A Caching Model of Operating System Kernel Functionality

David R. Cheriton and Kenneth J. Duda
Computer Science Department
Stanford University

1 Introduction

Operating system design has had limited success in providing adequate application functionality and a poor record in avoiding excessive growth in size and complexity, especially with protected operating systems. Applications require far greater control over memory, I/O and processing resources to meet their requirements. For example, database transaction processing systems include their own "kernel" which can much better manage resources for the application than can the application-ignorant general-purpose conventional operating system mechanisms. Large-scale parallel applications have similar requirements. The same requirements arise with servers implemented outside the operating system kernel.

In our research, we have been exploring the approach of making the operating system kernel a *cache* for active operating systems objects such as processes, address spaces and communication channels, rather than a complete manager of these objects. The resulting system is smaller than recent so-called microkernels, and also provides greater flexibility for applications, including real-time applications, database management systems and large-scale simulations. As part of this research, we have developed what we call a *cache kernel*, a new generation of microkernel that supports operating system configurations across these dimensions.

The cache kernel can also be regarded as providing a hardware adaptation layer (HAL) to operating system services rather than trying to just provide a key subset of OS services, as has been the common approach in previous microkernel work. However, in contrast to conventional HALs, the cache kernel is fault-tolerant because it is protected from the rest of the operating system (and applications), it is replicated in large-scale configurations and it includes audit and recovery mechanisms. A cache kernel has been implemented on a scalable shared-memory and networked multicomputer [1] hardware which provides architectural support for the cache kernel approach.

Fig 1 illustrates a typical target configuration. There is an instance of the cache kernel per multi-processor module (MPM), each managing the processors, second-level cache and network interface of that MPM. The cache kernel executes out of PROM and local memory of the MPM, making it hardware-independent of the rest of the system except for power. That is, the separate cache kernels and MPMs fail independently. Operating system services are provided by application kernels, server kernels and conventional operating system emulation kernels in conjunction with privileged MPM resource managers (MRM) that execute on top of the cache kernel. These kernels may be in separate protected address spaces or a shared library within a sophisticated application address space. A system bus connects the MPMs to each other and the memory modules. A high-speed network interface per MPM connects this node to file servers and other similarly config-

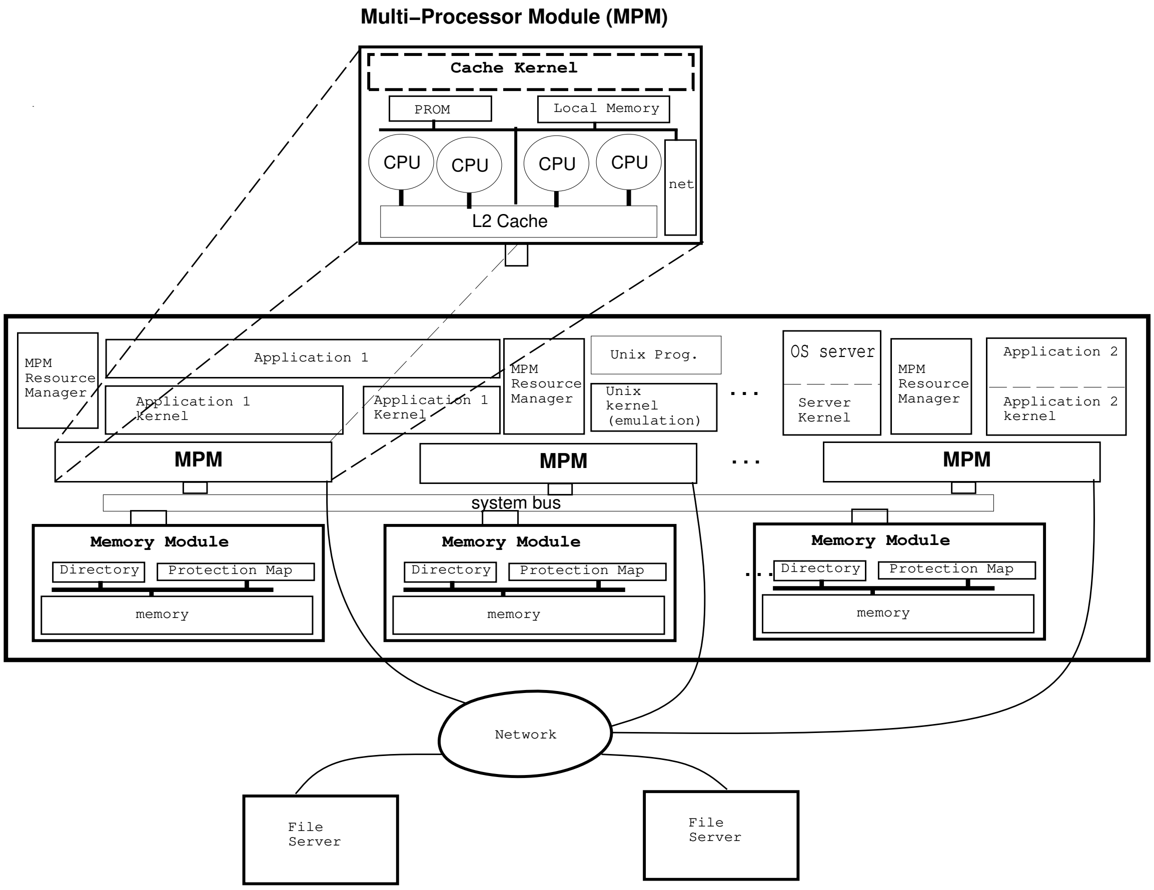

Figure 1: Large-scale Cache Kernel Configuration

ured processing nodes. This overall design can be simplified for real-time applications and similar restricted scenarios. For example, with relatively static partitioning of resources, an embedded real-time application could be structured as one or more application spaces incorporating application kernels as shared libraries executing directly on top of the cache kernel.

2 The Cache Kernel Design

The cache kernel acts as a cache for four major types of operating system objects, namely kernels, address spaces, memory mappings, and processes. The primary cache kernel interface consists of operations to load and unload these objects, signals to the application kernel level that a particular object is missing, and writeback communication to the application kernel level when an object is displaced in the cache kernel by the loading of another object. Any of these system objects that is *active* must be first loaded into the cache kernel, just as for active data must be loaded in a conventional data cache to be accessed. For example, a process must be loaded in the cache kernel to be eligible for execution. Each of these objects is viewed as being implemented outside of the cache kernel, just as data in a conventional cache has a backing memory area where it "really" resides. In the case of the cache kernel, each object is actually implemented in an application kernel that loads the object into the cache kernel when it is to be active, and stores the written back version of the object when it is deactivated. For example, each application kernel maintains the "true" descriptor of each process it has, and loads that process into the cache kernel when appropriate and saves the updated state of that process when the process is written back because of replacement in the cache kernel.

The cache kernel minimizes the number of kernel calls beyond those required for loading and unloading the cached objects it supports. Many of the plethora of query and modify operations that conventional operating systems support are provided with the cache kernel by unloading the appropriate object, examining its state, and, if a modify operation, loading a modified version of that state back into the cache kernel. With experience, we are adding some special kernel query and modify operations as optimizations of this basic mechanism. For example, there is a kernel call to modify the page groups associated with a kernel, given the frequency of this operation and the cost of unloading an application kernel and all its address spaces and processes. This approach minimizes the code in the cache kernel for these operations while providing the same basic performance as a conventional design.

3 Experience to Date

The cache kernel has been implemented on our hardware prototype. This hardware uses an MPM with 4 Motorola 68040 processors running at 25Mhz, 2 megabytes of local memory and 512 kilobytes of PROM. The cache kernel manages 4 to 8 megabytes of high-speed 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 board for the cache kernel and 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 size is currently 13,541 lines (of C++ with some assembly). The time to deliver a signal from one process 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 [3]. The basic cost of page fault handling time is 99 microseconds, which includes 32 microseconds for transfer to the application kernel and 67 microseconds for the `NewMappingReturn` operation. This cost is comparable to the page fault cost 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 [4]. Extrapolating from the application-level performance measured by this previous work indicates that performance of applications on the cache kernel to be comparable to that of conventional operating systems on the same hardware.

4 Concluding Remarks

The caching model of operating system kernel functionality has lead to a microkernel with smaller size and better support for application flexibility than previous design by our group and others. Its facilities allow emulation of conventional operating systems using an "application kernel" that implements the system calls of such a conventional system.

The cache kernel design exploits memory-based messaging and application controlled physical memory to minimize the required mechanism while providing performance and control to sophisticated applications that wish to provide their own kernel. 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.

As part of our future work, we plan to develop sophisticated application kernels and operating system emulators that exploit the cache kernel capabilities, including distribution and application control of hardware resources. In particular, we hope to modify a revised version of the MP3D wind tunnel simulation [2] to execute using a special simulation kernel running on the cache kernel running on our multiprocessor hardware.

References

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

[2] D.R. Cheriton, H.A. Goosen, and P. Machanick. Restructuring a parallel simulation to improve cache behavior in a shared-memory multiprocessor: A first experience. Submitted for publication.

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

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