size

The Cache Kernel represents a significant reduction in size and complexity over previous kernels. For example, the virtual memory code in the Cache Kernel is a little under 1,500 lines of C++ code, whereas the V kernel virtual memory support for the same hardware is 13,087 lines of C/C++. The virtual memory system for Ultrix 4.1 for MIPS is 23,400 lines, for SunOS 4.1.2 for Sparc is 14,400 lines, and for Mach for MIPS is a little over 20,000 lines. In total, the Cache Kernel consists of 14,958 lines of C++ code, of which roughly 6000 lines (40 percent) is PROM monitor, remote debugging and booting support (including implementations of UDP, IP, ARP, RARP, and TFTP.) The second-level cache manager software requires a further 1262 lines of code, which would be eliminated if the second-level cache management was implemented entirely in hardware, as is the more conventional approach.

The Cache Kernel has a binary code and data size of 139 kilobytes allowing it to fit easily into the PROM. We expect the size of the Cache Kernel to grow somewhat as we extend and refine its functionality, but do not see it moving beyond 150 kilobytes.

Based on these measurements and our experience writing and debugging the code, we conclude that the caching model, the minimal object approach, and memory-based messaging significantly reduce the size and complexity of kernel code over conventional approaches, including our previous micro-kernel work.

5.2 Caching Performance

The Cache Kernel as a cache of descriptors can be expected to perform well with programs that are reasonably structured, and is not the key performance problem for those that are not, as argued below. First, as shown in Table 1, the size of the descriptors is relatively small, allowing the Cache Kernel to hold enough descriptors to avoid thrashing under reasonable application demands. For instance, our prototype configuration provides 256 thread descriptors for a space cost in local RAM of about 128 kilobytes. (The number designates the maximum number of thread descriptors that can be loaded in the Cache Kernel at one

Object	*Size (bytes)*	*Cache Size*
Kernel	2160	16
AddrSpace	60	64
Thread	532	256
MemMapEntry	16	65536

Table 1: Cache Kernel Object Sizes(bytes)

time.) A system that is actively switching among more than 256 threads is incurring a context switching overhead that would dominate the cost of loading and unloading thread descriptors from the Cache Kernel. With this number of thread descriptors, 64 address space descriptors and 16 kernel descriptors, these descriptors constitute about 10 percent of the 2 megabytes of local memory on our hardware prototype MPM.

Approximately 50 percent of the local RAM is used to store the `MemMapEntry` descriptors listed last in Table 1, providing roughly 65,000 descriptors. Assuming on average at least four cache lines (less than four percent) of each page mapped by these descriptors is accessed, then this number of mapping descriptors covers that accommodated in our 8-megabyte second-level cache[3]. Consequently, software that actively accesses more pages than there are mapping descriptors will thrash the second-level data cache as well as the Cache Kernel memory mappings. Moreover, a program that has poorer page locality than we have hypothesized (*i.e.,* less than four percent usage of pages) also suffers a significant performance penalty from TLB miss behavior on most architectures [3]. For example, we measured up to a 25 percent degradation in performance in the MP3D program mentioned above from processors accessing particles scattered across too many pages. The solution with MP3D was to enforce page locality as well as cache line locality by copying particles in some cases as they moved between processors during the computation. In general, reasonable page locality is necessary for performance, and programs with reasonable page locality execute with minimal replacement interference on page mappings in the Cache Kernel. With programs whose lack of locality leads to extra paging to disk or over the network, the Cache Kernel overhead for loading and unloading mappings is dominated by the page I/O overhead.

The mapping descriptors represent as little as 0.4 percent overhead on the space that they map, so the actual space overhead is acceptable, even considering some duplication of this information at the application kernel level.

The mapping descriptors typically require two to four times the space of the page table descriptors, which are also part of the space overhead. The top-level 512-byte page tables consume a small amount of space because their number is exactly equal to the number of address space descriptors. Assuming reasonable page clustering, the space for the 512-byte second-level tables is also small, bringing the space required for first- and second-level tables to about 5K per address space. Finally, the 256-byte third-level page tables map 64 pages each, *i.e.,* there is one third-level page table for up to 64 16-byte mapping descriptors. Assuming the table is at least half-full, at least two times as much space is used for mapping descriptors as for third-level page tables.

[3]Our hardware has 32-byte cache line size, 8 megabytes of cache, and a page size of 4k (128 lines).

The execution time costs of Cache Kernel object loading and unloading operations are shown in Table 2 for each type of object, with and without writeback occurring. The

Object Types	load No writeback	load Writeback	unload
Mappings	45	145	160
(optimized)	67	167	
Threads	113	489	206
AddrSpaces	101	229	152
Kernel	244	291	80

Table 2: Basic Operations — Elapsed Time in Microseconds

optimized mapping load operation combines loading a new mapping with restarting the faulting thread. This operation is an important optimization for page fault handling.

Cache Kernel loading and writeback overhead can be expected to be minimal in typical system operation. Loading page mappings is expected to be the most common operation, occurring as new page mappings are loaded, and it is also the least expensive. The time to load a mapping could be increased somewhat by the cost of reclamation if reclamation ended up scanning a large range of locked mappings, but the large number of mapping descriptors makes this situation unlikely. The thread loading and unloading corresponds more or less to blocking on long-term events and so occurs infrequently. The loading and unloading of address spaces and kernels typically corresponds to loading and unloading these entities to disk or over the network. Thus, their respective costs are not significant compared to the times associated with these disk/network operations. In the worst case, a kernel descriptor needs to be reclaimed, causing writeback of all the address spaces, threads and mappings associated with the kernel. While this operation can take several milliseconds, it is performed with interrupts enabled and very infrequently. Of course, with a real-time configuration in which objects are locked in the Cache Kernel, the overhead would be essentially zero.

5.3 Trap, Communication and Page Fault Times

The performance of applications using the Cache Kernel is most dependent on the trap handling, signal delivery and page fault handling times.

The cost of a simple trap from a UNIX program to its emulator is 37 microseconds, effectively the cost of a `getpid` operation. This time is 12 microseconds longer than the same operation performed in Mach 2.5 running on a NextStation with a comparable speed processor. For most system calls, the extra trap overhead is insignificant compared to the system call processing time, and that processing time is largely independent of the Cache Kernel.

The time to deliver a signal from one thread to another running on a separate processor is 71 microseconds, composed of 44 microseconds for signal delivery and 27 microseconds for the return from signal handler. These measurements are in agreement with (in fact slightly better than) those reported for a similar implementation of memory-based messaging [7]. Because the communication schemes are identical at the higher levels, and no Cache Kernel involvement occurs on data transfer, the communication using the Cache Kernel is as efficient as the communication in the V implementation of memory-based messaging.

The basic cost of page fault handling is 99 microseconds, which includes 32 microseconds for transfer to the application kernel and 67 microseconds for the optimized mapping load operation. This cost is comparable to the page fault cost of Ultrix and other conventional systems and also comparable to the cost of the comparable operation for external page cache management in the V kernel as described by Harty and Cheriton [16]. A page fault that entails page zeroing, page copying or page read from backing store incurs costs that make the Cache Kernel overhead insignificant. Extrapolating from the application-level performance measured by Harty and Cheriton [16] indicates that performance of applications on the Cache Kernel will be comparable to that of conventional operating systems on the same hardware.

6 Related Work

The Cache Kernel builds on the experience and insight gained in the previous work on micro-kernels, such as V [4] and Mach [21]. As demonstrated in Mach, a dominant contributor to the complexity of these kernels is the virtual memory mechanisms for recording shared mappings, such as shadow object chains. With the Cache Kernel, this complexity has been moved outside of the kernel, cleanly partitioned from the performance-critical mapping supported by the Cache Kernel.

The Cache Kernel interface and facilities for virtual memory support bear some similarity to Mach's "Pmap" interface [11, 15]. However, the Cache Kernel includes additional support for deferred copy as well as page group protection, which was not needed in Mach because the Pmap interface was only an internal interface. The Pmap interface also does not consider multi-mapping consistency support, as required for memory-based messaging. In contrast to the caching of operating system objects in the Cache Kernel, which writes back the objects to untrusted application kernels, KeyKOS [10] writes back objects to protected disk pages. That is, it is only caching the objects in the sense of paging those data structures.

A redesign of Multics [20] proposed the idea of virtual processes that were loaded and saved from a fixed number of "real processes," similar to the thread caching mechanism in the Cache Kernel, but this proposal was never implemented.

Finally, the Cache Kernel exploits memory-based messaging [7] and application-controlled physical memory [16] to minimize mechanism while providing performance and control to sophisticated applications that wish to provide their own operating system kernel. It also builds on experience in implementing binary UNIX emulation [8, 19]. In contrast to Chorus [14], which loads operating system emulator modules directly into the kernel, the Cache Kernel executes its emulators in a separate address space and in non-privileged mode. The lock-free implementation uses similar techniques to that used by Massalin and Pu [18]. These advances together with the caching approach reduce the complexity of the Cache Kernel such that it can be integrated with the PROM for further stability and reliability. They also provide application performance that is competitive with conventional monolithic operating systems.

In contrast to the highly optimized, same-CPU and cross-address space IPC in L3 [12] and KeyKOS [10], the Cache Kernel supports inter-CPU peer-to-peer "horizontal" communication through memory-based messaging. The Cache Kernel trap forwarding facility most closely resembles the sort of same-CPU IPC found in L3, providing efficient transfer of control in the special case of an application communicating with its kernel.

A different approach to application-specific customization is being explored by the SPIN micro-kernel effort [2]. In SPIN, untrusted users write kernel extensions in a pointer-safe language. The extensions are compiled by a trusted compiler and dynamically loaded into the micro-kernel, where they are activated by system events (such as context switch or page fault). They interact with the micro-kernel through protected interfaces but without paying the system call cost. Thus, SPIN allows user modifications to the kernel whereas the Cache Kernel does not. However, with SPIN, the integrity of the micro-kernel is highly dependent on the adequacy of the compiler checking. Customizability is also limited by the set of events one can hook into, and by the expressiveness of the protected interface. Moreover, these user customizations appear to require a complex framework in the micro-kernel, including a supervisor-level garbage collector to reclaim memory allocations made by these extensions and mechanisms to limit resource consumption by these extensions. In contrast, the Cache Kernel is protected from user programming by hardware, does not significantly depend on extended languages and trusted compilers, and implements a relatively simple resource management model, given the simple set of objects it provides. Moreover, the mechanisms in the user class libraries, such as the virtual memory support, are more readily user customizable using the C++ inheritance mechanism.

Like the Cache Kernel, the Aegis exokernel [9] enables application-specific customization through a micro-kernel implementing a minimal machine-dependent interface to the underlying hardware. Like SPIN, Aegis allows untrusted users to extend the supervisor-level portion of the operating system using a variety of techniques to achieve saftey, including code examination, sandboxing, and the use of type-safe languages. Most hardware-level traps are reflected to application-specific trap handlers. For example, an application can install its own TLB miss handler that navigates application-specific page tables. This approach depends on the benefits of application-specific page table structures justifying the cost of ensuring safety in the performance-critical TLB miss handler, and other similar examples.

In overall comparison to these last two competing approaches, the Cache Kernel places a hard protection boundary at a lower level than conventional micro-kernels and exports more control for user customizability while SPIN and Aegis allow controlled entry of supposedly "safe" user software through the protection boundary. We conjecture that a hard protection boundary is required for reliable systems (compilers have enough trouble just generating correct code, never mind checking it for safety), and that the control exported by the caching model is adequate to implement the required application mechanisms. However, further experience is required with all these approaches.

7 Concluding Remarks

The caching model of operating system kernel functionality has produced a small, fast micro-kernel, namely the V++ Cache Kernel, providing system performance that appears competitive with monolithic kernels and well-suited for building robust scalable parallel systems.

As realized in the V++ Cache Kernel, the caching model offers three key benefits. First, the low-level caching interface provides application control of hardware resource management. An application kernel can load and unload objects to implement any desired resource management policy, only relying on the Cache Kernel to handle the loaded active objects (over short time intervals) according to this policy.

Second, the low-level Cache Kernel interface and its forwarding of exceptions to the application kernel allows application-specific exception handling and recovery. The caching approach also means that an application never encounters the "hard" error of the kernel running out of thread or address space descriptors as can occur with conventional systems like UNIX. The Cache Kernel always allows more objects to be loaded, writing back other objects to make space if necessary.

Finally, the caching model has led to a fundamental reduction in the complexity of supervisor mode software compared to prior micro-kernel work, measured by both lines of code and binary code size. The plethora of query and modify operations of conventional operating systems are absent from the Cache Kernel. Instead, the application kernel unloads the appropriate object, examines its state, and, if a modify operation, loads a modified version of that state back into the Cache Kernel. With experience, we are adding a small number of special query and modify operations as optimizations of this basic mechanism, such as a kernel call to modify the page groups associated with a kernel. However, these few optimizations do not significantly increase the size or complexity of the Cache Kernel. The use of memory-based messaging further simplifies the Cache Kernel and minimizes data copying and traps across the kernel protection boundary. We have taken advantage of this smaller size and stable functionality by incorporating the Cache Kernel into the PROM monitor code.

The Cache Kernel's small size allows it to be used economically in embedded real-time systems as well as to be replicated on each node of a large-scale parallel architecture for fault containment. Exploiting the Cache Kernel facilities, sophisticated application kernels can support efficient, robust parallel computation, real-time processing and database management systems while sharing all or part of a multiprocessor with other application kernels.

We are currently developing application kernels and operating system emulators that exploit the Cache Kernel capabilities. In particular, we are developing a simulation kernel (running on the Cache Kernel) that supports applications such as the MP3D wind tunnel simulation [6]. The operating systems emulators, such as one for UNIX, allow simple applications to share the same hardware concurrently with these sophisticated applications. We are also exploring the use of the Cache Kernel and modular application kernels for fault-tolerant scalable parallel and distributed computing, as described in Section 3.

Looking ahead, hardware manufacturers might reasonably provide a Cache-Kernel-like PROM monitor for their future hardware. This approach would allow a wide range of applications and operating systems to use the hardware without introducing as many dependencies on low-level hardware characteristics. The result would be better portability as well as greater freedom for the hardware manufacturers to revise their implementation of the Cache Kernel abstraction. In fact, it would allow independently developed operating systems to execute concurrently on the same hardware, a situation similar to that provided by the *virtual machine* operating system efforts of the 1960's and 70's. However, the Cache Kernel "virtual machine" supports scalable high-performance parallel distributed systems, not just the conventional single processor, single-node configurations of yore.

8 Acknowledgements

This work was sponsored in part by ARPA under US Army contract DABT63-91-K-0001. Equipment and funding by IBM for the ParaDiGM hardware is also gratefully acknowledged. This paper has benefited considerably from the comments of the reviewers, our "shepherds" Brian Bershad and

Willy Zwaenepoel, and various colleagues, including Mary Baker, Fusun Ertemalp, Hugh Holbrook, Michael Greenwald, Tim Mann and Ross Finlayson.

References

[1] B.N. Bershad, T.E. Anderson, E.D. Lazowska, and H.M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990.

[2] B.N. Bershad, C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. Gün Sirer. Spin — an extensible microkernel for application-specific operating system services. University of Washington Computer Science and Engineering Technical Report 94-03-03, February 1994.

[3] J.B. Chen, A. Borg, and N.P. Jouppi. A simulation-based study of TLB performance. In *Proc. 19th Annual Intl. Symposium on Computer Architecture*, pages 114–123. ACM SIGARCH, IEEE Computer Society, May 1992.

[4] D.R. Cheriton. The V distributed system. *Comm. ACM*, 31(3):314–333, March 1988.

[5] D.R. Cheriton, H. Goosen, and P. Boyle. ParaDiGM: A highly scalable shared-memory multi-computer architecture. *IEEE Computer*, 24(2), February 1991.

[6] D.R. Cheriton, H. Goosen, and P. Machanick. Restructuring a parallel simulation to improve shared memory multiprocessor cache behavior: A first experience. In *Shared Memory Multiprocessor Symposium*, pages 23–31. ACM, April 1991.

[7] D.R. Cheriton and R. Kutter. Optimizing memory-based messaging for scalable shared memory multiprocessor architectures. Stanford Computer Science Technical Report CS-93-123, December 1993.

[8] D.R. Cheriton, G.R. Whitehead, and E.W. Sznyter. Binary emulation of UNIX using the V kernel. In *USENIX Summer Conference*. USENIX, June 1990.

[9] D.R. Engler, M.F. Kaashoek, and J.W. O'Toole Jr. The operating system kernel as a secure programmable machine. *Proceedings of the ACM European SIGOPS Workshop*, September 1994.

[10] A.C. Bomberger et al. The KeyKOS nanokernel architecture. In *Proceedings of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*. USENIX, April 1992.

[11] D. Black et al. Translation lookaside consistency: A software approach. In *Proc. 17th Int. Symp. on Computer Architecture*, pages 113–122, April 1989.

[12] J. Liedtke et al. Two years of experience with a micro-kernel based os. *Operating Systems Review*, 25(2):57–62, 1991.

[13] K. Anderson et al. Tools for the development of application-specific virtual memory management. In *OOPSLA*, 1993.

[14] M. Rozier et al. Overview of the CHORUS distributed operating system. In *Proceedings of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*. USENIX, April 1992.

[15] R. Rashid et al. Machine-independent virtual memory management for paged unitprocessor and multiprocessor architectures. *IEEE Trans Comput.*, 37(8):896–908, August 1988.

[16] K. Harty and D.R. Cheriton. Application-controlled physical memory using external page cache management. In *ASPLOS*, pages 187–197. ACM, October 1992.

[17] J. Kearns and S. DeFazio. Diversity in database reference behavior. *Performance Evaluation Review*, 1989.

[18] H. Massalin and C. Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91, Computer Science Department, Columbia University, October 1991.

[19] R. Rashid and D. Goluv. UNIX as an application process. In *USENIX Summer Conference*. Usenix, June 1990.

[20] M. Schroeder, D. Clark, and J. Saltzer. The MULTICS kernel design project. In *Proceedings of the 6th Symposium on Operating Systems Principles*, pages 43–56. ACM, November 1977.

[21] M. Young et al. The duality of memory and communication in the implementation of a multiprocessor operating system. In *11th Symp. on Operating Systems Principles*. ACM, November 1987.